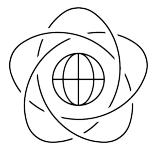
Programming 3 Advanced

Attributes, Tuples, Anonymous Classes
Events Workshop

Tomasz Herman

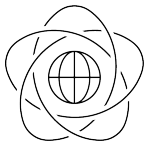
Faculty of Mathematics and Information Science
Warsaw University of Technology

Lecture 7, 26 listopada 2024



Outline

1. Attributes
2. Anonymous Types
3. Tuples



Attributes

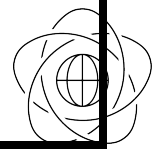
```
1  [Description("""
2      Attributes add metadata to your program.
3      Metadata is information about the types defined
4          in a program.
5      You can apply one or more attributes to entire
6          assemblies, modules, or smaller program
7          elements such as classes and properties.
8      Attributes can accept arguments in the same way
9          as methods and properties.
10     Your program can examine its own metadata or the
11         metadata in other programs by using reflection.
12     """)]
13 public static void AttributesIntroduction() {}
```

Documentation

https://learn.microsoft.com/en-us/dotnet/csharp/advanced-topics/reflection-and-attributes/

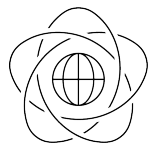
Attributes Targets

```
1  [assembly: AssemblyDescription("Attributes demonstration")]
2
3  [field: NonSerialized]
4  private static bool Property { get; } = false;
5
6  [Description("By default it applies to a method")]
7  private static int Method1() { return 0; }
8
9  [method: Description("Explicitely applied to a method")]
10 private static int Method2() { return 0; }
11
12 private static int Method3(
13     [Description("Applied to a parameter")] string parameter)
14 { return 0; }
15
16 [return: Description("Applied to a return value")]
17 private static int Method4() { return 0; }
```



Multiple Attributes

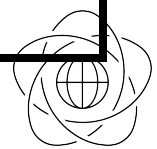
```
1  [Description("Multiple attributes applied to a method"),
2   Conditional("DEBUG")]
3  private static void Method5() { }
4
5  [Description("Multiple attributes applied to a method")]
6  [Conditional("DEBUG")]
7  private static void Method6() { }
```



Caller Attributes

Caller Info

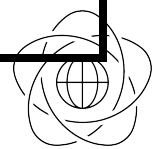
```
1  public static void CallerInfo()
2  {
3      Logger.Log("Hello, World!");
4      // [CallerInfo] /some/path/Program.cs:3: Hello, World!
5  }
6
7  public static void Log(string message,
8      [CallerMemberName] string memberName = "",
9      [CallerFilePath] string sourceFilePath = "",
10     [CallerLineNumber] int sourceLineNo = 0)
11 {
12     Console.WriteLine(
13       $"[{memberName}] {sourceFilePath}:{sourceLineNo}: {message}"
14     );
15 }
```



Caller Attributes

Caller Argument Expression

```
1  public static void CallerArgument()
2  {
3      Assertion.Assert("Hello, World!" is { Length: < 5 });
4      // Assertion failed: "Hello, World!" is { Length: < 5 }
5  }
6
7  public static void Assert(bool condition,
8      [CallerArgumentExpression(nameof(condition))]
9      string? message = null)
10 {
11     if (!condition)
12     {
13         Console.Error.WriteLine($"Assertion failed: {message}");
14     }
15 }
```



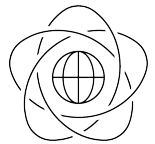
Anonymous Types

An anonymous type is a simple class created by the compiler

```
1  var bob = new { Name = "Bob", Age = 23 };
2  Console.WriteLine(
3    $"{bob.GetType()} is {bob.Name} and {bob.Age} years old.");
```

Documentation

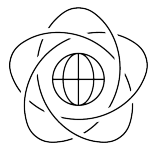
https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/types/anonymous-types



Inferring Property Names

The property name can be inferred from an identifier

```
1  int age = 23;
2  var bob = new { Name = "Bob", age, age.ToString().Length };
3  Console.WriteLine($"His {nameof(bob.Name)} is {bob.Name}");
4  Console.WriteLine($"His {nameof(bob.age)} is {bob.age}");
5  Console.WriteLine(
6    $"His {nameof(bob.Length)} is {bob.Length}");
```



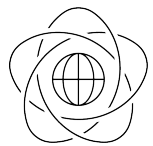
Equals is overridden to perform structural equality comparison

While == operator performs referential comparison

```
1 var a = new { X = 1, Y = 2, Z = 3 };
2 var b = new { X = 1, Y = 2, Z = 3 };
3 Console.WriteLine(a.Equals(b)); // True
4 Console.WriteLine(a == b); // False
```

GetHashCode and ToString

Anonymous types also override the GetHashCode and ToString methods.

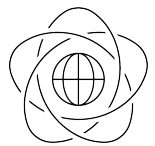


Non-destructive Mutation

C# 10

`with` **keyword allows to create a copy of another anonymous type.**

```
1  var a = new { X = 1, Y = 2, Z = 3 };
2  var b = a with { X = 10, Y = 20 };
3  Console.WriteLine($"a = (X = {a.X}, Y = {a.Y}, Z = {a.Z})");
4  Console.WriteLine($"b = (X = {b.X}, Y = {b.Y}, Z = {b.Z})");
```



Tuples

Tuples are value types, with mutable (read/write) elements

```
1  (double, int) t1 = (4.5, 3);
2  Console.WriteLine(
3    $"Tuple with elements {t1.Item1} and {t1.Item2}.");
4  // Output:
5  // Tuple with elements 4.5 and 3.
6
7  (double Sum, int Count) t2 = (4.5, 3);
8  Console.WriteLine(
9    $"Sum of {t2.Count} elements is {t2.Sum}.");
10 // Output:
11 // Sum of 3 elements is 4.5.
```

Documentation

https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/value-tuples

Tuple Field Names

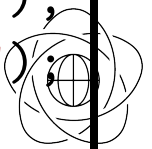
```
1  var t = (Sum: 4.5, Count: 3);
2  Console.WriteLine($"Sum of {t.Count} elements is {t.Sum}.");
3  (double Sum, int Count) d = (4.5, 3);
4  Console.WriteLine($"Sum of {d.Count} elements is {d.Sum}.");
```

It may be inferred from the name of the corresponding variable

```
1  var sum = 4.5;
2  var count = 3;
3  var t = (sum, count);
4  Console.WriteLine($"Sum of {t.count} elements is {t.sum}.");
```

The default names of tuple fields are Item1, Item2, Item3...

```
1  var a = 1;
2  var t = (a, b: 2, 3);
3  Console.WriteLine($"The 1st element is {t.Item1} ({t.a}).");
4  Console.WriteLine($"The 2nd element is {t.Item2} ({t.b}).");
5  Console.WriteLine($"The 3rd element is {t.Item3}.");
```



Tuple

Backing type

C# tuples are backed by System.ValueTuple type

```
1 ValueTuple<string, int> bob = ("Bob", 23);
2 (string, int) alice = ("Alice", 42);
3 (string name, int age) chad = ValueTuple.Create("Chad", 19);
4 alice = bob = chad;
```

Non default field names

At compile time, the compiler replaces non-default field names with the corresponding default names. As a result, explicitly specified or inferred field names aren't available at run time.
With methods/properties that return named tuple types, the compiler emits the element names by applying a custom attribute called TupleElementNamesAttribute to the member's return type.

Tuple equality

Tuple types support the == and != operators.

```
1  (int a, byte b) left = (5, 10);
2  (long a, int b) right = (5, 10);
3  Console.WriteLine(left == right);  // output: True
4  Console.WriteLine(left != right);  // output: False
5
6  var t1 = (A: 5, B: 10);
7  var t2 = (B: 5, A: 10);
8  Console.WriteLine(t1 == t2);  // output: True
9  Console.WriteLine(t1 != t2);  // output: False
```

Equals and GetHashCode

Tuples also override the Equals and GetHashCode method, making it practical to use tuples as keys in dictionaries.

Tuple Deconstructtion

```
1  var bob = ("Bob", 23);
2  {   // Deconstructs tuple fields into separate variables
3      (string name, int age) = bob;
4  }
5  {   // Might use existing variables
6      string name; int age;
7      (name, age) = bob;
8  }
9  {   // Might use var to deduct types
10     var (name, age) = bob;
11 }
```