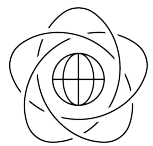
Programming 3 Advanced

Generics, Enumeration, Nullable

Tomasz Herman

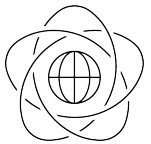
Faculty of Mathematics and Information Science
Warsaw University of Technology

Lecture 4, 28 października 2024



Outline

1. Generics
2. Enumeration
3. Nullable



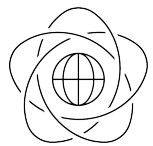
Generics

Generics in .NET

https://learn.microsoft.com/en-us/dotnet/standard/generics/

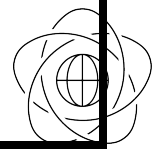
Generics Overview

https://learn.microsoft.com/en-us/dotnet/standard/generics



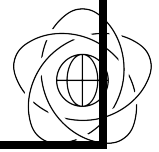
Generics

```
1  public class Stack<T>
2  {
3      private T[] _items = new T[100];
4      public int Count { get; private set; }
5      public void Push(T item) => _items[Count++] = item;
6      public T Pop() => _items[--Count];
7  }
8  Stack<int> intStack = new Stack<int>();
9  for (int i = 0; i < 10; i++)
10 {
11     intStack.Push(i);
12 }
13 while (intStack.Count > 0)
14 {
15     int item = intStack.Pop();
16     Console.Write(item);
17 }
```



Why not use *object*?

```
1  public class Stack
2  {
3      private object[] _items = new T[100];
4      public int Count { get; private set; }
5      public void Push(object item) => _items[Count++] = item;
6      public object Pop() => _items[--Count];
7  }
8  Stack intStack = new Stack();
9  for (int i = 0; i < 10; i++)
10 {
11     intStack.Push(i);
12 }
13 while (intStack.Count > 0)
14 {
15     int item = (int)intStack.Pop();
16     Console.Write(item);
17 }
```

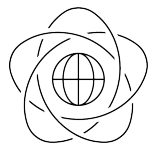


Generic methods

```
1 public static void Swap<T>(ref T a, ref T b)
2 {
3     T temp = a;
4     a = b;
5     b = temp;
6 }
```

Documentation

https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/generics/generic-methods



Generics constraints

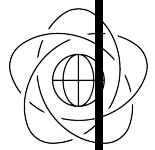
```
1  where T : base-class // Base-class constraint
2  where T : interface  // Interface constraint
3  where T : class      // Reference-type constraint
4  where T : class?     // Nullable reference-type constraint
5  where T : struct     // Reference-type constraint
6  where T : unmanaged  // Unmanaged constraint
7  where T : new()      // Parameterless constructor constraint
8  where U : T          // Naked type constraint
9  where T : notnull    // Non-nullable value type, or (C# 8+)
10                      // a non-nullable reference type
```

Documentation

https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/generics/constraints-on-type-parameters

Generics constraints

```
1  Console.WriteLine(Max(1, 5, 2, -4, 0, 210, 42));
2  Console.WriteLine(Max("the", "quick", "brown", "fox"));
3  // Compile time error, fruits do not implement IComparable:
4  // Console.WriteLine(Max(new Apple(), new Orange());
5
6  public static T Max<T>(T value, params T[] values)
7  where T : IComparable<T>
8  {
9      var max = value;
10     foreach (var t in values)
11     {   // we know T implements IComparable<T>
12         if (max.CompareTo(t) < 0)
13         {
14             max = t;
15         }
16     }
17     return max;
18 }
```



Generics Variance

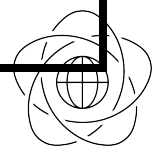
```
1  // Covariant T type parameter
2  // (can be used only as a return value)
3  public interface IPoppable<out T>
4  {
5      T Pop();
6  }
7
8  // Contravariant T type parameter
9  // (can be used only as an input parameter)
10 public interface IPushable<in T>
11 {
12     void Push(T item);
13 }
```

Documentation

https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/covariance-contravariance/variance-in-generic-interfaces

Generics Variance

```
1  // Covariance allows down-casting generic type
2  var apples = new VariantStack<AppleTree>();
3  apples.Push(new AppleTree());
4  apples.Push(new AppleTree());
5  IPoppable<Tree> poppableTrees = apples;
6
7  // Contravariance allows up-casting generic type
8  var trees = new VariantStack<Tree>();
9  trees.Push(new AppleTree());
10 trees.Push(new OrangeTree());
11 IPushable<AppleTree> pushableApples = trees;
12
13 private abstract class Tree;
14 private class AppleTree : Tree;
15 private class OrangeTree : Tree;
```



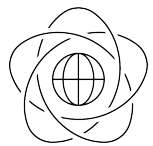
Enumerator

Enumerator

An enumerator is a read-only, forward-only cursor over a sequence of values.

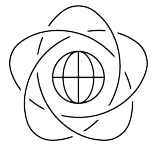
Type is an Enumerator if one of the conditions is fulfilled:

- has public parameterless method `bool MoveNext()` and `Current` property
- implements `System.Collections.Generics.IEnumerable<T>`
- implements `System.Collections.IEnumerable`



IEnumerator interfaces

```
1  public interface IEnumerator
2  {
3      bool MoveNext();
4      object Current { get; }
5      void Reset();
6  }
7
8  public interface IEnumerator<out T> : IDisposable,
9                                        IEnumerator
10 {
11     T Current { get; }
12 }
```



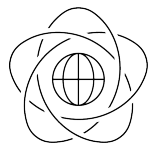
Enumerable

Enumerator

An enumerable object is the logical representation of a sequence. It is not itself a cursor but an object that produces cursors over itself.

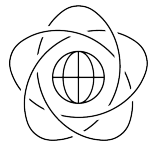
Type is an Enumerable if one of the conditions is fulfilled:

- has a public parameterless method named `GetEnumerator` that returns an enumerator
- implements `System.Collections.Generic.IEnumerable<T>`
- implements `System.Collections.IEnumerable`
- has an extension method named `GetEnumerator` that returns an enumerator



IEnumerable interface

```
1  public interface IEnumerable
2  {
3      IEnumerator GetEnumerator();
4  }
5  public interface IEnumerable<out T> : IEnumerable
6  {
7      IEnumerator<T> GetEnumerator();
8  }
```



foreach

```
1  List<int> collection = [1, 2, 3, 4, 5, 6, 7, 8];
2
3  // Collection needs to be 'Enumerable'
4  foreach(int item in collection)
5  {
6      Console.WriteLine(item.ToString());
7  }
8  // Is translated to (roughly):
9  IEnumerator<int> enumerator = collection.GetEnumerator();
10 while (enumerator.MoveNext())
11 {
12     var item = enumerator.Current;
13     Console.WriteLine(item.ToString());
14 }
```

Documentation

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

Iterator methods

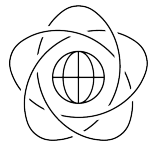
```
1  // return type might be either IEnumerable or IEnumerator
2  private static IEnumerable<int> Fibonacci(int n)
3  {
4      int current = 0, next = 1;
5      for (int i = 0; i < n; i++)
6      {
7          yield return current;
8          (current, next) = (next, current + next);
9      }
10 }
```

Iterators with Yield

https://learn.microsoft.com/en-us/archive/msdn-magazine/2017/june/essential-net-custom-iterators-with-yield

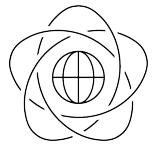
Null-Coalescing Operator

```
1  string s1 = null;
2  string s2 = s1 ?? "non-null";
3  Console.WriteLine($"Value of s2: {s2}");
4  // equivalent:
5  // string s2 = (s1 == null) ? "non-null": s1;
6
7  // Can also throw exception if s1 is null:
8  // s2 = s1 ?? throw new ArgumentNullException();
```



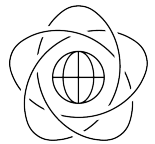
Null-Coalescing Assignment Operator

```
1  string s = null;
2  s ??= "non-null";
3  Console.WriteLine($"Value of s: {s}");
4
5  // equivalent:
6  // string s = (s == null) ? "non-null" : s;
```



Null-Conditional Operator

```
1 System.Text.StringBuilder sb = null;
2 string s = sb?.ToString();
3 // equivalent:
4 // string s = (sb == null ? null : sb.ToString());
```

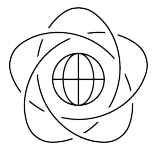


Nullable value types

```
1 // Cannot assign null to a value types:
2 // int integer = null; // Compile-time error
3
4 // Can assign null to a nullable value type (denoted with ?)
5 int? i = null;
6 Console.WriteLine (i == null); // True
```

Documentation

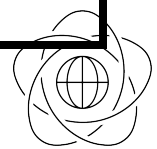
https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/nullable-value-types



Nullable struct

```
1  public struct Nullable<T> where T : struct
2  {
3      public T Value {get;}
4      public bool HasValue {get;}
5      public T GetValueOrDefault();
6      public T GetValueOrDefault(T defaultValue);
7      //...
8  }
```

```
1  int? i = null;
2  Console.WriteLine (i == null);
3  // Equivalent:
4  Nullable<int> i = new Nullable<int>();
5  Console.WriteLine (! i.HasValue);
```

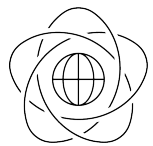


Nullable conversions

```
1  int? i = 5;       // implicit conversion
2  int j = (int)i;   // explicit conversion
3
4  // Equivalent:
5  // int j = i.Value;
```

Explicit conversion

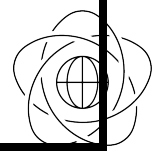
Both explicit conversion and accessing Value property might throw an exception if the value is null.



Nullable

Operator Lifting

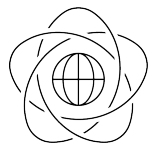
```
1  int? x = 5;
2  int? y = null;
3  // Equality operator examples
4  Console.WriteLine(x == y);    // false
5  Console.WriteLine(x == null); // false
6  Console.WriteLine(x == 5);    // true
7  Console.WriteLine(y == null); // true
8  Console.WriteLine(y == 5);    // false
9  Console.WriteLine(y != 5);    // true
10 // Relational operator examples
11 Console.WriteLine(x < 6);     // true
12 Console.WriteLine(y < 6);     // false
13 Console.WriteLine(y > 6);     // false
14 // All other operator examples
15 Console.WriteLine(x + 5);     // 10
16 Console.WriteLine(x + y);     // null
```



Nullable

bool? logic

```
1  bool? n = null;
2  bool? f = false;
3  bool? t = true;
4  Console.WriteLine (n | n); // null
5  Console.WriteLine (n | f); // null
6  Console.WriteLine (n | t); // true
7  Console.WriteLine (n & n); // null
8  Console.WriteLine (n & f); // false
9  Console.WriteLine (n & t); // null
```



Nullable Reference Types

```
1  #nullable enable
2  #nullable disable
3  #nullable restore
4
5  string s1 = null;  // Generates a compiler warning
6  string? s2 = null;  // OK, s2 is marked as nullable
7
8  class Foo
9  {
10     // Also a warning, since x is not initialized
11     string x;
12 }
```

Documentation

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

Nullable Reference Types

Null-forgiving operator

```
1  string s1 = null!;          // `!` Silences the warning
2  string? s2 = null;
3  int s2Length = s2!.Length; // `!` Silences the warning
4
5  // Either is better (exception-safe):
6  // int? s2Length2 = s2?.Length;
7  // if (s2 != null) Console.Write(s2.Length);
```