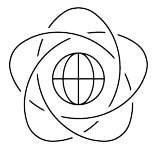
Programming 3 Advanced

Variables and Parameters, Namespaces, Custom Types

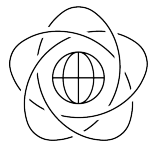
Tomasz Herman

Faculty of Mathematics and Information Science
Warsaw University of Technology

Lecture 3, 28 października 2024



Outline



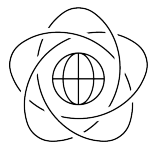
Stack and Heap

Stack allocated

- value types,
  - unless they are boxed,
  - part of an array,
  - part of a reference type,
  - or captured in some kind of closure

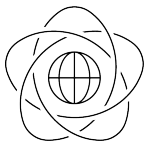
Heap allocated

- reference types
- arrays
- static variables
- value types
  - that are boxed,
  - part of an array,
  - part of a reference type,
  - or captured in some kind of closure



Definite Assignment

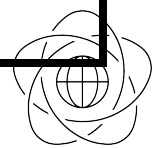
```
1  int x;
2  Console.WriteLine(x); // Compilation error
3
4  int[] ints = new int[2];
5  Console.WriteLine(ints[1]); // OK
6
7  Test test = new Test();
8  Console.WriteLine(test.X); // OK
9
10 public class Test { public int X; }
```



Default values

Type	**Default value**
Reference types (and nullable value types)	null
Numeric types	0
char	'\0'
bool	false
enum	0
struct	value with all fields set to default

```
1 float x = default;
2 Console.WriteLine(default(float));
```

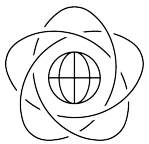


Parameters

Modifier	Passed by	Definite assignment	Notes
(None)	Value	before call	
ref	Reference	before call	
in	Reference	before call	Read-only
out	Reference	before returning	

Documentation

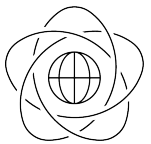
`https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/method-parameters`



Parameters

Passing by value

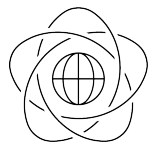
```
1  int x = 8;
2  Foo(x);
3  Console.WriteLine(x);
4
5  static void Foo(int p)
6  {
7      p = p + 1;
8      Console.WriteLine(p);
9  }
```



Parameters

Passing by value

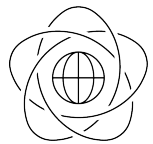
```
1  StringBuilder sb = new StringBuilder();
2  Foo(sb);
3  Console.WriteLine(sb.ToString());
4
5  static void Foo(StringBuilder foo)
6  {
7      foo.Append("test");
8      foo = null;
9  }
```



Parameters

`ref` keyword

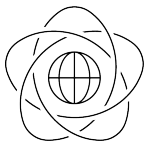
```
1  int x = 8;
2  Foo(ref x);
3  Console.WriteLine(x);
4
5  static void Foo(ref int p)
6  {
7      p = p + 1;
8      Console.WriteLine(p);
9  }
```



Parameters

`ref` keyword

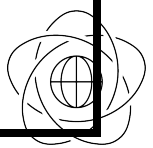
```
1  string foo = "Foo";
2  string bar = "bar";
3  Swap(ref foo, ref bar);
4  Console.WriteLine($"Foo: {foo}, bar: {bar}");
5
6  static void Swap(ref string a, ref string b)
7  {
8      string temp = a;
9      a = b;
10     b = a;
11 }
```



Parameters

out keyword

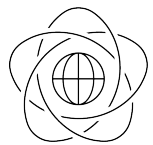
```
1  string[] firstNames; string lastName;
2  GetFirstAndLastNames("Tim Berners Lee",
3                       out firstNames, out lastName);
4  GetFirstAndLastNames("John Fitzgerald Kennedy",
5                       out var firsts, out string last);
6  GetFirstAndLastNames("Julius Robert Oppenheimer",
7                       out firsts, out _);
8
9  static void GetFirstAndLastNames(string name,
10                                  out string[] firstNames,
11                                  out string lastName)
12 {
13     string[] words = name.Split(' ');
14     firstNames = words[..^1];
15     lastName = words[^1];
16 }
```



Parameters

`in` keyword

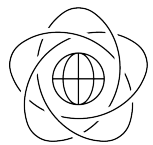
```
1  Matrix4 a = Matrix4.CreateRotationX(90.0f);
2  Matrix4 b = Matrix4.CreateRotationY(45.0f);
3  Multiply(in a, in b, out var result);
4  Console.WriteLine(result);
5
6  static void Multiply(in Matrix4 a,
7                       in Matrix4 b,
8                       out Matrix4 result)
9  {
10     /**/
11 }
```



Variable number of parameters

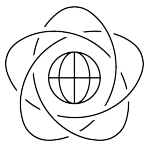
params keyword

```
1  var s1 = Concat("The", "Quick", "Brown", "Fox");
2  var s2 = Concat("Jumps", "Over", "The", "Lazy", "Dog");
3  Console.WriteLine(Concat(s1, s2));
4  Concat("This function accepts at least 1 parameter");
5
6  static string Concat(string str, params string[] strings)
7  {
8      StringBuilder sb = new StringBuilder(str);
9      foreach (string s in strings)
10         sb.Append(s);
11     return sb.ToString();
12 }
```



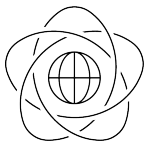
Optional and named parameters

```
1  public void ExampleMethod(int required,
2                            string optionalStr = "default",
3                            int optionalInt = 10) {}
4
5  ExampleMethod(3, "parameter", 7);
6  ExampleMethod(3, "parameter");
7  ExampleMethod(3);
8  ExampleMethod(3, optionalInt: 4);
```



Reference variables

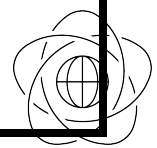
```
1 int[] storage = [1, 30, 7, 1557, 381];
2 ref int intRef = ref storage[3];
3 intRef = 20;
4 Console.WriteLine(intRef);
5 Console.WriteLine(storage[3]);
```



Reference variables

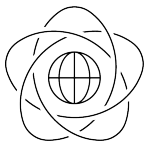
ref return

```
1  var storage = new IntStorage();
2  ref int max = ref storage.GetMax();
3
4  public class IntStorage
5  {
6      int[] storage = [1, 30, 7, 1557, 381];
7
8      public ref int GetMax()
9      {
10         ref int max = ref storage[0];
11         for (int i = 1; i < storage.Length; i++)
12             if (storage[i] > max)
13                 max = ref storage[i];
14         return ref max;
15     }
16 }
```



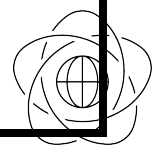
Implicitly-typed local variables

```
1 var greeting = "Hello class";
2 var i = 0;
3 var x = 0.15f;
4 var list = new List<int>();
5 var dayOfWeek = DateTime.Now.DayOfWeek;
```



Namespaces

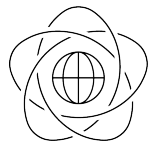
```
1  Outer.Middle.Inner.ClassA a;
2
3  namespace Outer
4  {
5      namespace Middle
6      {
7          namespace Inner
8          {
9              class ClassA {}
10         }
11     }
12 }
13
14 namespace Outer.Middle.Inner
15 {
16     class ClassB {}
17 }
```



File-scoped namespaces

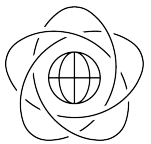
C# 10

```
1  namespace MyNamespace;
2
3  class ClassA {}
4  class ClassB {}
```



using directive

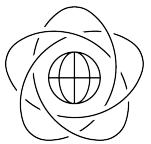
```
1  using Outer.Middle.Inner;
2
3  Outer.Middle.Inner.ClassB b1;
4  ClassB b2; // No need for full name
5
6  namespace Outer.Middle.Inner
7  {
8      class ClassB {}
9  }
```



global using directive

C# 10

```
1  global using Outer.Middle.Inner;
2  global using System;
3
4  Outer.Middle.Inner.ClassB b1;
5  ClassB b2; // No need for full name
6
7  namespace Outer.Middle.Inner
8  {
9      class ClassB {}
10 }
```

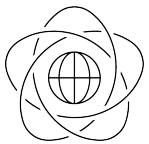


Implicit global `using` directive

.NET 6

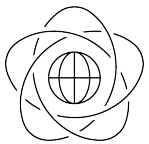
Automatically globally imported

- System
- System.Collections.Generic
- System.IO
- System.Linq
- System.Net.Http
- System.Threading
- System.Threading.Tasks
- others, depending on the SDK



using static directive

```
1  using static System.Console;
2  using static System.IO.FileAccess;
3
4  WriteLine("Hello, world!");
5  FileStream stream = new FileStream(name,
6                                     FileMode.Open,
7                                     Read, //FileAccess.Read
8                                     FileShare.ReadWrite);
```



Aliasing types and namespaces

Namespace alias

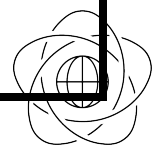
```
1 using Refl = System.Reflection;
2 Refl.PropertyInfo propInfo;
```

Type alias

```
1 using MyInt = System.Int32;
2 MyInt number = 42;
```

Any type alias (C# 12)

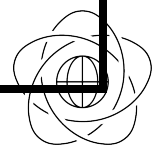
```
1 using NumberList = double[];
2 using PersonInfo = (string Name, int Age);
3 NumberList numbers = { -0.5, 0.5 };
4 PersonInfo person = ("Alice", 42);
```



Global namespace

```
1  global::System.Console.WriteLine("Hello world.");
```

```
1  namespace N
2  {
3      class A
4      {
5          static void Main()
6          {
7              System.Console.WriteLine (new A.B());
8              System.Console.WriteLine (new global::A.B());
9          }
10     public class B {}
11     }
12 }
13
14 namespace A
15 {
16     class B {}
17 }
```



Classes

```
1 // [modifiers] class [identifier] [generics and inheritance]
2 public class MyClass
3 {
4     // Fields, properties, methods...
5 }
```

Class modifiers (non-nested)

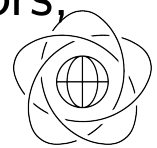
- public, internal, file, abstract, sealed, static, unsafe, and partial

Generics and inheritance

- generic type parameters and constraints, a base class, and interfaces.

Fields, properties, methods...

- methods, properties, indexers, events, fields, constants, constructors, overloaded operators, nested types, and a finalizer

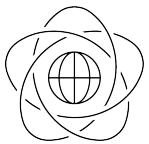


Fields

```
1  public class Person
2  {
3      public int Age = 0;
4      public string Name;
5  }
```

Field modifiers

- public, private, protected, internal
- static
- readonly
- required
- volatile
- new
- unsafe

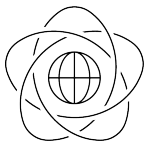


Constants

```
1  class Calendar
2  {
3      public const int Months = 12;
4      public const int Weeks = 52;
5      public const int Days = 365;
6  }
```

Constant modifiers

- public, private, protected, internal
- new

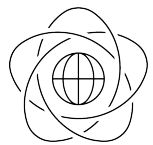


Methods

```
1  public class Account
2  {
3      private decimal _balance = 0.0m;
4      public decimal Deposit(decimal amount)
5      {
6          _balance += amount;
7          return _balance;
8      }
9  }
```

Method modifiers

- public internal private protected
- static
- new virtual abstract override sealed
- partial
- unsafe extern
- async

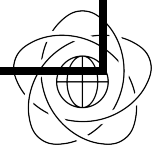


Methods

expression-bodied methods

```
1  public class Circle
2  {
3      public float Radius;
4      public decimal Area()
5      {
6          return float.Pi * Radius * Radius;
7      }
8  }
```

```
1  public class Circle
2  {
3      public float Radius;
4      public float Area() => float.Pi * Radius * Radius;
5  }
```

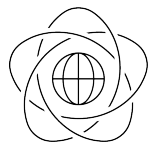


Constructors

```
1  Person person = new Person("Alice", 37);
2  public class Person
3  {
4      public int Age;
5      public string Name;
6      public Person(string name, int age)
7      {
8          Name = name;
9          Age = age;
10     }
11 }
```

Constructor modifiers

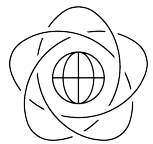
- public internal private protected
- unsafe extern



Constructors

expression-bodied constructors

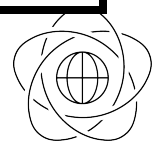
```
1  public class Person
2  {
3      public int Age;
4      public string Name;
5      public Person(string name, int age) =>
6          (Name, Age) = (name, age);
7  }
```



Constructors

overloading constructors

```
1  public class Person
2  {
3      public int Age;
4      public string Name;
5      public Person(string name, int age) : this(name)
6      {
7          Age = age;
8      }
9      public Person(string name)
10     {
11         Name = name;
12     }
13 }
```

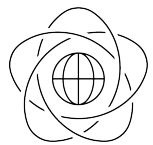


Properties

```
1  public class Player
2  {
3      public int Health { get; private set; } = 20;
4  }
5
6  Player player = new Player();
7  player.Health -= 5;
8  Console.WriteLine($"Health is {player.Health}");
9  player.Health = 0;
```

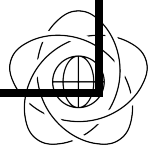
Property modifiers

- static
- public internal private protected
- new virtual abstract override sealed
- unsafe extern



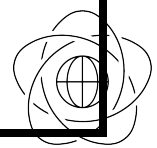
Deconstructors

```
1  public class Person
2  {
3      public int Age { get; set; }
4      public string Name { get; set; }
5
6      public void Deconstruct(out int age, out string name)
7      {
8          age = Age;
9          name = Name;
10     }
11 }
12
13 Person person = new Person {Age = 37, Name = "Bob"};
14 (int age, var name) = person;
15 // Equivalent call:
16 // person.Deconstruct(out int age, out var name);
```



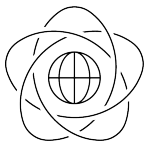
Object initializers

```
1  public class Hamster
2  {
3      public int Age { get; set; }
4      public string Name { get; set; }
5      public bool Friendly { get; set; }
6
7      public Hamster() {}
8      public Hamster(string name) => Name = name;
9  }
10
11 Hamster foo = new Hamster { Name = "Foo", Friendly = false };
12 Hamster boo = new Hamster("Boo") {Age = 1, Friendly = true };
13 // Equivalent:
14 Hamster temp = new Hamster();
15 temp.Name = "Foo";
16 temp.Friendly = false;
17 Hamster foo = temp;
```



this

```
1  public class Hamster
2  {
3      private string name;
4      public Hamster Mate { get; set; }
5
6      public Hamster(string name, Hamster partner)
7      {
8          this.name = name;
9          Mate = partner;
10         partner.Mate = this;
11     }
12 }
```



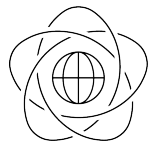
Primary constructors

C# 12

```
1  public class Person(int age, string name)
2  {
3      public int Age { get; set; } = age;
4      public string Name { get; set; } = name
5  }
6
7  Person person = new Person(37, "Alice");
```

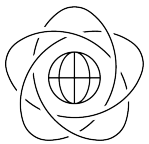
Documentation

https://learn.microsoft.com/en-us/dotnet/csharp/whats-new/tutorials/primary-constructors



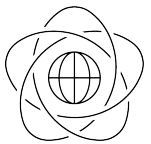
Static constructors

```
1  public class Person
2  {
3      public int Age { get; set; };
4      public string Name { get; set; }
5      private static int population = 0;
6
7      static Person() {
8          Console.Writeline("Static constructor");
9      }
10 }
11
12 Person person = new Person(37, "Alice");
```



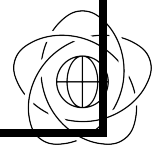
Static classes

```
1  public static class FileSystemUtils
2  {
3      public static string GetCwd()
4      {
5          // ...
6      }
7
8      public static string[] GetDirectories(string path)
9      {
10         // ...
11     }
12 }
```



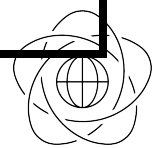
Finalizers

```
1  public class Person
2  {
3      public int Age { get; set; };
4      public string Name { get; set; }
5
6      ~Person()
7      {
8          Console.Writeline("Finalizer");
9      }
10
11     // Compilers rewrites the finalizer into:
12     // protected override void Finalize()
13     // {
14     //     Console.Writeline("Finalizer");
15     //     base.Finalize();
16     // }
17 }
```



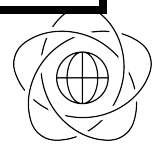
Partial class

```
1  // File Person.cs
2  public partial class Person
3  {
4      public int Age { get; set; };
5      public string Name { get; set; }
6  }
7
8  // File Person.gen.cs
9  public partial class Person
10 {
11     public void WhoAmI()
12     {
13         Console.WriteLine($"I'm {Name}, {Age}");
14     }
15 }
```



Partial methods

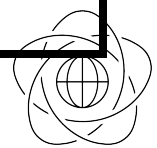
```
1  // File Person.cs
2  public partial class Foo
3  {
4      partial void Bar(int i);
5  }
6
7  // File Person.gen.cs
8  public partial class Foo
9  {
10     partial void Bar(int i)
11     {
12         Console.WriteLine($"Bar implementation: {i}");
13     }
14 }
```



Extended partial methods

C# 9

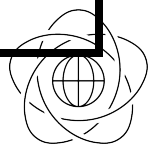
```
1  // File Foo.cs
2  public partial class Foo
3  {
4      public partial int Bar(string s, out int i);
5  }
6
7  // File Foo.gen.cs
8  public partial class Foo
9  {
10     public partial int Bar(string s, out int i);
11     {
12         // ...
13     }
14 }
```



Properties

overview

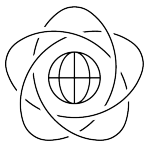
```
1  Player player = new Player();
2  player.Health = 20;
3  player.Health -= 4;
4
5  public class Player
6  {
7      private int health;
8
9      public int Health
10     {
11         get { return health; }
12         set { health = value; }
13     }
14 }
```



Properties

Auto-properties

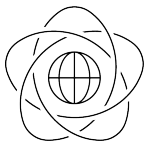
```
1  public class Person
2  {
3      public string Name { get; set; }
4      public int Age { get; set; } = 0;
5  }
```



Properties

Read-only and computed properties

```
1  public class Stock
2  {
3      public decimal CurrentPrice { get; set; }
4      public decimal Shares { get; set; }
5
6      public decimal Worth
7      {
8          get { return CurrentPrice * Shares; }
9      }
10 }
```

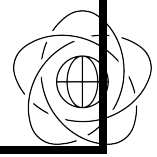


Properties

Expression-bodied properties

```
1  public class Stock
2  {
3      public decimal CurrentPrice { get; set; }
4      public decimal Shares { get; set; }
5
6      public decimal Worth => CurrentPrice * Shares;
7  }
```

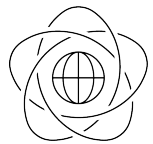
```
1  public class Stock
2  {
3      public decimal Worth
4      {
5          get => CurrentPrice * Shares;
6          set => Shares = value / CurrentPrice;
7      }
8  }
```



Properties

access modifiers

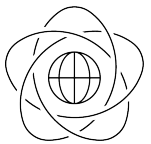
```
1  public class Player
2  {
3      public int Health { get; private set; }
4  }
```



Properties

Init only properties (C# 9)

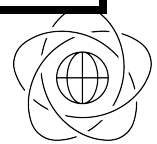
```
1  var person = new Person { Name = "Anne" };
2  person.Name = "Bob"; // Compilation time error
3
4  public class Person
5  {
6      public string Name { get; init; }
7      public int Age { get; set; } = 0;
8  }
```



Properties

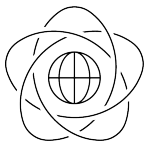
Read-only property initialization

```
1  var person = new Person("Anne");
2  person.Name = "Bob"; // Compilation time error
3
4  public class Person
5  {
6      public string Name { get; }
7      public int Age { get; set; } = 0;
8
9      public Person(string name)
10     {
11         Name = name;
12     }
13 }
```



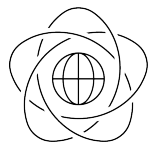
Structs

- It is a custom value type
- Does not support inheritance
- Can have same members as a class
  - except finalizers



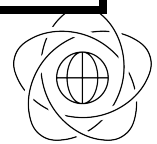
struct Construction

```
1  public struct Vector2
2  {
3      public float X = 1.0f;
4      public float Y;
5
6      public Vector2() => Y = -1.0f;
7  }
8
9  Vector2 p1 = new Vector2();
10 Vector2 p2 = default;
11 Console.WriteLine($"{p1.X}, {p1.Y}"); // 1, -1
12 Console.WriteLine($"{p2.X}, {p2.Y}"); // 0, 0
```



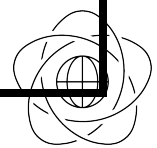
readonly structs and methods

```
1  public readonly struct Vector2
2  {
3      public float X, Y;
4      public void Foo() => X = 0.0f; // Compile error
5  }
6
7  public struct Vector3
8  {
9      public float X, Y, Z;
10     public readonly void Foo()
11     {
12         X = 1.0f; // Compile error
13     }
14 }
```



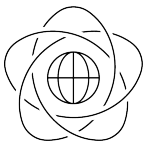
ref structs

```
1  public ref struct Vector2
2  {
3      public float X, Y;
4  }
5
6  public class Person
7  {
8      public Vector2 Position; // Error
9
10     public void Foo() {
11         Vector2 pos = new Vector2(); // OK, lives on stack
12         Vector2[] positions = new Vector2[10]; // Error
13     }
14 }
15
16
```



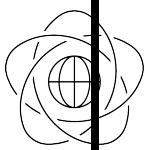
Interfaces

- similar to a class
- can define only functions and not fields
- its members are implicitly abstract
- a class or struct can implement multiple interfaces



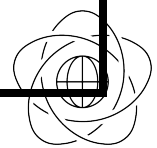
Interfaces

```
1  public interface IEnumerable
2  {
3      bool MoveNext();
4      object Current { get; }
5      void Reset();
6  }
7
8  public class Counter : IEnumerator
9  {
10     public int Count { get; private set; }
11     public Counter(int count) => Count = count;
12     public bool MoveNext() => Count-- > 0;
13     public object Current => Count;
14     public void Reset()
15     {
16         throw new NotSupportedException();
17     }
18 }
```



Interfaces

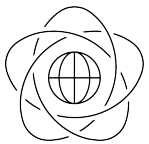
```
1  public class Counter : IEnumerator
2  {
3      public int Count { get; private set; }
4      public Counter(int count) => Count = count;
5      public bool MoveNext() => Count-- > 0;
6      public object Current => Count;
7      public void Reset()
8      {
9          throw new NotSupportedException();
10     }
11 }
12
13 IEnumerator e = new Counter(10);
14 while (e.MoveNext())
15     Console.Write(e.Current);
16 Console.WriteLine();
```



Interfaces

Extending Interfaces

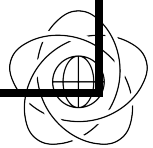
```
1  public interface IUndoable
2  {
3      void Undo();
4  }
5
6  public interface IRedoable : IUndoable
7  {
8      void Redo();
9  }
```



Interfaces

Explicit Implementation

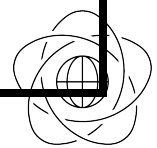
```
1  public interface IFoo1 { void Bar(); }
2  public interface IFoo2 { void Bar(); }
3
4  public class Foo : IFoo1, IFoo2
5  {
6      public void Bar()
7      {
8          Console.WriteLine("Implementation of IFoo1.Bar");
9      }
10
11     void IFoo2.Bar()
12     {
13         Console.WriteLine("Implementation of IFoo2.Bar");
14     }
15 }
```



Interfaces

boxing

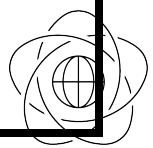
```
1  public interface IFoo1 { void Bar(); }
2
3  public struct FooStruct : IFoo1
4  {
5      public void Bar()
6      {
7          Console.WriteLine("Implementation of IFoo1.Bar");
8      }
9  }
10
11 FooStruct s = new FooStruct();
12 s.Bar(); // no boxing
13
14 IFoo1 foo1 = s; // boxing
15 foo1.Bar();
```



Interfaces

default members

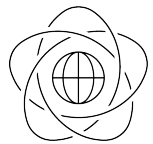
```
1  public interface ILogger
2  {
3      void Message(string message)
4      {
5          Console.WriteLine(message);
6      }
7  }
8
9  public class Logger : ILogger
10 {
11     public void Message(string message) // optional
12     {
13         Console.WriteLine(
14             $"{DateTime.Now:HH:mm:ss}: {message}");
15     }
16 }
```



Interfaces

static nonvirtual members

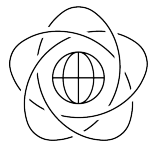
```
1  public interface ILogger
2  {
3      public const string DefaultFile = "default.log";
4      public static string LogPrefix { get; set; } = "Log: ";
5
6      void Message(string message)
7      {
8          Console.WriteLine($"{LogPrefix} {message}");
9      }
10 }
```



Interfaces

static virtual members (C# 11)

```
1  public interface IDescriptable
2  {
3      static abstract string Description { get; }
4      static virtual string Category => null;
5  }
6
7  public class Cat : IDescriptable
8  {
9      public static string Description => "It's a cat";
10     public static string Category => "Mammal"; // optional
11 }
```



Enums

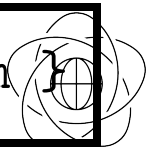
- It is a custom value type
- groups named numeric constants

Example

```
1  BorderSide side = BorderSide.Top;
2  if (side == BorderSide.Top)
3  {
4      Console.WriteLine("Top border side");
5  }
6
7  public enum BorderSide { Left, Right, Top, Bottom }
```

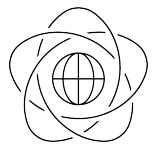
- By default underlying type is int
- The constants 0, 1, 2... are automatically assigned

```
1  public enum BorderSide : byte { Left=1,Right,Top=63,Bottom }
```



enum Conversions

```
1 int i = (int) BorderSide.Top; // requires explicit cast
2 BorderSide side = (BorderSide) i; // requires explicit cast
3 BorderSide side = 0; // might assign 0 implicitly
```



Flags enum

```
1  BorderSide allButTop = BorderSide.All ^ BorderSide.Top;
2  Console.WriteLine((allButTop & BorderSide.Left) != 0);
3  Console.WriteLine((allButTop & BorderSide.Right) != 0);
4  Console.WriteLine((allButTop & BorderSide.Top) != 0);
5  Console.WriteLine((allButTop & BorderSide.Bottom) != 0);
6
7  [Flags]
8  public enum BorderSide
9  {
10     None=0, Left=1, Right=1<<1, Top=1<<2, Bottom=1<<3,
11     LeftRight=Left|Right,
12     TopBottom=Top|Bottom,
13     All=LeftRight|TopBottom
14 }
```