Attributes

1. Introduction to Attributes

- Attributes provide a powerful way to add metadata and declarative information to code elements.
  - Adding descriptive information to types and members
  - Changing runtime behavior through metadata
  - Enabling framework features (serialization, COM interop, etc.)

2. Core Attribute Concepts

2.1 Attribute Targets

- Attributes can be applied to various program elements:

```
[Serializable]                  // Class
public class MyClass {
    [Obsolete("Use NewMethod instead")]  // Method
    public void OldMethod() {}

    [JsonProperty("user_name")]  // Property
    public string UserName { get; set; }

    [NonSerialized]              // Field
    public int Value;
}
```

2.2 Built-in Attribute Categories

1. **Serialization**: [Serializable], [DataContract]
2. **COM Interop**: [ComVisible], [Guid]

3. **Compiler Directives**: `[Conditional]`, `[Obsolete]`
4. **Testing**: `[TestMethod]`, `[DataRow]`
5. **Security**: `[Authorize]`, `[AllowAnonymous]`

3. Creating Custom Attributes

3.1 Basic Attribute Definition

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Method,
                AllowMultiple = false,
                Inherited = true)]
public class AuditAttribute : Attribute
{
    public string Category { get; }
    public AuditLevel Level { get; set; } = AuditLevel.Info;

    public AuditAttribute(string category)
    {
        Category = category;
    }
}

public enum AuditLevel { Info, Warning, Error }
```

3.2 Attribute Usage Parameters

- **AttributeTargets**: Where the attribute can be applied
- **AllowMultiple**: Whether multiple instances are allowed
- **Inherited**: Whether derived classes inherit the attribute

4. Attribute Reflection

4.1 Reading Attributes at Runtime

```
Type type = typeof(MyClass);

// Get class-level attributes
var attrs = type.GetCustomAttributes(typeof(AuditAttribute), false);

// Get method-specific attributes
MethodInfo method = type.GetMethod("Process");
var methodAttrs = method.GetCustomAttributes<ObsoleteAttribute>();
```

4.2 Practical Example: API Documentation Generator

```
public class DocumentationAttribute:Attribute {
  public DocumentationAttribute(string desc) {
    Description = desc;
  }
  public string Description { get; set; }
}

[Documentation("Represents a customer entity")]
public class Customer {
    [Documentation("Unique identifier")]
    public int Id { get; set; }

    [Documentation("Customer's full name")]
    public string Name { get; set; }
}

public static void GenerateDocs(Type type) {
    var classDoc = type.GetCustomAttribute<DocumentationAttribute>();
    Console.WriteLine($"Class: {type.Name} - {classDoc?.Description}");
```

```
        foreach (var prop in type.GetProperties()) {
            var propDoc = prop.GetCustomAttribute<DocumentationAttribute>();
            Console.WriteLine($"  {prop.Name}: {propDoc?.Description}");
        }
    }
```

MSDN References

- Creating Custom Attributes
- Attribute Usage
- Retrieving Attributes

`async` and `await`

1. Introduction

- Introduced in **.NET Framework 4.5 (2012)** with C# 5.0 to simplify **asynchronous programming**.
- Before `async/await`, developers used `Task`, `ThreadPool`, or `Begin/End` **async patterns**, which were complex and error-prone.
- Designed to make asynchronous code **read like synchronous code** while avoiding thread-blocking calls.
- Built on the **Task Parallel Library (TPL)** and `Task<T>` type for representing asynchronous operations.

2. Key Concepts & Definitions

- `async` **Modifier**: Marks a method as asynchronous, allowing the use of `await`.
- `await` **Keyword**: Pauses method execution until the awaited `Task` completes, **without blocking the thread**.
- **Task**: Represents an asynchronous operation (e.g., `Task`, `Task<T>`).
- **Task Statuses**:
  - `RanToCompletion` (success),
  - `Faulted` (exception),
  - `Canceled` (via `CancellationToken`).
- `ConfigureAwait(false)`: Optimizes performance by avoiding unnecessary context switching.

3. How `async/await` Works Under the Hood

- The compiler **rewrites** `async` methods into a **state machine**, preserving local variables across `await` points.
- No additional threads are created unless explicitly required (e.g., `Task.Run`).
- Uses **SynchronizationContext** (UI thread in WPF/WinForms) or **ThreadPool** (console apps) for continuations.

4. Advantages

- **Simplified Code**: No more nested callbacks ("callback hell").
- **Improved Scalability**: Frees up threads during I/O operations (e.g., HTTP requests, file reads).
- **Better Error Handling**: Exceptions are propagated naturally (unlike event-based async).

5. Common Pitfalls & Best Practices

- **Avoid** `async void` (except for event handlers) → Prevents proper exception handling.
- **Prefer** `Task.WhenAll` over sequential `await` for independent tasks.
- **Use** `CancellationToken` to support task cancellation.
- **Beware of deadlocks** (e.g., `.Result` or `.Wait()` on UI thread).

6. Example

```
// Example 1: Basic async/await
public async Task<string> FetchDataAsync(string url) {
    HttpClient client = new HttpClient();
    string result = await client.GetStringAsync(url);
    return result;
}

// Usage - in another async method (can be async Main())
string url = "https://raw.githubusercontent.com/nilesh-g/learn-web/refs/heads/main/data/novels.json";
string result = await FetchDataAsync(url);
Console.WriteLine(result);
// JSON processing ...
```

```
// Example 2: Parallel async tasks
public async Task ProcessMultipleRequestsAsync() {
    Task<string> task1 = FetchDataAsync("url1");
    Task<string> task2 = FetchDataAsync("url2");
    string[] results = await Task.WhenAll(task1, task2);
}
```

7. Exception Handling

- Exceptions in `async` methods are **captured in the `Task`** (use `try/catch` with `await`):

```
try {
    await FaultyMethodAsync();
}
catch (HttpRequestException ex) {
    Console.WriteLine($"Failed: {ex.Message}");
}
```

8. Performance Considerations

- **Overhead**: State machine generation adds minor memory/CPU cost (negligible for I/O-bound work).
- **Thread Pool Usage**: CPU-bound work should use `Task.Run` to avoid UI thread starvation.

9. MSDN References

- Asynchronous Programming (C#)
- Task Class
- Async/Await Best Practices