Static Keyword Considerations

In C#, the static keyword is used to declare members of a class that belong to the class itself rather than to any specific instance of the class. When you apply static to a class member, that member is shared among all instances of the class, and it can also be accessed without creating an instance of the class. This is a fundamental concept in many programming languages, including C#, and it helps in managing data and behavior that is common to the class, rather than to individual objects.

Types of Static Members

1. Static Fields: Used to store data that is shared among all instances of the class. For example, you might use a static field to count how many instances of a class have been created or to hold configuration information that applies to all objects of that type.

2. Static Methods: Methods that do not operate on instance data. These methods belong to the class itself and can only access other static members of the class. They are typically used for utility functions that do not require any object state from the class to perform their tasks.

3. Static Properties: Like static methods, these properties are related to the class, not to any instance. They can be used to control access to static data or to provide properties that apply globally to the class.

4. Static Constructors: A special type of constructor that initializes static members of the class. This constructor is called automatically and exactly once, before any static members are accessed or an instance of the class is created.

5. Static Classes: You can also declare an entire class as static. This is often done for utility classes where you want to group related static methods (like a mathematical utility class) without allowing instantiation of the class.

Examples

Here are examples demonstrating the use of static members:

Static Field and Method

```
1. public class Car
2. {
3.     public static int NumberOfCars = 0;
```

```
4.
5.      public Car()
6.      {
7.          // Increment the static field value by one each time a new Car object is created.
8.          NumberOfCars++;
9.      }
10.
11.     public static void DisplayNumberOfCars()
12.     {
13.         Console.WriteLine($"Total cars: {NumberOfCars}");
14.     }
15. }
```

In this example, NumberOfCars is a static field that tracks the number of Car instances created. DisplayNumberOfCars is a static method that displays this number.

Both are accessed via the class name:

```
1. Car car1 = new Car();
2. Car car2 = new Car();
3. Car.DisplayNumberOfCars(); // Output: Total cars: 2
4. Static Class
5.
6. public static class Utilities
7. {
8.     public static int Add(int a, int b)
9.     {
10.        return a + b;
11.    }
```

12. }

Here, Utilities is a static class that contains a static method Add. This method can be called without creating an instance of the class:

```
int result = Utilities.Add(5, 10); // Output: 15
```

Usage Considerations

- Memory Usage: Static members are allocated once and live for the duration of the application, which can be more memory-efficient for shared data.

- Global State: Static members maintain a global state which can lead to issues with data consistency, especially in multithreaded scenarios. Care should be taken to manage access to static members, possibly using synchronisation mechanisms if needed.

- Testing Challenges: Classes that use static members can be harder to test due to their persistent state across tests. This can be mitigated by using patterns such as dependency injection to abstract away static dependencies.

Overall, static is a powerful keyword in C# that enables class-level functionality, making it suitable for scenarios where you need consistent behavior or data management that is common across all instances of a class.

For everything we have used so far and are going to use until quite late into the course, these are things that you can ignore for now.