C# Advanced Topics

Reflection

Reflection in C# allows you to inspect and interact with the metadata of assemblies, types, and members at runtime. You can dynamically create instances of types, invoke methods, access fields and properties, and even manipulate attributes.

Use Cases: Reflection is used in scenarios like creating dynamic applications, building frameworks (e.g., dependency injection), or tools like serializers.

Key Classes:

- Type: Represents type declarations (classes, interfaces, arrays).
- Assembly: Represents an assembly, which is a collection of modules.
- MethodInfo, PropertyInfo, FieldInfo, ConstructorInfo: Represent members of a class.

Example:

```
Type myType = typeof(MyClass);
MethodInfo method = myType.GetMethod("MyMethod");
method.Invoke(myInstance, new object[] { /* parameters */ });
```

Reflection and Attributes

Attributes in C# provide a powerful way to add metadata to your code. Reflection allows you to read these attributes at runtime.

Pre-Defined Attributes

C# has several predefined attributes that are commonly used:

- [Obsolete]: Marks a method or class as obsolete.

- [Serializable]: Indicates that a class can be serialized.

- [DllImport]: Used for P/Invoke to call unmanaged code.

Example:

```
[Obsolete("Use NewMethod instead")]
public void OldMethod() { }
```

Custom Attributes

You can create custom attributes by inheriting from System.Attribute.

Example:

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Method)]
public class MyCustomAttribute : Attribute
{
    public string Description { get; }
    public MyCustomAttribute(string description)
    {
        Description = description;
    }
}
```

Using Reflection to Retrieve Attributes:

```
Type type = typeof(MyClass);
var attributes = type.GetCustomAttributes(typeof(MyCustomAttribute), true);
```

Invoking Members Using Reflection with Binding Options

Binding options control how reflection searches for members. Common options include BindingFlags.Public, BindingFlags.NonPublic, BindingFlags.Instance, and BindingFlags.Static.

Example:

```
var method = typeof(MyClass).GetMethod("MyMethod", BindingFlags.NonPublic | BindingFlags.Instance);
method.Invoke(myInstance, null);
```

Declaring and Using Delegates

Delegates are type-safe function pointers. They are used to pass methods as arguments to other methods.

Example:

```
public delegate void MyDelegate(string message);
public void MyMethod(string message) { /*...*/ }
MyDelegate del = MyMethod;
del("Hello");
```

Singlecast & Multicast Delegates

- Singlecast Delegates: Point to a single method.
- Multicast Delegates: Point to multiple methods.

Example:

```
MyDelegate del = Method1;
del += Method2;  // Now del is a multicast delegate
del("Hello");
```

Anonymous Delegates

Anonymous delegates allow you to define a delegate inline without a named method.

Example:

```
MyDelegate del = delegate(string message) { Console.WriteLine(message); };

del("Hello");
```

Covariance and Contravariance

Covariance and contravariance provide flexibility in assigning delegates and interfaces.

- Covariance: Allows a method to return a more derived type.
- Contravariance: Allows a method to accept parameters of a less derived type.

Example:

```
Func<object> objFunc = () => "Hello";

Func<string> strFunc = objFunc;  // Covariance


Action<string> strAction = (string s) => Console.WriteLine(s);

Action<object> objAction = strAction;  // Contravariance
```

Async Callbacks

Asynchronous callbacks are used to execute a method asynchronously, typically with a delegate or Task.

Example:

```
public delegate void MyCallback(int result);
public void StartAsyncOperation(MyCallback callback)
{
    Task.Run(() =>
    {
        int result = LongRunningOperation();
        callback(result);
    });
}
```

Declaring & Handling Custom Events

Events in C# are based on delegates and provide a way for a class to notify other classes or objects when something happens.

Example:

```
public delegate void MyEventHandler(object sender, EventArgs e);
public event MyEventHandler MyEvent;

protected virtual void OnMyEvent()
{
    MyEvent?.Invoke(this, EventArgs.Empty);
}
```

Multithreading Overview

Multithreading allows a program to run multiple threads concurrently, making it possible to perform multiple tasks simultaneously.

Programming Threads

In C#, threads are represented by the Thread class.

Example:

```
Thread t = new Thread(new ThreadStart(MyMethod));
t.Start();
```

Thread Priority

Thread priority determines the order in which threads are scheduled for execution.

Example:

```
t.Priority = ThreadPriority.Highest;
```

Suspend, Resume, Interrupt, and Cross-Thread Operations

- Suspend/Resume: Suspending and resuming threads is deprecated due to safety issues.
- Interrupt: Used to interrupt a thread in a waiting state.
- Cross-Thread Operations: Accessing UI elements from a thread that did not create them can lead to issues.

Background & Foreground Threads

Foreground threads keep the application alive, while background threads do not.

Example:

```
t.IsBackground = true;
```

Thread Pool

The thread pool provides a collection of worker threads that can be used to execute tasks.

Example:

```
ThreadPool.QueueUserWorkItem(MyMethod);
```

Synchronization Using Monitor

Monitor provides a mechanism for synchronizing access to objects by multiple threads.

Example:

```
lock (lockObject)
{
    // Critical section
}
```

Synchronization Using Mutex

A mutex is similar to a lock but can work across multiple processes.

Example:

```
using (Mutex mutex = new Mutex(false, "MyMutex"))
{
    mutex.WaitOne();
    // Critical section
    mutex.ReleaseMutex();
}
```

Lock Statement

The lock statement is shorthand for using Monitor.

Example:

```
lock (lockObject)
{
    // Critical section
}
```

Synchronization Using Semaphore & Events

- Semaphore: Limits the number of threads that can access a resource.
- Events: Used for signaling between threads.

Example:

```
Semaphore semaphore = new Semaphore(0, 3);
semaphore.WaitOne();
// Critical section
semaphore.Release();
```