**Exercise 1: Use Method Overriding Correctly**

**Problem Statement:** Create a parent class Animal with a method MakeSound(). Then, create a Dog class that overrides this method using override.

**Steps to Follow:**

1. Define a MakeSound() method in the Animal class.
2. Override it in the Dog class with override.
3. Instantiate Dog and call MakeSound().

| using System; using System.Reflection;  class Animal {  public virtual void MakeSound()  {  Console.WriteLine("Animal makes a sound.");  } }  class Dog : Animal {  public override void MakeSound()  {  Console.WriteLine("Dog barks: Woof! Woof!");  } }  class Program {  static void Main()  {  Dog myDog = new Dog();  myDog.MakeSound(); // Calls the overridden method in Dog class   // Using Reflection to get method details  MethodInfo methodInfo = typeof(Dog).GetMethod("MakeSound");  Console.WriteLine("\nMethod Name: " + methodInfo.Name);  Console.WriteLine("Declaring Type: " + methodInfo.DeclaringType);  Console.WriteLine("Is Overridden: " + (methodInfo.DeclaringType != typeof(Animal)));  } } |
| --- |

**Exercise 2: Use Obsolete Attribute to Mark an Old Method**

**Problem Statement:** Create a class LegacyAPI with an old method OldFeature(), which should not be used anymore. Instead, introduce a new method NewFeature().

**Steps to Follow:**

1. Define a class LegacyAPI.
2. Mark OldFeature() as [Obsolete].
3. Call both methods and observe the warning.

| using System;  class LegacyAPI {  [Obsolete("OldFeature() is deprecated. Use NewFeature() instead.")]  public void OldFeature()  {  Console.WriteLine("This is the old feature. It should not be used.");  }   public void NewFeature()  {  Console.WriteLine("This is the new recommended feature.");  } }  class Program {  static void Main()  {  LegacyAPI api = new LegacyAPI();    api.OldFeature(); // This will show a warning  api.NewFeature(); // This is the new method  } } |
| --- |

**Exercise 3: Suppress Warnings for Unchecked Operations**

**Problem Statement:** Create an ArrayList without generics and use #pragma warning disables to hide compilation warnings.

| using System; using System.Collections;  class Program {  static void Main()  {  #pragma warning disable CS8600 // Disable warning for potential null assignment  #pragma warning disable CS8602 // Disable warning for possible null reference  #pragma warning disable CS0618 // Disable warning for obsolete code (if needed)   ArrayList list = new ArrayList(); // Non-generic collection    list.Add(10); // Adding int  list.Add("Hello"); // Adding string  list.Add(3.14); // Adding double   Console.WriteLine("ArrayList elements:");  foreach (var item in list)  {  Console.WriteLine(item);  }   #pragma warning restore CS8600   #pragma warning restore CS8602   #pragma warning restore CS0618   } } |
| --- |

**Exercise 4: Create a Custom Attribute and Use It**

**Problem Statement:** Create a custom attribute TaskInfo to mark tasks with priority and assigned person.

**Steps to Follow:**

1. Define an attribute TaskInfo with fields Priority and AssignedTo.
2. Apply this attribute to a method in TaskManager class.
3. Retrieve the attribute details using Reflection.

| using System; using System.Reflection;  // Step 1: Define the custom attribute [AttributeUsage(AttributeTargets.Method)] public class TaskInfoAttribute : Attribute {  public int Priority { get; }  public string AssignedTo { get; }   public TaskInfoAttribute(int priority, string assignedTo)  {  Priority = priority;  AssignedTo = assignedTo;  } }  // Step 2: Apply the attribute to a method public class TaskManager {  [TaskInfo(1, "Aarushi")]  public void CompleteTask()  {  Console.WriteLine("Task Completed.");  } }  // Step 3: Retrieve attribute details using reflection class Program {  static void Main()  {  Type type = typeof(TaskManager);  MethodInfo method = type.GetMethod("CompleteTask");   if (method != null)  {  var attributes = method.GetCustomAttributes(typeof(TaskInfoAttribute), false);  foreach (TaskInfoAttribute attr in attributes)  {  Console.WriteLine($"Task Priority: {attr.Priority}");  Console.WriteLine($"Assigned To: {attr.AssignedTo}");  }  }  } } |
| --- |

**Exercise 5: Create and Use a Repeatable Attribute**

**Problem Statement:** Define an attribute BugReport that can be applied multiple times on a method.

**Steps to Follow:**

1. Define BugReport with a Description field.
2. Use AllowMultiple = true to allow multiple bug reports.
3. Apply it twice on a method.
4. Retrieve and print all bug reports.

| using System; using System.Reflection;  // Step 1: Define the repeatable attribute [AttributeUsage(AttributeTargets.Method, AllowMultiple = true)] public class BugReportAttribute : Attribute {  public string Description { get; }   public BugReportAttribute(string description)  {  Description = description;  } }  // Step 2: Apply the attribute multiple times on a method public class SoftwareModule {  [BugReport("NullReferenceException occurs when input is null.")]  [BugReport("Performance issue when processing large datasets.")]  public void ProcessData()  {  Console.WriteLine("Processing data...");  } }  // Step 3: Retrieve and print all bug reports using reflection class Program {  static void Main()  {  Type type = typeof(SoftwareModule);  MethodInfo method = type.GetMethod("ProcessData");   if (method != null)  {  var attributes = method.GetCustomAttributes(typeof(BugReportAttribute), false);  foreach (BugReportAttribute attr in attributes)  {  Console.WriteLine($"Bug Report: {attr.Description}");  }  }  } } |
| --- |

**Practice Problems for Custom Attributes in C#**

**Beginner Level**

**1️⃣ Create an Attribute to Mark Important Methods**

**Problem Statement:** Define a custom attribute ImportantMethod that can be applied to methods to indicate their importance.

**Requirements:**

1. Define ImportantMethod with an optional Level parameter (default: "HIGH").
2. Apply it to at least two methods.
3. Retrieve and print annotated methods using Reflection.

| using System; using System.Reflection;  // Step 1: Define the custom attribute [AttributeUsage(AttributeTargets.Method)] public class ImportantMethodAttribute : Attribute {  public string Level { get; }   public ImportantMethodAttribute(string level = "HIGH")  {  Level = level;  } }  // Step 2: Apply the attribute to multiple methods public class SampleClass {  [ImportantMethod]  public void CriticalTask()  {  Console.WriteLine("Executing critical task...");  }   [ImportantMethod("MEDIUM")]  public void NormalTask()  {  Console.WriteLine("Executing normal task...");  }   public void RegularTask()  {  Console.WriteLine("Executing regular task...");  } }  // Step 3: Retrieve and print important methods using reflection class Program {  static void Main()  {  Type type = typeof(SampleClass);  MethodInfo[] methods = type.GetMethods();   foreach (MethodInfo method in methods)  {  var attribute = method.GetCustomAttribute<ImportantMethodAttribute>();  if (attribute != null)  {  Console.WriteLine($"Important Method: {method.Name}, Level: {attribute.Level}");  }  }  } } |
| --- |

**2️⃣ Create a Todo Attribute for Pending Tasks**

**Problem Statement:** Define an attribute Todo to mark pending features in a project.

**Requirements:**

* The attribute should have fields:
  + Task (string) → Description of the task
  + AssignedTo (string) → Developer responsible
  + Priority (default: "MEDIUM")
* Apply it to multiple methods.
* Retrieve and print all pending tasks using Reflection.

| using System; using System.Reflection;  // Step 1: Define the custom Todo attribute [AttributeUsage(AttributeTargets.Method, AllowMultiple = true)] public class TodoAttribute : Attribute {  public string Task { get; }  public string AssignedTo { get; }  public string Priority { get; }   public TodoAttribute(string task, string assignedTo, string priority = "MEDIUM")  {  Task = task;  AssignedTo = assignedTo;  Priority = priority;  } }  // Step 2: Apply the attribute to multiple methods public class ProjectModule {  [Todo("Implement user authentication", "Aarushi", "HIGH")]  [Todo("Fix session timeout issue", "Rohan", "CRITICAL")]  public void AuthenticateUser()  {  Console.WriteLine("Authenticating user...");  }   [Todo("Optimize database queries", "Sneha")]  public void FetchData()  {  Console.WriteLine("Fetching data...");  }   [Todo("Improve UI responsiveness", "Raj", "LOW")]  public void UpdateUI()  {  Console.WriteLine("Updating UI...");  }   public void CompletedFeature()  {  Console.WriteLine("This feature is already completed.");  } }  // Step 3: Retrieve and print pending tasks using reflection class Program {  static void Main()  {  Type type = typeof(ProjectModule);  MethodInfo[] methods = type.GetMethods();   foreach (MethodInfo method in methods)  {  var attributes = method.GetCustomAttributes<TodoAttribute>();  foreach (var attr in attributes)  {  Console.WriteLine($"Pending Task: {attr.Task}");  Console.WriteLine($"Assigned To: {attr.AssignedTo}");  Console.WriteLine($"Priority: {attr.Priority}");  Console.WriteLine($"Method: {method.Name}");  Console.WriteLine("------------------------------");  }  }  } } |
| --- |

**Intermediate Level**

**3️⃣ Create an Attribute for Logging Method Execution Time**

**Problem Statement:** Define an attribute LogExecutionTime to measure method execution time.

**Requirements:**

* Apply LogExecutionTime to a method.
* Use Stopwatch before and after execution.
* Print execution time.
* Apply it to different methods and compare the time taken.

| using System; using System.Diagnostics; using System.Reflection;  // Step 1: Define the custom LogExecutionTime attribute [AttributeUsage(AttributeTargets.Method)] public class LogExecutionTimeAttribute : Attribute {  public LogExecutionTimeAttribute() { } }  // Step 2: Apply the attribute to multiple methods public class TaskProcessor {  [LogExecutionTime]  public void TaskA()  {  // Simulate task processing with a delay  System.Threading.Thread.Sleep(1000);  Console.WriteLine("Task A completed.");  }   [LogExecutionTime]  public void TaskB()  {  // Simulate task processing with a delay  System.Threading.Thread.Sleep(500);  Console.WriteLine("Task B completed.");  }   public void TaskC()  {  // Simulate task processing without logging  System.Threading.Thread.Sleep(200);  Console.WriteLine("Task C completed.");  } }  // Step 3: Create a custom attribute handler to log execution time class Program {  static void Main()  {  Type type = typeof(TaskProcessor);  MethodInfo[] methods = type.GetMethods();   foreach (MethodInfo method in methods)  {  // Check if the method has the LogExecutionTime attribute  var attribute = method.GetCustomAttribute<LogExecutionTimeAttribute>();  if (attribute != null)  {  // Measure and log execution time  Stopwatch stopwatch = new Stopwatch();  stopwatch.Start();   method.Invoke(Activator.CreateInstance(type), null);   stopwatch.Stop();  Console.WriteLine($"Execution Time for {method.Name}: {stopwatch.ElapsedMilliseconds} ms");  Console.WriteLine("--------------------------------");  }  }  } } |
| --- |

**4️⃣ Create a MaxLength Attribute for Field Validation**

**Problem Statement:** Define a field-level attribute MaxLength(int value) that restricts the maximum length of a string field.

**Requirements:**

* Apply it to a User class field (Username).
* Validate length in the constructor.
* Throw ArgumentException if the limit is exceeded.

| using System; using System.Reflection;  // Step 1: Define the MaxLength attribute [AttributeUsage(AttributeTargets.Field | AttributeTargets.Property)] public class MaxLengthAttribute : Attribute {  public int Value { get; }   public MaxLengthAttribute(int value)  {  Value = value;  } }  // Step 2: Define the User class and apply the MaxLength attribute public class User {  [MaxLength(10)]  public string Username { get; }   // Constructor with validation  public User(string username)  {  var usernameField = typeof(User).GetProperty(nameof(Username));  var maxLengthAttribute = usernameField.GetCustomAttribute<MaxLengthAttribute>();   if (maxLengthAttribute != null && username.Length > maxLengthAttribute.Value)  {  throw new ArgumentException($"Username cannot be longer than {maxLengthAttribute.Value} characters.");  }   Username = username;  } }  // Step 3: Test the implementation class Program {  static void Main()  {  try  {  var user1 = new User("ShortName");  Console.WriteLine($"User created with username: {user1.Username}");   var user2 = new User("ThisUsernameIsWayTooLong");  Console.WriteLine($"User created with username: {user2.Username}");  }  catch (ArgumentException ex)  {  Console.WriteLine($"Error: {ex.Message}");  }  } } |
| --- |

**Advanced Level**

**5️⃣ Implement Role-Based Access Control with RoleAllowed**

**Problem Statement:** Define a class-level attribute RoleAllowed to restrict method access based on roles.

**Requirements:**

* [RoleAllowed("ADMIN")] should only allow ADMIN users to execute the method.
* Simulate user roles and validate access before invoking the method.
* If a non-admin tries to access it, print Access Denied!

| using System; using System.Reflection;  // Step 1: Define the RoleAllowed attribute [AttributeUsage(AttributeTargets.Method)] public class RoleAllowedAttribute : Attribute {  public string Role { get; }   public RoleAllowedAttribute(string role)  {  Role = role;  } }  // Step 2: Define the User class with role simulation public class User {  public string Role { get; set; }   public User(string role)  {  Role = role;  } }  // Step 3: Define the class with methods to simulate access control public class SystemAccess {  [RoleAllowed("ADMIN")]  public void AdminMethod()  {  Console.WriteLine("Admin method executed.");  }   [RoleAllowed("USER")]  public void UserMethod()  {  Console.WriteLine("User method executed.");  }   public void TryExecuteMethod(User user, string methodName)  {  MethodInfo method = this.GetType().GetMethod(methodName);  var roleAllowedAttribute = method.GetCustomAttribute<RoleAllowedAttribute>();   if (roleAllowedAttribute != null)  {  if (roleAllowedAttribute.Role == user.Role)  {  method.Invoke(this, null);  }  else  {  Console.WriteLine("Access Denied!");  }  }  else  {  Console.WriteLine("No role restriction on this method.");  method.Invoke(this, null);  }  } }  // Step 4: Simulate user roles and test the access control class Program {  static void Main()  {  User adminUser = new User("ADMIN");  User regularUser = new User("USER");   SystemAccess systemAccess = new SystemAccess();   // Try executing methods with different users  Console.WriteLine("Admin trying to access AdminMethod:");  systemAccess.TryExecuteMethod(adminUser, "AdminMethod");   Console.WriteLine("\nUser trying to access AdminMethod:");  systemAccess.TryExecuteMethod(regularUser, "AdminMethod");   Console.WriteLine("\nUser trying to access UserMethod:");  systemAccess.TryExecuteMethod(regularUser, "UserMethod");   Console.WriteLine("\nAdmin trying to access UserMethod:");  systemAccess.TryExecuteMethod(adminUser, "UserMethod");  } } |
| --- |

**6️⃣ Implement a Custom Serialization Attribute JsonField**

**Problem Statement:** Define an attribute JsonField to mark fields for JSON serialization.

**Requirements:**

* [JsonField(Name = "user\_name")] should map field names to custom JSON keys.
* Apply it on a User class.
* Write a method to convert an object to a JSON string by reading the attributes.

| using System; using System.Collections.Generic; using System.Reflection; using System.Text;  // Step 1: Define the JsonField attribute [AttributeUsage(AttributeTargets.Field | AttributeTargets.Property)] public class JsonFieldAttribute : Attribute {  public string Name { get; }   public JsonFieldAttribute(string name)  {  Name = name;  } }  // Step 2: Define the User class with JsonField attributes public class User {  [JsonField("user\_name")]  public string Username { get; set; }   [JsonField("user\_age")]  public int Age { get; set; }   [JsonField("user\_email")]  public string Email { get; set; }   public User(string username, int age, string email)  {  Username = username;  Age = age;  Email = email;  } }  // Step 3: Implement serialization method public class JsonSerializer {  public static string Serialize(object obj)  {  StringBuilder jsonString = new StringBuilder();  jsonString.Append("{");   // Get all fields and properties of the object  var fields = obj.GetType().GetProperties();  bool isFirst = true;   foreach (var field in fields)  {  var jsonFieldAttribute = field.GetCustomAttribute<JsonFieldAttribute>();  if (jsonFieldAttribute != null)  {  // If not the first element, add a comma  if (!isFirst)  jsonString.Append(", ");   // Get the custom name from the attribute  jsonString.Append($"\"{jsonFieldAttribute.Name}\": \"{field.GetValue(obj)}\"");  isFirst = false;  }  }   jsonString.Append("}");  return jsonString.ToString();  } }  // Step 4: Test the implementation class Program {  static void Main()  {  User user = new User("JohnDoe", 25, "johndoe@example.com");   string jsonString = JsonSerializer.Serialize(user);  Console.WriteLine(jsonString);  } } |
| --- |

**7️⃣ Implement a Custom Caching System with CacheResult**

**Problem Statement:** Define CacheResult to store method return values and avoid repeated execution.

**Requirements:**

* Apply CacheResult to a computationally expensive method.
* Implement a cache (Dictionary) to store previously computed results.
* If a method is called with the same input, return the cached result instead of recomputation.

| using System; using System.Collections.Generic; using System.Reflection;  // Step 1: Define the CacheResult attribute [AttributeUsage(AttributeTargets.Method)] public class CacheResultAttribute : Attribute { }  // Step 2: Implement a class to manage caching public class CacheManager {  private static Dictionary<string, object> \_cache = new Dictionary<string, object>();   public static object GetFromCache(string key)  {  return \_cache.ContainsKey(key) ? \_cache[key] : null;  }   public static void AddToCache(string key, object value)  {  if (!\_cache.ContainsKey(key))  {  \_cache.Add(key, value);  }  } }  // Step 3: Define the class with computationally expensive methods public class ComputationService {  private Random \_random = new Random();   // Example of an expensive method  [CacheResult]  public int ExpensiveComputation(int input)  {  // Simulate a time-consuming computation  Console.WriteLine("Performing computation...");  int result = \_random.Next(1, 1000) \* input; // Fake computation  return result;  }   public int CachedExpensiveComputation(int input)  {  var method = typeof(ComputationService).GetMethod("ExpensiveComputation");  var cacheResultAttribute = method.GetCustomAttribute<CacheResultAttribute>();   if (cacheResultAttribute != null)  {  string cacheKey = input.ToString();   // Check if the result is cached  var cachedResult = CacheManager.GetFromCache(cacheKey);  if (cachedResult != null)  {  Console.WriteLine("Returning cached result...");  return (int)cachedResult;  }   // Otherwise, compute the result and cache it  int result = ExpensiveComputation(input);  CacheManager.AddToCache(cacheKey, result);  return result;  }   return ExpensiveComputation(input);  } }  // Step 4: Test the caching system class Program {  static void Main()  {  ComputationService service = new ComputationService();   Console.WriteLine("First call with input 5:");  int result1 = service.CachedExpensiveComputation(5);  Console.WriteLine($"Result: {result1}");   Console.WriteLine("\nSecond call with input 5 (should use cache):");  int result2 = service.CachedExpensiveComputation(5);  Console.WriteLine($"Result: {result2}");   Console.WriteLine("\nCall with input 10:");  int result3 = service.CachedExpensiveComputation(10);  Console.WriteLine($"Result: {result3}");   Console.WriteLine("\nCall with input 5 (should use cache again):");  int result4 = service.CachedExpensiveComputation(5);  Console.WriteLine($"Result: {result4}");  } } |
| --- |