CHAPTER 14

Static

The `static` keyword can be used to declare fields and methods that can be accessed without having to create an instance of the class. Static (class) members only exist in one copy, which belongs to the class itself, whereas instance (non-static) members are created as new copies for each new object. This means that static methods cannot use instance members since these methods are not part of an instance. On the other hand, instance methods can use both static and instance members.

```
class MyCircle
{

  // Instance variable (one per object)
  public float r = 10F;

  // Static/class variable (only one instance)
  public static float pi = 3.14F;

  // Instance method
  public float GetArea()
  {
    return ComputeArea(r);
  }
```


M. Olsson, *C# 8 Quick Syntax Reference*, https://doi.org/10.1007/978-1-4842-5577-3_14

```
  // Static/class method
  public static float ComputeArea(float a)
  {
    return pi*a*a;
  }
}
```

Accessing Static Members

To access a static member from outside the class, the class name is used followed by the dot operator. This operator is the same as the one used to access instance members, but to reach them, an object reference is required. An object reference cannot be used to access a static member.

```
class MyApp
{
  static void Main()
  {
    float f = MyCircle.ComputeArea(MyCircle.pi);
  }
}
```

Static Methods

The advantage of static members is that they can be used by other classes without having to create an instance of the class. Fields should therefore be declared static when only a single instance of the variable is needed. Methods should be declared static if they perform a generic function that is independent of any instance variables. A good example of this is the `System.Math` class, which provides a multitude of mathematical methods. This class contains only static members and constants.

```
static void Main()
{
  double pi = System.Math.PI;
}
```

Static Fields

Static fields have the advantage that they persist throughout the life of the application. A static variable can therefore be used, for example, to record the number of times that a method has been called.

```
static int count = 0;
public static void Dummy()
{
  count++;
}
```

The default value for a static field will be set only once before it is first used.

Static Classes

A class can also be marked `static` if it only contains static members and constant fields. A static class cannot be inherited or instantiated into an object. Attempting to do so will cause a compile-time error.

```
static class MyCircle {}
```

Static Constructor

A static constructor can perform any actions needed to initialize a class. Typically, these actions involve initializing static fields that cannot be initialized as they are declared. This can be necessary if their initialization requires more than one line, or some other logic, to be initialized.

```
class MyClass
{
  static int[] array = new int[5];
  static MyClass()
  {
    for(int i = 0; i < array.Length; i++)
      array[i] = i;
  }
}
```

The static constructor, in contrast to the regular instance constructor, will only be run once. This occurs automatically either when an instance of the class is created or when a static member of the class is referenced. Static constructors cannot be called directly and are not inherited. In case the static fields also have initializers, those initial values will be assigned before the static constructor is run.

Static Local Functions

A local function automatically captures the context of its enclosing scope, enabling it to reference members outside of itself such as variables local to the parent method.

```
string GetName()
{
  string name = "John";
  return LocalFunc();
  string LocalFunc() { return name; }
}
```

As of C# 8.0, the static modifier can be applied to local functions to disable this behavior. The compiler will then ensure that the static local function does not reference any members outside of its own scope. Limiting access in this way can help simplify debugging, because you will know that the local function does not modify any external variables.

```
string GetName()
{
  string name = "John";
  return LocalFunc(name);
  static string LocalFunc(string s) { return s; }
}
```

Extension Methods

A feature added in C# 3.0 is extension methods, which provide a way to seemingly add new instance methods to an existing class outside its definition. An extension method must be defined as `static` in a static class and the keyword `this` is used on the first parameter to designate which class to extend.

```
static class MyExtensions
{
  // Extension method
  public static int ToInt(this string s) {
    return Int32.Parse(s);
  }
}
```

The extension method is callable for objects of its first parameter type, in this case, `string`, as if it were an instance method of that class. No reference to the static class is needed.

```
class MyApp
{
  static void Main() {
    string s = "10";
    int i = s.ToInt();
  }
}
```

Because the extension method has an object reference, it can use instance members of the class it is extending. However, it cannot use members of any class that are inaccessible due to their access level. The benefit of extension methods is that they enable you to "add" methods to a class without having to modify or derive the original type.