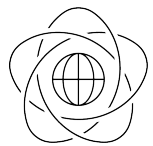
Programming 3 Advanced

Records, Preprocessor directives, Exceptions, Reflection

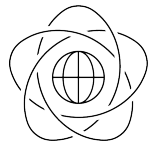
Tomasz Herman

Faculty of Mathematics and Information Science
Warsaw University of Technology

Lecture 8, 26 listopada 2024



Outline



Records

Records

- designed to work with immutable data
- supports non-destructive mutation
- eliminates boilerplate code
- follows value-based equality semantics

```
1 public record Person(string FirstName, string LastName);
2 public readonly record struct Point(double X, double Y);
3 public record struct Vector3(double X, double Y, double Z);
```

Documentation

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

Totorial

https://learn.microsoft.com/en-us/dotnet/csharp/tutorials/records

To enforce value semantics, the compiler generates:

- An override of Object.Equals(Object)
- A virtual Equals method whose parameter is the record type
- An override of Object.GetHashCode()
- Methods for operator == and operator !=
- Record types implement System.IEquatable<T>

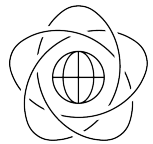
Other Compiler Generated Methods

The compiler also generates a ToString Method, a protected Copy Constructor (to support non-destructive mutation) and a Deconstruct method.

Records

Value Semantics

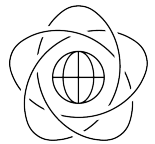
```
1  public record Person(string FirstName, string LastName);
2  public readonly record struct Point(double X, double Y);
3  public record struct Vector3(double X, double Y, double Z);
4
5  var john = new Person("John", "Doe");
6  var doe = new Person("John", "Doe");
7  Point p1 = new (10, 10), p2 = new (10, 10);
8  Vector3 v1 = new (1, 0, 1), v2 = new (1, 0, 1);
9  Console.WriteLine($"Person: {john == doe}"); // True
10 Console.WriteLine($"Point: {p1 == p2}");     // True
11 Console.WriteLine($"Vector3: {v1 == v2}");   // True
```



Records

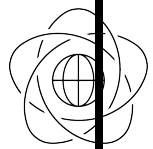
Non-destructive Mutation

```
1  public record Person(string FirstName, string LastName);
2  public readonly record struct Point(double X, double Y);
3  public record struct Vector3(double X, double Y, double Z);
4
5  Point p1 = new Point (1, 1);
6  Point p2 = p1 with { Y = 2 };
7  Console.WriteLine(p1);
8  Console.WriteLine(p2);
```



Customizing Records

```
1  public record Product(string Name, decimal Price)
2  {
3      public int Quantity { get; set; }
4
5      public Product(Product original)
6      {
7          Name = original.Name;
8          Price = original.Price;
9          Quantity = 0;
10     }
11
12     public override string ToString() =>
13         $"{Name}({Quantity}): {Price:C}";
14 }
15 Product apple = new Product("Apple", 1.99m) { Quantity = 5 };
16 Product copy = apple with {Price = 2.99m};
17 Console.WriteLine(apple);
18 Console.WriteLine(copy);
```

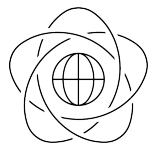


Preprocessor Directives

- Conditional Compilation
- Conditional Attribute
- Nullable Context
- Regions
- Pragma Warning
- Error and Warning

Documentation

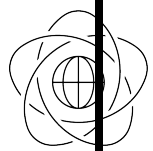
`https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/preprocessor-directives`



Conditional Compilation

```
#define TEST
#undef TRACE
#if TEST
    Console.WriteLine("TEST is defined");
#endif
#if !TEST
    Console.WriteLine("TEST is not defined");
#endif
#if TEST && DEBUG
    Console.WriteLine("DEBUG and TEST is defined");
#else
    Console.WriteLine("DEBUG or TEST is not defined");
#endif
```

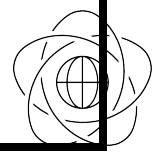
```
<PropertyGroup>
    <DefineConstants>TRACE;DEMO</DefineConstants>
</PropertyGroup>
```



Conditional Compilation

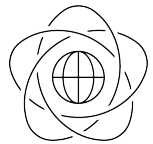
Attribute

```
1  [Conditional("TRACE"), Conditional("DEBUG")]
2  public static void Trace(string message)
3  {
4      Console.WriteLine(message);
5  }
6
7  [Conditional("DEBUG")]
8  public static void Debug(string message)
9  {
10     Console.WriteLine(message);
11 }
12
13 public static void Info(string message)
14 {
15     Console.WriteLine(message);
16 }
```



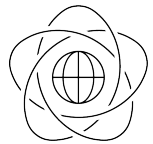
Nullable Context

```
1  #nullable disable
2      string str1 = null;
3  #nullable enable
4      string str2 = null!;
5  #nullable restore
6      string str3 = null!;
```



Regions

```
1  #region REGION_EXPLANATION
2  private static void Region()
3  {
4      PrintCurrentMethodName();
5      Console.WriteLine("""
6          #region is a preprocessor directive
7          that is used to group related pieces
8          of code together in a way that can
9          be collapsed or expanded in the code editor.
10         """);
11 }
12 #endregion
```

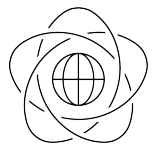


Pragma Warning

```
1 #pragma warning disable CS0219 // unused variable
2         string str = "Hello, world!";
3 #pragma warning restore CS0219
```

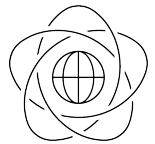
Warning Codes

https://github.com/thomaslevesque/GenerateCSharpErrors/blob/master/CSharpErrorsAndWarnings.md



Error and warning

```
1  #if !DEMO
2      #error DEMO is not defined
3          Console.WriteLine("This code will not "+
4              "compile if demo is not defined.");
5  #else
6      #warning DEMO is not defined
7          Console.WriteLine("This code will generate " +
8              "a warning message if demo is defined.");
9  #endif
```



Exceptions

Documentation

Fundamentals

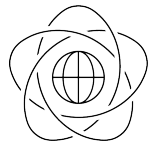
https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/exceptions/

Language Specification

https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/exceptions

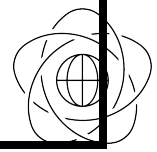
try, catch, finally, throw

https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/statements/exception-handling-statements



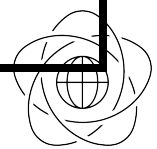
Catching exceptions

```
1  try
2  {
3      byte b = byte.Parse(input);
4      Console.WriteLine(b);
5  }
6  catch (ArgumentNullException)
7  {
8      Console.WriteLine("Input is null");
9  }
10 catch (FormatException)
11 {
12     Console.WriteLine("Not a number");
13 }
14 catch (OverflowException)
15 {
16     Console.WriteLine("More than a byte");
17 }
```



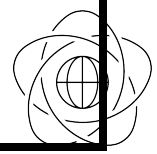
Throwing exceptions

```
1  public static void ValidateInput(string input, int maxLength)
2  {
3      if (string.IsNullOrWhiteSpace(input))
4      {
5          throw new ArgumentNullException(nameof(input),
6              "Input cannot be null or empty.");
7      }
8
9      if (input.Length > maxLength)
10     {
11         throw new ArgumentOutOfRangeException(nameof(input),
12             "Input exceeds the maximum length of" +
13             $"{maxLength} characters.");
14     }
15 }
```



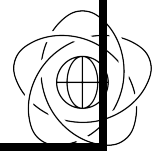
Rethrowing exception

```
1  try
2  {
3      ValidateInput("Hello, world", 16);
4  }
5  catch (ArgumentNullException e)
6  {
7      Console.WriteLine(e);
8  }
9  catch (ArgumentOutOfRangeException e)
10 {
11     Console.WriteLine(e);
12 }
13 catch (Exception e)
14 {
15     Console.WriteLine(e);
16     throw;
17 }
```



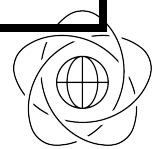
Exception filters

```
1  try
2  {
3      throw new HttpRequestException(
4          "Resource not found",
5          null,
6          System.Net.HttpStatusCode.NotFound);
7  }
8  catch (HttpRequestException ex) when
9      (ex.StatusCode == System.Net.HttpStatusCode.NotFound)
10 {
11     Console.WriteLine("Resource not found.");
12     Console.WriteLine(ex);
13 }
14 catch (HttpRequestException ex)
15 {
16     Console.WriteLine(ex);
17 }
```



Finally block

```
1  string tempFilePath = Path.GetTempFileName();
2  try
3  {
4      Console.WriteLine($"Temporary file created: {tempFilePath}");
5      File.WriteAllText(tempFilePath, "Temporary content.");
6  }
7  finally
8  {
9      if (File.Exists(tempFilePath))
10     {
11         File.Delete(tempFilePath);
12         Console.WriteLine("Temporary file deleted.");
13     }
14 }
```



Reflection

Reflection

C# program compiles into an assembly that includes metadata, compiled code, and resources. Inspecting the metadata and compiled code at runtime is called reflection.

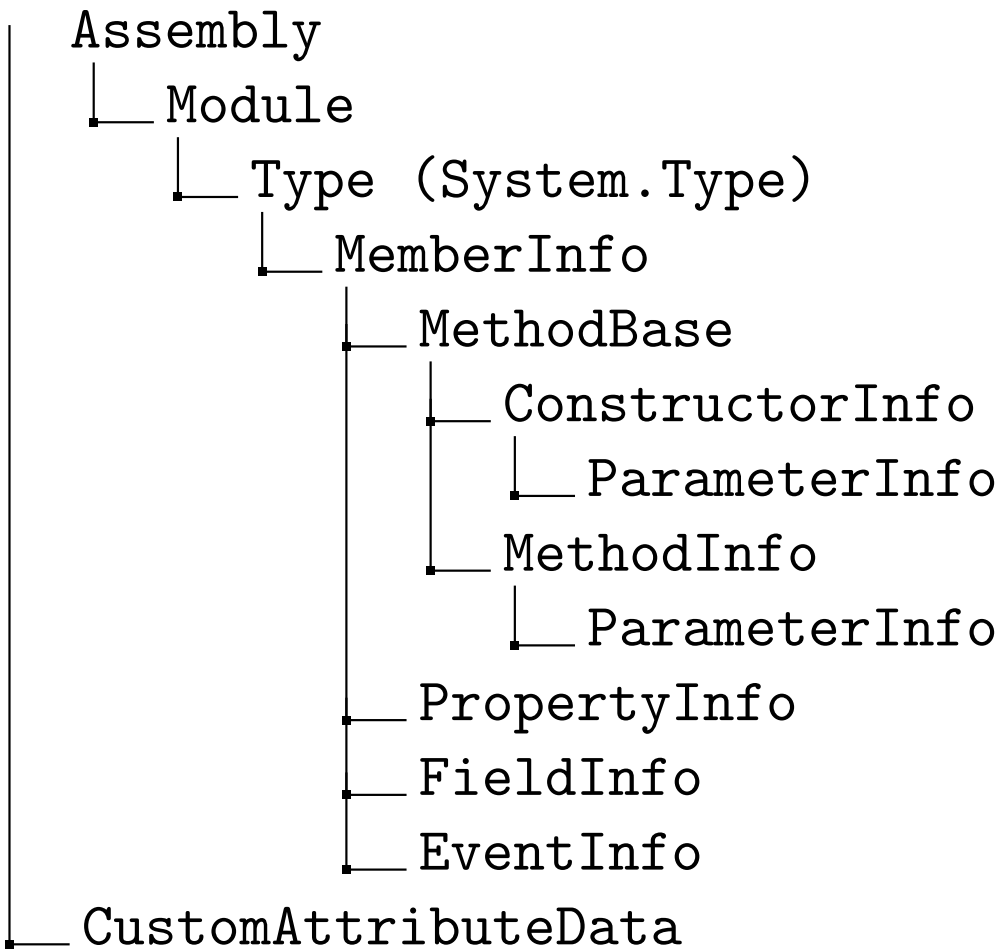
Namespaces

- `System.Type`
- `System.Reflection`
- `System.Reflection.Emit`

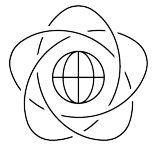
Documentation

`https://learn.microsoft.com/en-us/dotnet/fundamentals/reflection/reflection`

System.Reflection Namespace



- Assembly contains Modules.
- Modules contain Types.
- Types define Members (derived from MemberInfo), such as methods, fields, properties, and events.
- Methods and constructors contain Parameters.
- Everything can contain Attributes.



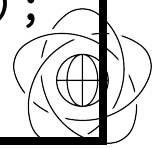
Querying Members

Using GetMembers()

```
1  MemberInfo[] members = typeof(Person).GetMembers();
2  foreach (MemberInfo member in members)
3  {
4      Console.WriteLine($"{member.MemberType,12}: {member}");
5  }
```

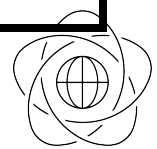
Alternatively Using GetTypeInfo().DeclaredMembers

```
1  IEnumerable<MemberInfo> members = typeof(Person)
2      .GetTypeInfo()
3      .DeclaredMembers;
4  foreach (MemberInfo member in members)
5  {
6      Console.WriteLine($"{member.MemberType,14}: {member}");
7  }
```



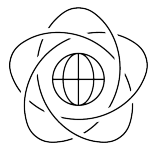
Dynamically Invoking a Member

```
1  Person person = new Person("John", "Doe", 19);
2  MethodInfo? method = typeof(Person)
3      .GetMethod(nameof(Person.IsAdult));
4  if (method is null)
5  {
6      Console.WriteLine(
7          $"Method {nameof(Person.IsAdult)} not found");
8  }
9  else
10 {
11     bool? isAdult = method
12         .Invoke(person, new object?[] { }) as bool?;
13     Console.WriteLine($"Is Adult: {isAdult}");
14 }
```



Dealing with parameters

```
1  ConstructorInfo? constructor = typeof(Person)
2      .GetConstructor(
3          [typeof(string), typeof(string), typeof(int)]);
4  Person? person = constructor?.Invoke(
5      ["John", "Doe", 42]) as Person;
6  Console.WriteLine(
7      $"Name: {person?.FullName}, Age: {person?.Age}");
```



Binding method to a delegate

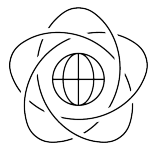
```
1  Person person = new Person("John", "Doe", 19);
2  MethodInfo? method = typeof(Person)
3      .GetMethod(nameof(Person.IsAdult));
4  // Binding to a delegate:
5  Func<bool> isAdult = (Func<bool>)Delegate
6      .CreateDelegate(typeof(Func<bool>), person, method);
7  for (int i = 0; i < 1_000_000; i++)
8  {
9      isAdult();
10 }
```

Performance

Late binding (choosing which member to invoke at runtime rather than compile time) is expensive. It is faster to bind method to a delegate, and then call the delegate because the late binding happens just once.

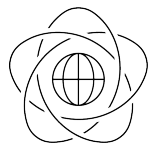
Querying non-public members

```
1  MemberInfo[] members = typeof(string).GetMembers(
2      BindingFlags.NonPublic | BindingFlags.Instance);
3  foreach (MemberInfo member in members)
4  {
5      Console.WriteLine($"{member.MemberType,12}: {member}");
6  }
```



Reflecting assemblies

```
1  Assembly? assembly;
2  assembly = Assembly.GetEntryAssembly();
3  assembly = Assembly.GetCallingAssembly();
4  assembly = Assembly.GetExecutingAssembly();
5  assembly = Assembly.GetAssembly(typeof(Person));
6  if (assembly is null) return;
7  foreach (var type in assembly.GetTypes())
8  {
9      Console.WriteLine(type.FullName);
10 }
```



Custom attributes

```
1  [AttributeUsage(AttributeTargets.Method)]
2  public class BenchmarkAttribute : Attribute
3  {
4      public int Repetitions { get; }
5
6      public BenchmarkAttribute(int repetitions = 1)
7      {
8          Repetitions = repetitions;
9      }
10 }
```

Documentation

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