.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

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**:

```csharp
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**:

```csharp
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**:

```csharp
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**:

```csharp
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.

```
public class ExampleClass
{
    public int PublicField; // Accessible from any code
    private int PrivateField; // Accessible only within this class
    protected int ProtectedField; // Accessible within this class and derived
classes
    internal int InternalField; // Accessible within this assembly
    protected internal int ProtectedInternalField; // Accessible within this
assembly and derived classes
```

```
    private protected int PrivateProtectedField; // Accessible within this class
and derived classes in the same assembly

    public void ExampleMethod()
    {
        // All fields are accessible within the class
    }
}
```

c.statement keywords in C#:

Statement keywords are related to program flow.

1. **if**: Used to specify a block of code to be executed if a specified condition is true.

```
if (condition) {
    // code to be executed if the condition is true
}
```

2. **else**: Used to specify a block of code to be executed if the same condition is false.

```
if (condition) {
    // code to be executed if the condition is true
} else {
    // code to be executed if the condition is false
}
```

3. **switch**: Allows a variable to be tested for equality against a list of values.

```
switch (expression) {
    case value1:
        // code to be executed if expression equals value1
        break;
    case value2:
        // code to be executed if expression equals value2
        break;
    default:
        // code to be executed if expression doesn't match any case
        break;
}
```

4. **case**: Defines a block of code to be executed in the switch statement for a particular match.

```
case value:
    // code to be executed if the case matches
    break;
```

5. **do**: Used in a loop that executes a block of code once before checking if a condition is true, then repeats the loop as long as the condition is true.

```
do {
    // code to be executed
} while (condition);
```

6. **for**: Used to create a loop that consists of three parts: initialization, condition, and iteration.

```
for (initialization; condition; iteration) {
    // code to be executed
}
```

7. **foreach**: Used to iterate over a collection or array.

```
foreach (type variable in collection) {
    // code to be executed for each item in the collection
}
```

8. **in**: Used in the `foreach` loop to iterate through items in a collection.

```
foreach (type variable in collection) {
    // code to be executed for each item in the collection
}
```

9. **while**: Creates a loop that executes a block of code as long as a specified condition is true.

```
while (condition) {
    // code to be executed
}
```

10. **break**: Terminates the closest enclosing loop or switch statement.

```
break;
```

11. **continue**: Skips the current iteration of the closest enclosing loop and proceeds with the next iteration.

```
continue;
```

12. **default**: Specifies the default block of code in a switch statement if no case matches.

```
default:
    // code to be executed if no case matches
    break;
```

13. **goto**: Transfers control to a labeled statement within the same function.

```
goto label;
...
label:
    // code to be executed
```

14. **return**: Exits a function and optionally returns a value.

```
return value;
```

15. **yield**: Used in an iterator block to return each element one at a time.

```
yield return value;
```

16. **throw**: Used to signal the occurrence of an exception.

```
throw new Exception("Error message");
```

17. **try**: Defines a block of code in which exceptions will be checked.

```
try {
    // code that may cause an exception
} catch (ExceptionType e) {
    // code to handle the exception
} finally {
    // code to be executed regardless of an exception
}
```

18. **catch**: Defines a block of code to handle exceptions thrown by the `try` block.

    ```
    try {
        // code that may cause an exception
    } catch (ExceptionType e) {
        // code to handle the exception
    }
    ```

19. **finally**: Defines a block of code that will be executed after a `try` block, regardless of whether an exception was thrown or not.

    ```
    try {
        // code that may cause an exception
    } catch (ExceptionType e) {
        // code to handle the exception
    } finally {
        // code to be executed regardless of an exception
    }
    ```

20. **checked**: Used to explicitly enable overflow checking for integral-type arithmetic operations and conversions.

    ```
    checked {
        // code that requires overflow checking
    }
    ```

21. **unchecked**: Used to explicitly disable overflow checking for integral-type arithmetic operations and conversions.

    ```
    unchecked {
        // code that does not require overflow checking
    }
    ```

22. **fixed**: Prevents the garbage collector from relocating a movable variable, allowing pointers to be used on managed objects.

    ```
    fixed (type* ptr = &variable) {
        // code that uses the fixed pointer
    }
    ```

23. **lock**: Ensures that one thread does not enter a critical section of code while another thread is in the critical section.

```
lock (object) {
    // code that is thread-safe
}
```

d. Method Parameters

params, ref, and out

`params`

Definition

The `params` keyword allows you to pass a variable number of arguments to a method. This is useful when you don't know beforehand how many arguments will be passed.

Code Snippet

```csharp
using System;

public class Example
{
    // Using the params keyword to accept a variable number of integers
    public void PrintNumbers(params int[] numbers)
    {
        foreach (int number in numbers)
        {
            Console.WriteLine(number);
        }
    }
}

public class Program
{
    public static void Main()
    {
        Example example = new Example();
        // Calling the method with a varying number of arguments
        example.PrintNumbers(1, 2, 3, 4, 5);
        example.PrintNumbers(10, 20, 30);
    }
}
```

Use Case

Imagine you're writing a logging function where you might need to log different numbers of messages. Using `params`, you can easily handle this scenario without overloading methods multiple times.

`ref`

Definition

The `ref` keyword allows a method to modify the value of an argument passed to it. This means any changes made to the parameter inside the method will reflect outside the method as well.

Code Snippet

```csharp
using System;

public class Example
{
    // Using the ref keyword to modify the argument's value
    public void AddTen(ref int number)
    {
        number += 10;
    }
}

public class Program
{
    public static void Main()
    {
        int myNumber = 5;
        Example example = new Example();
        // Calling the method with the ref keyword
        example.AddTen(ref myNumber);
        Console.WriteLine(myNumber); // Output: 15
    }
}
```

Use Case

As a developer, you might use `ref` in scenarios where you need to modify the state of an input variable, like updating a counter or modifying an object within a method without returning it.

`out`

Definition

The `out` keyword is similar to `ref`, but it requires that the called method must assign a value to the out parameter before the method returns. The initial value of the argument passed to `out` doesn't need to be set beforehand.

Code Snippet

```
using System;

public class Example
{
    // Using the out keyword to return multiple values from a method
    public void GetValues(out int number1, out int number2)
    {
        number1 = 10;
        number2 = 20;
    }
}

public class Program
{
    public static void Main()
    {
        int myNumber1, myNumber2;
        Example example = new Example();
        // Calling the method with the out keyword
        example.GetValues(out myNumber1, out myNumber2);
        Console.WriteLine(myNumber1); // Output: 10
        Console.WriteLine(myNumber2); // Output: 20
    }
}
```

Use Case

Using `out` is particularly useful when you need a method to return multiple values. For example, you might use it in a method that parses a string and returns both the parsed value and a status code indicating success or failure.

Interview Perspective

When I interview candidates, I often ask about these keywords because they reveal a lot about a developer's understanding of method parameters and how to manage data within methods.

1. **`params`**: I want to see if the candidate knows how to handle variable-length argument lists, which is essential for creating flexible APIs.
2. **`ref`**: This keyword is about understanding references and value types. It shows if the candidate knows how to work with data that needs to be modified directly.
3. **`out`**: I look for an understanding of scenarios where multiple return values are needed. This often indicates the candidate's ability to design methods that provide clear and multiple outputs.

d. Namespace keywords

`using`

Definition

The `using` keyword in C# has two main uses:

1. To include a namespace in your file, which allows you to use the types defined in that namespace without needing to specify the full namespace path.
2. To define a `using` statement for resource management, which ensures that `IDisposable` objects are disposed of properly.

Code Snippet

1. **Including a Namespace:**

   ```csharp
   using System;
   using System.Collections.Generic;

   public class Example
   {
       public void PrintMessage()
       {
           Console.WriteLine("Hello, World!");
       }
   }
   ```

2. **Using Statement for Resource Management:**

   ```csharp
   using System;
   using System.IO;

   public class Example
   {
       public void WriteToFile(string filePath, string content)
       {
           using (StreamWriter writer = new StreamWriter(filePath))
           {
               writer.WriteLine(content);
           } // StreamWriter is disposed of here
       }
   }
   ```

Use Case

- **Including a Namespace**: Simplifies code by removing the need to fully qualify types.
- **Resource Management**: Ensures that resources like file handles, database connections, etc., are properly released.

`. Operator`

Definition

The `.` operator is used to access members (methods, properties, fields) of a type or namespace.

Code Snippet

```
using System;

public class Example
{
    public void PrintCurrentDate()
    {
        Console.WriteLine(DateTime.Now); // Using . operator to access the Now
property of DateTime
    }
}

public class Program
{
    public static void Main()
    {
        Example example = new Example();
        example.PrintCurrentDate();
    }
}
```

Use Case

The `.` operator is fundamental in accessing the members of types and namespaces, allowing you to use their functionality.

`::` Operator

Definition

The `::` operator, known as the namespace alias qualifier, is used to access members of a namespace or type that is explicitly specified by an alias.

Code Snippet

```
extern alias MyAlias;

public class Example
{
    public void UseAlias()
    {
        MyAlias::SomeNamespace.SomeClass instance = new
MyAlias::SomeNamespace.SomeClass();
        instance.SomeMethod();
    }
}
```

Use Case

The `::` operator is used when dealing with namespace aliases, particularly useful in scenarios where there might be conflicts between types in different namespaces.

`extern alias`

Definition

The `extern alias` keyword is used to reference external assemblies with the same fully qualified type names, allowing the use of different versions of the same assembly.

Code Snippet

1. **Assembly Reference (assumed in .csproj or similar):**

```xml
<ItemGroup>
    <Reference Include="LibraryA">
        <Aliases>AliasA</Aliases>
    </Reference>
    <Reference Include="LibraryB">
        <Aliases>AliasB</Aliases>
    </Reference>
</ItemGroup>
```

2. **C# Code:**

```csharp
extern alias AliasA;
extern alias AliasB;

public class Example
{
    public void UseAliases()
    {
        AliasA::Library.SomeClass instanceA = new
AliasA::Library.SomeClass();
        AliasB::Library.SomeClass instanceB = new
AliasB::Library.SomeClass();

        instanceA.SomeMethod();
        instanceB.SomeMethod();
    }
}
```

Use Case

The `extern alias` keyword is particularly useful when you need to reference different versions of the same assembly or avoid type name conflicts in large projects.

By understanding and utilizing these namespace-related keywords, you can write cleaner, more maintainable, and conflict-free C# code, especially in large projects with multiple dependencies.

Operator Keywords

they perform miscellaneous actions.

1. `as` Keyword

The `as` keyword is used for type conversion. It performs a safe cast and returns `null` if the conversion fails instead of throwing an exception.

Definition

```
<target-type> variableName = expression as <target-type>;
```

Code Snippet

```
object obj = "Hello, world!";
string str = obj as string; // Successful cast
Console.WriteLine(str); // Output: Hello, world!

object number = 123;
string notString = number as string; // Unsuccessful cast, returns null
Console.WriteLine(notString == null); // Output: True
```

Use Cases

- Safe type conversion without exceptions.
- Useful in scenarios where the type may not be guaranteed and you want to avoid exceptions.

2. `await` Keyword

The `await` keyword is used in asynchronous programming to wait for the completion of an async operation.

Definition

```
await expression;
```

Code Snippet

```
public async Task<string> GetDataAsync()
{
```

```
        await Task.Delay(1000); // Simulate asynchronous operation
        return "Data retrieved";
    }

    public async Task ExampleMethod()
    {
        string data = await GetDataAsync();
        Console.WriteLine(data); // Output: Data retrieved
    }
```

Use Cases

- Waiting for asynchronous operations to complete.
- Used in conjunction with `async` methods to perform non-blocking operations.

3. `is` Keyword

The `is` keyword checks if an object is of a specified type.

Definition

```
if (expression is <target-type> variableName)
{
    // Code block
}
```

Code Snippet

```
object obj = "Hello, world!";
if (obj is string str)
{
    Console.WriteLine(str); // Output: Hello, world!
}

object number = 123;
if (number is int intValue)
{
    Console.WriteLine(intValue); // Output: 123
}
```

Use Cases

- Type checking before casting.
- Pattern matching and conditional logic based on type.

4. `new` Keyword

The `new` keyword is used to create instances of types and to hide inherited members.

Definition

```
TypeName variableName = new TypeName();
```

Code Snippet

```
// Creating an instance
MyClass instance = new MyClass();

// Hiding inherited members
class BaseClass
{
    public void Display() => Console.WriteLine("Base class");
}

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

BaseClass obj = new DerivedClass();
obj.Display(); // Output: Base class
```

Use Cases

- Creating objects.
- Hiding inherited members with the same name.

5. `sizeof` Keyword

The `sizeof` keyword returns the size of a value type in bytes.

Definition

```
int size = sizeof(<value-type>);
```

Code Snippet

```
int sizeOfInt = sizeof(int); // Output: 4
Console.WriteLine(sizeOfInt);

unsafe
```

```
{
    int sizeOfStruct = sizeof(MyStruct);
    Console.WriteLine(sizeOfStruct);
}

struct MyStruct
{
    public int Field1;
    public double Field2;
}
```

Use Cases

- Determining the size of value types in unsafe contexts.
- Useful in low-level programming and interop scenarios.

6. `typeof` Keyword

The `typeof` keyword obtains the `System.Type` object for a type.

Definition

```
Type type = typeof(<type>);
```

Code Snippet

```
Type intType = typeof(int);
Console.WriteLine(intType.FullName); // Output: System.Int32

Type stringType = typeof(string);
Console.WriteLine(stringType.FullName); // Output: System.String
```

Use Cases

- Reflection to get type information at runtime.
- Metadata inspection and dynamic type handling.

7. `stackalloc` Keyword

The `stackalloc` keyword allocates a block of memory on the stack.

Definition

```
int* pointer = stackalloc int[10];
```

Code Snippet

```
unsafe
{
    int* array = stackalloc int[10];
    for (int i = 0; i < 10; i++)
    {
        array[i] = i;
        Console.WriteLine(array[i]); // Output: 0 to 9
    }
}
```

Use Cases

- High-performance scenarios where stack allocation is preferred.
- Low-level memory management in unsafe contexts.

8. `checked` Keyword

The `checked` keyword enables overflow checking for arithmetic operations and conversions.

Definition

```
checked
{
    // Code block
}

int result = checked(expression);
```

Code Snippet

```
try
{
    int maxInt = int.MaxValue;
    int result = checked(maxInt + 1); // Throws OverflowException
}
catch (OverflowException ex)
{
    Console.WriteLine("Overflow exception: " + ex.Message);
}
```

Use Cases

- Ensuring that arithmetic operations do not silently overflow.

- Debugging and validating numerical computations.

9. `unchecked` Keyword

The `unchecked` keyword disables overflow checking for arithmetic operations and conversions.

Definition

```
unchecked
{
    // Code block
}

int result = unchecked(expression);
```

Code Snippet

```
int maxInt = int.MaxValue;
int result = unchecked(maxInt + 1); // Overflow is ignored
Console.WriteLine(result); // Output: -2147483648
```

Use Cases

- Situations where overflow is expected and should be ignored.
- Performance optimization by avoiding overflow checks.

Access Keyword

these keywords are used to access the containing class or base class of an object or instance of a class. Certainly! Here are detailed explanations of the `base` and `this` access keywords in C#, including their definitions, code snippets demonstrating their use, and real-life project use cases.

1. `base` Keyword

Definition

The `base` keyword is used to access members of the base class from within a derived class. It can be used to call base class constructors, methods, or access base class properties and fields.

Code Snippets

1. **Calling Base Class Constructor**

```
public class Animal
{
    public string Name { get; private set; }
```

```
    public Animal(string name)
    {
        Name = name;
    }
}

public class Dog : Animal
{
    public string Breed { get; private set; }

    public Dog(string name, string breed) : base(name)
    {
        Breed = breed;
    }
}
```

2. **Calling Base Class Method**

```
public class Animal
{
    public virtual void Speak()
    {
        Console.WriteLine("Animal sound");
    }
}

public class Dog : Animal
{
    public override void Speak()
    {
        base.Speak(); // Calls the Speak method in the base class
        Console.WriteLine("Bark");
    }
}
```

Real-Life Project Use Case

Scenario: Creating a base class for different types of vehicles with common functionality and deriving specific vehicle classes.

```
public class Vehicle
{
    public string Make { get; private set; }
    public string Model { get; private set; }

    public Vehicle(string make, string model)
    {
        Make = make;
```

```
            Model = model;
        }

        public virtual void DisplayInfo()
        {
            Console.WriteLine($"Vehicle: {Make} {Model}");
        }
    }

    public class Car : Vehicle
    {
        public int NumberOfDoors { get; private set; }

        public Car(string make, string model, int numberOfDoors)
            : base(make, model)
        {
            NumberOfDoors = numberOfDoors;
        }

        public override void DisplayInfo()
        {
            base.DisplayInfo();
            Console.WriteLine($"Number of Doors: {NumberOfDoors}");
        }
    }

    // Usage
    class Program
    {
        static void Main()
        {
            Car myCar = new Car("Toyota", "Corolla", 4);
            myCar.DisplayInfo();
        }
    }
```

2. `this` Keyword

Definition

The `this` keyword refers to the current instance of the class. It is used to access members of the current instance, distinguish between instance members and parameters with the same name, or pass the current instance as a parameter to another method.

Code Snippets

1. **Accessing Instance Members**

```
    public class Person
    {
        private string name;
```

```csharp
    public Person(string name)
    {
        this.name = name; // Refers to the instance variable
    }

    public void Display()
    {
        Console.WriteLine(this.name);
    }
}
```

2. **Calling Another Constructor**

```csharp
public class Person
{
    private string name;
    private int age;

    public Person(string name)
    {
        this.name = name;
    }

    public Person(string name, int age) : this(name)
    {
        this.age = age;
    }

    public void Display()
    {
        Console.WriteLine($"{this.name}, Age: {this.age}");
    }
}
```

Real-Life Project Use Case

Scenario: Managing and displaying user information in a web application.

```csharp
public class User
{
    public string Username { get; private set; }
    public string Email { get; private set; }

    public User(string username, string email)
    {
        this.Username = username;
        this.Email = email;
    }
```

```
    public void DisplayInfo()
    {
        Console.WriteLine($"Username: {this.Username}, Email: {this.Email}");
    }

    public void UpdateEmail(string newEmail)
    {
        if (IsValidEmail(newEmail))
        {
            this.Email = newEmail;
        }
    }

    private bool IsValidEmail(string email)
    {
        // Simple validation for the sake of example
        return email.Contains("@");
    }
}

// Usage
class Program
{
    static void Main()
    {
        User user = new User("john_doe", "john@example.com");
        user.DisplayInfo();

        user.UpdateEmail("john.doe@example.com");
        user.DisplayInfo();
    }
}
```

Summary

- `base` **Keyword**:
  - Used to access members of the base class from a derived class.
  - Commonly used to call base class constructors and methods.
  - Helps in extending base class functionality in derived classes.
- `this` **Keyword**:
  - Refers to the current instance of the class.
  - Helps distinguish between instance members and parameters with the same name.
  - Can be used to call other constructors or pass the current instance to methods.

Literal Keywords

these keywords apply to current instance or value of an object

Sure! Let's go through the literal keywords in C# (`null`, `false`, `true`, `value`, and `void`) with their definitions, code snippets, and real-life project use cases.

1. `null` Keyword

Definition

The `null` keyword represents a null reference, indicating that a variable does not reference any object or value.

Code Snippet

```csharp
string str = null;

if (str == null)
{
    Console.WriteLine("The string is null.");
}
```

Real-Life Project Use Case

Scenario: Checking for null values before performing operations to avoid null reference exceptions.

```csharp
public class UserRepository
{
    private List<User> users;

    public UserRepository()
    {
        users = new List<User>();
    }

    public User GetUserById(int id)
    {
        return users.FirstOrDefault(u => u.Id == id);
    }
}

// Usage
class Program
{
    static void Main()
    {
        UserRepository userRepository = new UserRepository();
        User user = userRepository.GetUserById(1);

        if (user == null)
        {
            Console.WriteLine("User not found.");
```

```
        }
        else
        {
            Console.WriteLine($"User found: {user.Name}");
        }
    }
}
```

2. `false` Keyword

Definition

The `false` keyword represents the boolean value false.

Code Snippet

```csharp
bool isAvailable = false;

if (!isAvailable)
{
    Console.WriteLine("Item is not available.");
}
```

Real-Life Project Use Case

Scenario: Using boolean flags to manage state in an application.

```csharp
public class FeatureToggle
{
    public bool IsFeatureEnabled { get; private set; }

    public FeatureToggle()
    {
        IsFeatureEnabled = false;
    }

    public void EnableFeature()
    {
        IsFeatureEnabled = true;
    }
}

// Usage
class Program
{
    static void Main()
    {
        FeatureToggle featureToggle = new FeatureToggle();
```

```
        if (!featureToggle.IsFeatureEnabled)
        {
            Console.WriteLine("Feature is disabled.");
        }
        else
        {
            Console.WriteLine("Feature is enabled.");
        }

        featureToggle.EnableFeature();

        if (featureToggle.IsFeatureEnabled)
        {
            Console.WriteLine("Feature is now enabled.");
        }
    }
}
```

3. `true` Keyword

Definition

The `true` keyword represents the boolean value true.

Code Snippet

```
bool isAvailable = true;

if (isAvailable)
{
    Console.WriteLine("Item is available.");
}
```

Real-Life Project Use Case

Scenario: Using boolean flags to control access to application features.

```
public class FeatureToggle
{
    public bool IsFeatureEnabled { get; private set; }

    public FeatureToggle()
    {
        IsFeatureEnabled = true;
    }

    public void DisableFeature()
```

```
    {
        IsFeatureEnabled = false;
    }
}

// Usage
class Program
{
    static void Main()
    {
        FeatureToggle featureToggle = new FeatureToggle();

        if (featureToggle.IsFeatureEnabled)
        {
            Console.WriteLine("Feature is enabled.");
        }
        else
        {
            Console.WriteLine("Feature is disabled.");
        }

        featureToggle.DisableFeature();

        if (!featureToggle.IsFeatureEnabled)
        {
            Console.WriteLine("Feature is now disabled.");
        }
    }
}
```

4. `value` Keyword

Definition

The `value` keyword is used in property setters to refer to the value being assigned to the property.

Code Snippet

```
public class Person
{
    private string name;

    public string Name
    {
        get { return name; }
        set { name = value; }
    }
}

// Usage
class Program
```

```
{
    static void Main()
    {
        Person person = new Person();
        person.Name = "John Doe";
        Console.WriteLine(person.Name);
    }
}
```

Real-Life Project Use Case

Scenario: Implementing encapsulation in a class using properties.

```
public class Product
{
    private decimal price;

    public decimal Price
    {
        get { return price; }
        set
        {
            if (value < 0)
            {
                throw new ArgumentException("Price cannot be negative.");
            }
            price = value;
        }
    }
}

// Usage
class Program
{
    static void Main()
    {
        Product product = new Product();

        try
        {
            product.Price = -10;
        }
        catch (ArgumentException ex)
        {
            Console.WriteLine(ex.Message);
        }

        product.Price = 100;
        Console.WriteLine($"Product Price: {product.Price}");
    }
}
```

5. `void` Keyword

Definition

The `void` keyword specifies that a method does not return a value.

Code Snippet

```
public void PrintMessage(string message)
{
    Console.WriteLine(message);
}

// Usage
class Program
{
    static void Main()
    {
        PrintMessage("Hello, World!");
    }
}
```

Real-Life Project Use Case

Scenario: Defining methods that perform actions but do not return values.

```
public class Logger
{
    public void LogInfo(string message)
    {
        Console.WriteLine($"INFO: {message}");
    }

    public void LogError(string message)
    {
        Console.WriteLine($"ERROR: {message}");
    }
}

// Usage
class Program
{
    static void Main()
    {
        Logger logger = new Logger();
        logger.LogInfo("Application started.");
        logger.LogError("An error occurred.");
```

```
        }
    }
```

Summary

- `null` **Keyword**:
  - Represents a null reference.
  - Used to indicate that a variable does not reference any object.
- `false` **Keyword**:
  - Represents the boolean value false.
  - Used to manage state and control flow in boolean logic.
- `true` **Keyword**:
  - Represents the boolean value true.
  - Used to manage state and control flow in boolean logic.
- `value` **Keyword**:
  - Used in property setters to refer to the value being assigned.
  - Facilitates encapsulation by allowing validation and logic in property setters.
- `void` **Keyword**:
  - Specifies that a method does not return a value.
  - Used for methods that perform actions without returning results.

These keywords are fundamental in C# programming, providing essential functionality for handling references, boolean logic, properties, and method definitions.

Type Keywords

these keywords are used for datatypes Certainly! Here are the definitions with value ranges and real-world use cases for each type keyword:

Type Keywords

`bool`

- **Definition**: Represents a Boolean value (`true` or `false`).
- **Value Range**: Only two possible values: `true` or `false`.
- **Real-World Use Case**: Used in logic decisions, state flags, or conditions where binary states are needed, such as enabling/disabling features or checking conditions.

`byte`

- **Definition**: Represents an 8-bit unsigned integer.
- **Value Range**: Values range from `0` to `255`.

- **Real-World Use Case**: Used for efficiency in storage and operations involving small numbers, like image processing where pixel values or file byte data are managed.

`char`

- **Definition**: Represents a single 16-bit Unicode character.
- **Value Range**: Unicode characters ranging from `U+0000` to `U+FFFF`.
- **Real-World Use Case**: Used for handling text and character-based data, such as text parsing, internationalization, and text manipulation tasks in applications.

`class`

- **Definition**: Defines a reference type that can contain data members (fields) and function members (methods, properties, etc.).
- **Value Range**: N/A (Reference type, not a value type).
- **Real-World Use Case**: Used for modeling complex entities in object-oriented programming, such as representing entities like employees, products, or services with associated behaviors and properties.

`decimal`

- **Definition**: Represents a 128-bit precise decimal value suitable for financial and monetary calculations.
- **Value Range**: ±1.0 x 10^-28 to ±7.9 x 10^28.
- **Real-World Use Case**: Ideal for financial applications where precise decimal calculations are crucial, such as currency conversions, interest rate calculations, or any monetary computations.

`double`

- **Definition**: Represents a double-precision floating-point value (64-bit).
- **Value Range**: ±5.0 x 10^-324 to ±1.7 x 10^308.
- **Real-World Use Case**: Used in scientific calculations, engineering simulations, and scenarios requiring high precision with a wide range of values, such as physical simulations or astronomical calculations.

`enum`

- **Definition**: Defines a set of named constants, called enumerators.
- **Value Range**: Limited to the defined set of named constants.
- **Real-World Use Case**: Used to represent a fixed set of values that an object can have, such as days of the week, states of an application, or categories in a classification system.

`float`

- **Definition**: Represents a single-precision floating-point value (32-bit).
- **Value Range**: ±1.5 x 10^-45 to ±3.4 x 10^38.
- **Real-World Use Case**: Suitable for scenarios requiring floating-point calculations with moderate precision, such as graphics rendering, simulations, or real-time data processing where memory efficiency is crucial.

`int`

- **Definition**: Represents a 32-bit signed integer.
- **Value Range**: Values range from `-2,147,483,648` to `2,147,483,647`.
- **Real-World Use Case**: Commonly used for storing and manipulating whole numbers within this range, such as counting, indexing, or representing quantities in applications.

`long`

- **Definition**: Represents a 64-bit signed integer.
- **Value Range**: Values range from `-9,223,372,036,854,775,808` to `9,223,372,036,854,775,807`.
- **Real-World Use Case**: Used when dealing with large integers, such as representing timestamps, file sizes, or database identifiers that require a wider range than `int`.

`sbyte`

- **Definition**: Represents an 8-bit signed integer.
- **Value Range**: Values range from `-128` to `127`.
- **Real-World Use Case**: Used in scenarios where small integers are sufficient, such as low-level system programming, embedded systems, or hardware interfacing.

`short`

- **Definition**: Represents a 16-bit signed integer.
- **Value Range**: Values range from `-32,768` to `32,767`.
- **Real-World Use Case**: Suitable for scenarios where memory optimization or performance is critical but a wider range than `byte` is needed, such as in data compression algorithms or sensor data processing.

`string`

- **Definition**: Represents a sequence of characters.
- **Value Range**: Variable length depending on the number of characters.
- **Real-World Use Case**: Widely used for handling textual data in applications, including user input, data storage, formatting, and display of textual information.

`struct`

- **Definition**: Defines a value type that can contain data members and function members.
- **Value Range**: N/A (Value type, not a reference type).
- **Real-World Use Case**: Used for lightweight objects that do not require inheritance, such as representing coordinates, simple data structures, or immutable data types.

`uint`

- **Definition**: Represents a 32-bit unsigned integer.
- **Value Range**: Values range from `0` to `4,294,967,295`.
- **Real-World Use Case**: Used in scenarios where only non-negative integers are needed, such as bit manipulation, low-level programming, or when dealing with large sets of flags or options.

`ulong`

- **Definition**: Represents a 64-bit unsigned integer.
- **Value Range**: Values range from `0` to `18,446,744,073,709,551,615`.
- **Real-World Use Case**: Used when large non-negative integers are required, such as in cryptography, large-scale data indexing, or when dealing with extremely large quantities.

`ushort`

- **Definition**: Represents a 16-bit unsigned integer.
- **Value Range**: Values range from `0` to `65,535`.
- **Real-World Use Case**: Suitable for scenarios where small positive integers are sufficient, such as image processing, network protocols, or memory-constrained applications.

These definitions and real-world use cases illustrate the versatility and applicability of each type keyword in various programming scenarios, ranging from basic data storage to complex data manipulation and control flow within applications.

Contextual Keywords

these are considered keyword,only when used in specific context. they are not reserved keywords so can be used as names and identifiers.

Sure! Here is the information for the contextual keywords:

`add` Keyword

Definition

The `add` keyword is used to define a custom event accessor that provides the logic for subscribing to an event.

Use Case

Used in scenarios where you need custom logic for event subscription, such as logging when an event is subscribed to or adding additional behavior.

Code Snippet

```
using System;

public class CustomEvent
{
    private EventHandler _customEvent;

    public event EventHandler CustomEvent
    {
        add
        {
            Console.WriteLine("Event subscribed.");
            _customEvent += value;
        }
```

```
        remove
        {
            Console.WriteLine("Event unsubscribed.");
            _customEvent -= value;
        }
    }

    public void RaiseEvent()
    {
        _customEvent?.Invoke(this, EventArgs.Empty);
    }
}

// Usage
class Program
{
    static void Main()
    {
        CustomEvent customEvent = new CustomEvent();
        customEvent.CustomEvent += (sender, args) => { Console.WriteLine("Event
triggered!"); };
        customEvent.RaiseEvent();
    }
}
```

`var` Keyword

Definition

The `var` keyword is used to declare a variable with an implicit type, where the compiler determines the type based on the assigned value.

Use Case

Used when the type is evident from the right-hand side of the declaration or to simplify code readability, especially with complex types.

Code Snippet

```
using System;
using System.Collections.Generic;

public class Example
{
    public void PrintNames()
    {
        var names = new List<string> { "Alice", "Bob", "Charlie" };
        foreach (var name in names)
        {
            Console.WriteLine(name);
```

```
        }
    }
}

// Usage
class Program
{
    static void Main()
    {
        var example = new Example();
        example.PrintNames();
    }
}
```

`dynamic` Keyword

Definition

The `dynamic` keyword is used to declare a variable whose type is resolved at runtime rather than compile time.

Use Case

Used when the exact type is not known until runtime or when interacting with dynamic languages, COM objects, or APIs that return objects of various types.

Code Snippet

```
using System;

public class Example
{
    public void Display(dynamic value)
    {
        Console.WriteLine($"Value: {value}, Type: {value.GetType()}");
    }
}

// Usage
class Program
{
    static void Main()
    {
        var example = new Example();
        example.Display(123);
        example.Display("Hello, world!");
        example.Display(3.14);
    }
}
```

`global` Keyword

Definition

The `global` keyword is used to refer to the global namespace, which is the namespace at the outermost level.

Use Case

Used to disambiguate references to types or namespaces when there are conflicts with nested or local namespaces.

Code Snippet

```
using System;

namespace MyNamespace
{
    class Program
    {
        static void Main()
        {
            global::System.Console.WriteLine("Hello, global namespace!");
        }
    }
}
```

`set` Keyword

Definition

The `set` keyword defines the setter accessor of a property or indexer, which is used to assign a value.

Use Case

Used in properties to encapsulate the assignment logic, allowing validation, transformation, or other operations during the assignment.

Code Snippet

```
using System;

public class Person
{
    private int age;

    public int Age
    {
        get { return age; }
```

```
        set
        {
            if (value < 0 || value > 150)
                throw new ArgumentOutOfRangeException("Age must be between 0 and 150.");
            age = value;
        }
    }
}

// Usage
class Program
{
    static void Main()
    {
        var person = new Person();
        person.Age = 30;
        Console.WriteLine($"Person's age: {person.Age}");
    }
}
```

`value` Keyword

Definition

The `value` keyword represents the value assigned to a property or indexer within the `set` accessor.

Use Case

Used in the `set` accessor to handle the value being assigned to the property, enabling custom logic during assignment.

Code Snippet

```
using System;

public class Rectangle
{
    private int width;
    private int height;

    public int Width
    {
        get { return width; }
        set
        {
            if (value <= 0)
                throw new ArgumentOutOfRangeException("Width must be positive.");
            width = value;
        }
```

```
    }

    public int Height
    {
        get { return height; }
        set
        {
            if (value <= 0)
                throw new ArgumentOutOfRangeException("Height must be positive.");
            height = value;
        }
    }

    public int Area
    {
        get { return width * height; }
    }
}

// Usage
class Program
{
    static void Main()
    {
        var rectangle = new Rectangle();
        rectangle.Width = 10;
        rectangle.Height = 20;
        Console.WriteLine($"Rectangle area: {rectangle.Area}");
    }
}
```

These contextual keywords provide flexibility and functionality to your C# code, enabling custom behaviors and simplifying syntax for common programming scenarios.