.NET_WEB_Workspace

1. Keywords

- In C#, a keyword is a reserved word that has a special meaning in the language.
- Keywords are used to define the structure and syntax of the C# programming language and cannot be used as identifiers (names of variables, classes, methods, etc.)
- They are predefined and serve various purposes such as declaring types, controlling program flow, defining methods, and more.
- the categories of keywords are :

a. Modifier Keywords

- Modifier keywords are specific keywords that indicate who can modify types and type members.
- Modifiers allow or prevent certain parts of programs from being modified by other part

1. `abstract`

The `abstract` keyword is used to declare abstract classes and abstract class members. An abstract class cannot be instantiated, and it is meant to be inherited by other classes. Abstract members must be implemented in derived classes.

```
abstract class Animal
{
    public abstract void MakeSound();
}

class Dog : Animal
{
    public override void MakeSound()
    {
        Console.WriteLine("Bark");
    }
}

class Program
{
    static void Main()
    {
        Animal myDog = new Dog();
        myDog.MakeSound(); // Output: Bark
    }
}
```

2. `async`

The `async` keyword is used to define asynchronous methods, which can contain the `await` keyword to make asynchronous calls.

```
using System;
using System.Threading.Tasks;

class Program
{
    static async Task Main(string[] args)
    {
        await PrintMessage();
    }

    static async Task PrintMessage()
    {
        await Task.Delay(1000);
        Console.WriteLine("Hello, async world!");
    }
}
```

3. `const`

The `const` keyword is used to declare a constant field or local. Constants must be initialized as they are declared and cannot be changed thereafter.

```
class Program
{
    const double PI = 3.14159;

    static void Main()
    {
        Console.WriteLine($"The value of PI is {PI}");
    }
}
```

4. `event`

The `event` keyword is used to declare an event in a class. Events are a way for a class to notify other classes or objects when something of interest occurs.

```
using System;

class Publisher
{
    public event EventHandler RaiseEvent;

    public void DoSomething()
```

```
    {
        OnRaiseEvent();
    }

    protected virtual void OnRaiseEvent()
    {
        RaiseEvent?.Invoke(this, EventArgs.Empty);
    }
}

class Subscriber
{
    public void HandleEvent(object sender, EventArgs e)
    {
        Console.WriteLine("Event received.");
    }
}

class Program
{
    static void Main()
    {
        Publisher publisher = new Publisher();
        Subscriber subscriber = new Subscriber();

        publisher.RaiseEvent += subscriber.HandleEvent;
        publisher.DoSomething(); // Output: Event received.
    }
}
```

5. `extern`

The `extern` keyword is used to declare a method implemented externally, typically in a DLL (Dynamic Link Library) written in another language like C or C++. Here are the key points and use cases for extern:

- Calling Native Code
- DLLImport Attribute: When using extern, you typically pair it with the [DllImport] attribute.
- This attribute specifies the name of the DLL containing the external function and optionally its calling convention and other details.

```
using System.Runtime.InteropServices;

class Program
{
    [DllImport("User32.dll")]
    public static extern int MessageBox(IntPtr hWnd, string text, string caption,
int type);

    static void Main()
    {
        MessageBox(IntPtr.Zero, "Hello, World!", "Message", 0);
```

```
    }
}
```

6. new

The new keyword is used to hide an inherited member from a base class. It can also be used to create instances of types.

- while the new keyword itself isn't responsible for directly allocating memory (the runtime manages this),
- it triggers the allocation of memory on the heap for an object of the specified type (DerivedClass in this case) and initializes that memory by invoking the constructor.

```
class BaseClass
{
    public void Display()
    {
        Console.WriteLine("BaseClass Display");
    }
}

class DerivedClass : BaseClass
{
    public new void Display()
    {
        Console.WriteLine("DerivedClass Display");
    }
}

class Program
{
    static void Main()
    {
        DerivedClass obj = new DerivedClass();
        obj.Display(); // Output: DerivedClass Display

        BaseClass baseObj = obj;
        baseObj.Display(); // Output: BaseClass Display
    }
}
```

7. override

The override keyword is used to extend or modify the abstract or virtual implementation of an inherited method, property, indexer, or event.

- method

```
class BaseClass
{

    // property
      public virtual int MyProperty { get; set; }

   // method
    public virtual void Display()
    {
        Console.WriteLine("BaseClass Display");
    }
    // indexer
     public virtual int this[int index]
    {
        get { return 0; }
        set { }
    }

    //event
      public virtual event EventHandler MyEvent;
}

class DerivedClass : BaseClass
{
    public override int MyProperty
    {
        get { return base.MyProperty; }
        set { base.MyProperty = value; }
    }

    public override void Display()
    {
        Console.WriteLine("DerivedClass Display");
    }


    public override int this[int index]
    {
        get { return base[index]; }
        set { base[index] = value; }
    }

    //
    public override event EventHandler MyEvent
    {
        add { base.MyEvent += value; }
        remove { base.MyEvent -= value; }
    }
}

class Program
{
    static void Main()
```

```
    {
        DerivedClass obj = new DerivedClass();
        obj.Display(); // Output: DerivedClass Display
    }
}
```

8. `partial`

The `partial` keyword allows the definition of a class, struct, or interface to be split into multiple files. All parts are combined when the application is compiled.

Use cases:

- Code Organization
- Code Generation: Extending generated code without modifying original files.
- Team Collaboration
- Framework Extensibility: Allowing developers to add custom logic to framework-generated classes or structs.

```
partial class MyClass
{
    public void Method1()
    {
        Console.WriteLine("Method1");
    }
}

partial class MyClass
{
    public void Method2()
    {
        Console.WriteLine("Method2");
    }
}

class Program
{
    static void Main()
    {
        MyClass obj = new MyClass();
        obj.Method1(); // Output: Method1
        obj.Method2(); // Output: Method2
    }
}
```

9. `readonly`

The `readonly` keyword is used to declare a field that can only be assigned during declaration or in a constructor of the same class. It prevents the field from being modified after construction.

```
class Program
{
    readonly int number;

    public Program(int num)
    {
        number = num;
    }

    static void Main()
    {
        Program p = new Program(10);
        Console.WriteLine(p.number); // Output: 10
    }
}
```

10. `sealed`

The `sealed` keyword is used to prevent a class from being inherited or to prevent a method from being overridden.

```
sealed class SealedClass
{
    public void Display()
    {
        Console.WriteLine("SealedClass Display");
    }
}

// The following class declaration would cause a compile-time error
// class DerivedClass : SealedClass { }

class Program
{
    static void Main()
    {
        SealedClass obj = new SealedClass();
        obj.Display(); // Output: SealedClass Display
    }
}
```

11. `static`

The `static` keyword is used to declare a static member, which belongs to the type itself rather than to any specific object. Static members are accessed using the type name.

Use Cases

- Static Fields and Properties
- Static Methods:

1. Utility Methods: Define methods that perform common tasks and do not require instance-specific data.
2. Factory Methods: Provide methods to create instances of a class without directly using its constructors.

- Static Constructors
- Static Classes: Declare classes that can't be instantiated and can only contain static members
- Benefits of static Memory

    1. Efficiency: Static members are stored in a shared memory location, which can save memory compared to instance-specific data.
    2. Performance: Accessing static members can be faster because they do not require an instance reference.
    3. Global Accessibility

```
class Program
{
    static int counter = 0;

    static void Increment()
    {
        counter++;
    }

    static void Main()
    {
        Increment();
        Console.WriteLine($"Counter: {counter}"); // Output: Counter: 1
    }
}
// factory method
public class Logger
{
    private Logger() { } // Private constructor

    public static Logger CreateLogger()
    {
        return new Logger();
    }
}
```

12. `unsafe`

The `unsafe` keyword allows code that uses pointers to be written in C#. Unsafe code is not verified by the CLR for safety.

Use Case

1. Interop with Native Code
2. Performance Optimization when implementing high-performance algorithms or data structures.

```
public static class NativeMethods
{
    [DllImport("mylibrary.dll")]
    public static unsafe extern void ProcessData(int* data, int length);
}

class Program
{
    unsafe static void Main()
    {
        int number = 10;
        int* ptr = &number;
        Console.WriteLine($"Value: {*ptr}, Address: {(long)ptr}");
    }
}
```

13. `virtual`

The `virtual` keyword is used to modify a method, property, indexer, or event declaration to allow it to be overridden in a derived class.

- This enables polymorphism, where different classes can provide their own implementation of the method.
- the virtual keyword in C# facilitates object-oriented principles such as inheritance, polymorphism, and extensibility by allowing methods, properties, indexers, and events to be overridden in derived classes.
- It's essential for building flexible and reusable class hierarchies that accommodate varying implementation details across different subclasses.

```
public class Shape
{
    public virtual int Area
    {
        get { return 0; }
    }

    public virtual event EventHandler Calculate;
}

public class Rectangle : Shape
{
    private int length;
    private int width;
    public override int Area
```

```
    {
        get { return length * width; }
    }
     public override event EventHandler Calculate;
}


class Program
{
    static void Main()
    {
        BaseClass obj = new DerivedClass();
        obj.Display(); // Output: DerivedClass Display
    }
}
```

14. volatile

The volatile keyword indicates that a field might be modified by multiple threads that are executing at the same time.

- Fields declared as volatile are not subject to compiler optimizations that assume access by a single thread.
- The volatile keyword in C# is crucial for ensuring the visibility and consistency of shared fields across multiple threads.
- It prevents compiler and CPU optimizations that could compromise thread safety, making it essential in concurrent programming scenarios.

```
public class SharedData
{
    private volatile bool _flag;

    public void ToggleFlag()
    {
        _flag = !_flag; // Thread-safe toggle operation
    }

    public bool GetFlag()
    {
        return _flag; // Thread-safe read operation
    }
}
```

- In this example, _flag is declared as volatile to ensure that changes made to _flag by one thread are immediately visible to other threads. This prevents issues such as reading stale or inconsistent values of _flag due to optimizations or CPU caching.

```
public class Singleton
{
    private static volatile Singleton _instance;
    private static readonly object _lock = new object();

    public static Singleton Instance
    {
        get
        {
            if (_instance == null)
            {
                lock (_lock)
                {
                    if (_instance == null)
                    {
                        _instance = new Singleton();
                    }
                }
            }
            return _instance;
        }
    }

    private Singleton()
    {
        // Constructor logic
    }
}
```

- While volatile ensures visibility and synchronization, it does not provide atomicity for compound operations. For atomic operations, consider using locks (Monitor), Mutex, or Interlocked class methods.
- (Atomicity is a fundamental concept in concurrent programming that ensures operations on shared data are performed atomically, preserving data integrity and thread safety.)

b. Access modifier

Certainly! Access modifiers in C# define the accessibility or visibility of classes, structs, methods, properties, and other members within the same assembly or across assemblies. They control how other parts of the code can interact with and use these members. Here's a detailed breakdown of access modifiers, along with examples and use cases:

Access Modifiers in C#

Access modifiers in C# provide essential control over the visibility and accessibility of types and members within codebases.

- They enable encapsulation, promote reusability, and enforce design principles such as information hiding and separation of concerns.

1. Public

- **Definition**: Public members are accessible from any other code in the same assembly or a different assembly.
- **Example**:

```
public class MyClass
{
    public void PublicMethod()
    {
        // Method implementation
    }

    public int PublicProperty { get; set; }
}
```

- **Use Cases**:
    - Exposing APIs or components intended for use by other parts of the application or external assemblies.
    - Creating reusable libraries where public members define the library's interface.

2. Private

- **Definition**: Private members are accessible only within the containing type (class or struct).
- **Example**:

```
public class MyClass
{
    private int _privateField;

    private void PrivateMethod()
    {
        // Method implementation
    }

    private int PrivateProperty { get; set; }
}
```

- **Use Cases**:
    - Encapsulating implementation details and restricting access to internal workings of a class.
    - Holding state or methods that are implementation-specific and not meant to be exposed publicly.

3. Protected

- **Definition**: Protected members are accessible within the containing type and any derived types (subclasses).
- **Example**:

```
public class MyBaseClass
{
    protected int _protectedField;

    protected void ProtectedMethod()
    {
        // Method implementation
    }

    protected int ProtectedProperty { get; set; }
}

public class MyDerivedClass : MyBaseClass
{
    public void AccessProtectedMember()
    {
        _protectedField = 10; // Accessing protected field from base class
        ProtectedMethod();   // Calling protected method from base class
    }
}
```

- **Use Cases**:
  - Enabling derived classes to access and override certain behaviors or states defined in the base class.
  - Facilitating polymorphism and inheritance while controlling visibility of implementation details.

4. Internal

- **Definition**: Internal members are accessible within the same assembly (or module), but not from outside the assembly.
- **Example**:

```
internal class InternalClass
{
    internal void InternalMethod()
    {
        // Method implementation
    }

    internal int InternalProperty { get; set; }
}
```

- **Use Cases**:
  - Sharing implementation details among classes and components within the same assembly without exposing them to external assemblies.
  - Organizing code into cohesive units while enforcing access boundaries within a single project or library.

5. Protected Internal (protected internal)

- **Definition**: Protected internal members are accessible within the same assembly, as well as by derived types (subclasses) in other assemblies.
- **Example**:

```
public class MyBaseClass
{
    protected internal int _protectedInternalField;

    protected internal void ProtectedInternalMethod()
    {
        // Method implementation
    }

    protected internal int ProtectedInternalProperty { get; set; }
}

public class MyDerivedClass : MyBaseClass
{
    public void AccessProtectedInternalMember()
    {
        _protectedInternalField = 10;   // Accessing protected internal
field from base class
        ProtectedInternalMethod();     // Calling protected internal method
from base class
    }
}
```

- **Use Cases**:
  - Allowing derived classes from other assemblies to access and override members defined in the base class.
  - Sharing common functionality or state across related components or modules within the same assembly.