# The static Keyword

## Static Methods

### Different ways of calling methods

* Usually you call a method by using the “dot operator” (member access operator) on an object, like this:
* Rectangle rect = new Rectangle();  
  rect.SetLength(12);
* The SetLength method is defined in the Rectangle class. In order to call it, we need an *instance* of that class, which in this case is the object rect.
* However, sometimes we have written code where we call a method using the dot operator on the name of a class, not an object. For example, the familiar WriteLine method:
* Console.WriteLine("Hello!");
* Notice that we have never needed to write new Console() to instantiate a Console object before calling this method.
* More recently, we learned about the Array.Resize method, which can be used to resize an array. Even though arrays are objects, we call the Resize method on the Array class, not the particular array object we want to resize:
* int[] myArray = {10, 20, 30};  
  Array.Resize(ref myArray, 6);
* Methods that are called using the name of the class rather than an instance of that class are **static methods**

### Declaring static methods

* Static methods are declared by adding the static keyword to the header, like this:
* class Console  
  {  
   public static void WriteLine(string value)  
   {  
   ...  
   }  
  }
* The static keyword means that this method belongs to the class “in general,” rather than an instance of the class
* Thus, you do not need an object (instance of the class) to call a static method; you only need the name of the class

### static methods and instances

* Normal, non-static methods are always associated with a particular instance (object)
* When a normal method modifies an instance variable, it always “knows” which object to modify, because you need to specify the object when calling it
  + For example, the SetLength method is defined like this:
  + class Rectangle  
    {  
     private int length;  
     private int width;  
     public void SetLength(int lengthParameter)  
     {  
     length = lengthParameter;  
     }  
    }
  + When you call the method with rect.SetLength(12), the length variable automatically refers to the length instance variable stored in rect.
* Static methods are not associated with any instance, and thus **cannot use instance variables**
* For example, we could attempt to declare the ComputeArea method of Rectangle as a static method, but this would not compile:
* class Rectangle  
  {  
   private int length;  
   private int width;  
   public void SetLength(int lengthParameter)  
   {  
   length = lengthParameter;  
   }  
   public static int ComputeArea()  
   {  
   return length \* width;  
   }  
  }
  + To call this static method, you would write Rectangle.ComputeArea();
  + Since no Rectangle object is specified, which object’s length and width should be used in the computation?

### Uses for static methods

* Since static methods cannot access instance variables, they do not seem very useful
* One reason to use them: when writing a function that does not need to “save” any state, and just computes an output (its return value) based on some input (its parameters)
* Math-related functions are usually written as static methods. The .NET library comes with a class named Math that defines several static methods, like these:
* public static double Pow(double x, double y)  
  {  
   //Computes and returns x^y  
  }  
  public static double Sqrt(double x)  
  {  
   //Computes and returns the square root of x  
  }  
  public static int Max(int x, int y)  
  {  
   //Returns the larger of the two numbers x and y  
  }  
  public static int Min(int x, int y)  
  {  
   //Returns the smaller of the two numbers x and y  
  }
* Note that none of them need to use any instance variables.
* Defining several static methods in the same class (like in class Math) helps to group together similar or related functions, even if you never create an object of that class
* Static methods are also useful for providing the program’s “entry point.” Remember that your program must always have a Main method declared like this:
* class Program  
  {  
   static void Main(string[] args)  
   {  
   ...  
   }  
  }
  + When your program first starts, no objects exist yet, which means no “normal” methods can be called
  + The .NET run-time (the interpreter that runs a C# program) must decide what code to execute to make your program start running
  + It can call Program.Main() without creating an object, or knowing anything else about your program, because Main is a static method
* Static methods can be used to “help” other methods, both static and non-static
  + It’s easy to call a static method from within the same class: You can just write the name of the method, without the class name, i.e. MethodName(args) instead of ClassName.MethodName(args)
  + For example, the Array class has a static method named Copy that copies the contents of one array into another array. This makes it very easy to write the Resize method:
  + class Array  
    {  
     public static void Copy(Array source, Array dest, int length)  
     {  
     //Copy [length] elements from source to dest, in the same order  
     }  
     public static void Resize<T>(ref T[] array, int newSize)  
     {  
     T[] newArray = new T[newSize]  
     Copy(array, newArray, Math.Min(array.Length, newSize));  
     array = newArray;  
     }  
    }
  + Since arrays are fixed-size, the only way to resize an array is to create a new array of the new size and copy the data from the old array into the new array. This Resize method is easy to read because the act of copying the data (which would involve a for loop) is written separately, in the Copy method, and Resize just needs to call Copy.
  + Similarly, you can add additional static methods to the class that contains Main, and call them from within Main. This can help you separate a long program into smaller, easier-to-read chunks. It also allows you to re-use the same code multiple times without copying and pasting it.
  + class Program  
    {  
     static void Main(string[] args)  
     {  
     int userNum1 = InputPositiveNumber();  
     int userNum2 = InputPositiveNumber();  
     int part1Result = DoPart1(userNum1, userNum2);  
     DoPart2("Bananas", part1Result);  
     }  
     static int InputPositiveNumber()  
     {  
     int number;  
     bool success;  
     do  
     {  
     Console.WriteLine("Please enter a positive number");  
     success = int.TryParse(Console.ReadLine(), out number);  
     } while(!success || number < 0);  
     return number;  
     }  
     static int DoPart1(int a, int b)  
     {  
     ...  
     }  
     static void DoPart2(string x, int y)  
     {  
     ...  
     }  
    }
  + In this example, our program needs to read two different numbers from the user, so we put the input-validation loop into the InputPositiveNumber method instead of writing it twice in the Main method. It then has two separate “parts” (computing some result with the two user-input numbers, and combining that computed number with a string to display some output), which we write in the two methods DoPart1 and DoPart2. This makes our actual Main method only 4 lines long.

## Static Variables

### Defining static variables

* The static keyword can be used in something that looks like an instance variable declaration:
* class Rectangle  
  {  
   private static int NumRectangles = 0;  
   ...  
  }
* This declares a variable that is stored with the class definition, not inside an object (it is *not* an instance variable)
* Unlike an instance variable, there is only one copy in the entire program, and any method that refers to NumRectangles will access the *same* variable, no matter which object the method is called on
* Since it is not an instance variable, it does not get initialized in the constructor. Instead, you must initialize it with a value when you declare it, more like a local variable (in this case, NumRectangles is initialized to 0).
* It’s OK to declare a static variable with the public access modifier, because it is not part of any object’s state. Thus, accessing the variable from outside the class will not violate encapsulation, the principle that an object’s state should only be modified by that object.
  + For example, we could use the NumRectangles variable to count the number of rectangles in a program by making it public. We could define it like this:
  + class Rectangle  
    {  
     public static int NumRectangles = 0;  
     ...  
    }
  + and use it like this, in a Main method:
  + Rectangle myRect = new Rectangle();  
    Rectangle.NumRectangles++;  
    Rectangle myOtherRect = new Rectangle();  
    Rectangle.NumRectangles++;

### Using static variables

* Since all instances of a class share the same static variables, you can use them to keep track of information about “the class as a whole” or “all the objects of this type”
* A common use for static variables is to count the number of instances of an object that have been created so far in the program
  + Instead of “manually” incrementing this counter, like in our previous example, we can increment it inside the constructor:
  + class Rectangle  
    {  
     public static int NumRectangles = 0;  
     private int length;  
     private int width;  
     public Rectangle(int lengthP, int widthP)  
     {  
     length = lengthP;  
     width = widthP;  
     NumRectangles++;  
     }  
    }
  + Each time this constructor is called, it initializes a new Rectangle object with its own copy of the length and width variables. It also increments the single copy of the NumRectangles variable that is shared by all Rectangle objects.
  + The variable can still be accessed from the Main method (because it is public), where it could be used like this:
  + Rectangle rect1 = new Rectangle(2, 4);  
    Rectangle rect2 = new Rectangle(7, 5);  
    Console.WriteLine(Rectangle.NumRectangles  
     + " rectangle objects have been created");
  + When rect1 is instantiated, its copy of length is set to 2 and its copy of width is set to 4, then the single NumRectangles variable is incremented to 1. Then, when rect2 is instantiated, its copy of length is set to 7 and its copy of width is set to 5, and the NumRectangles variable is incremented to 2.
* Static variables are also useful for **constants**
  + The const keyword, which we learned about earlier, is actually very similar to static
  + A const variable is just a static variable that cannot be modified
  + Like a static variable, it can be accessed using the name of the class where it is defined (e.g. Math.PI), and there is only one copy for the entire program

### Static methods and variables

* Static methods cannot access instance variables, but they *can* access static variables
* There is no ambiguity when accessing a static variable: you do not need to know which object’s variable to access, because there is only one copy of the static variable shared by all objects
* This means you can write a “getter” or “setter” for a static variable, as long as it is a static method. For example, we could improve our NumRectangles counter by ensuring that the Main method can only read it through a getter method, like this:
* class Rectangle  
  {  
   private static int NumRectangles = 0;  
   private int length;  
   private int width;  
   public Rectangle(int lengthP, int widthP)  
   {  
   length = lengthP;  
   width = widthP;  
   NumRectangles++;  
   }  
   public static int GetNumRectangles()  
   {  
   return NumRectangles;  
   }  
  }
  + The NumRectangles variable is now declared private, which means only the Rectangle constructor will be able to increment it. Before, it would have been possible for the Main method to execute something liek Rectangle.NumRectangles = 1; and throw off the count.
  + The GetNumRectangles method cannot access length or width because they are instance variables, but it can access NumRectangles
  + The static method would be called from the Main method like this:
  + Rectangle rect1 = new Rectangle(2, 4);  
    Rectangle rect2 = new Rectangle(7, 5);  
    Console.WriteLine(Rectangle.GetNumRectangles()  
     + " rectangle objects have been created");

### Summary of static access rules

* Static variables and instance variables are both **fields** of a class; they can also be called “static fields” and “non-static fields”
* This table summarizes how methods are allowed to access them:
* | | Static Field | Non-static Field | | :: | :: | :: | | Static method | Yes | No | | Non-static method | Yes | Yes |

## Static Classes

* The static keyword can also be used in a class declaration
* If a class is declared static, all of its members (fields and methods) must be static
* This is useful for classes that serve as “utility libraries” containing a collection of functions, and are not supposed to be instantiated and used as objects
* For example, the Math class is declared like this:
* static class Math  
  {  
   public static double Sqrt(double x)  
   {  
   ...  
   }  
   public static double Pow(double x, double y)  
   {  
   ...  
   }  
  }
* There is no need to ever create a Math object, but all of these methods belong together (within the same class) because they all implement standard mathematical functions.