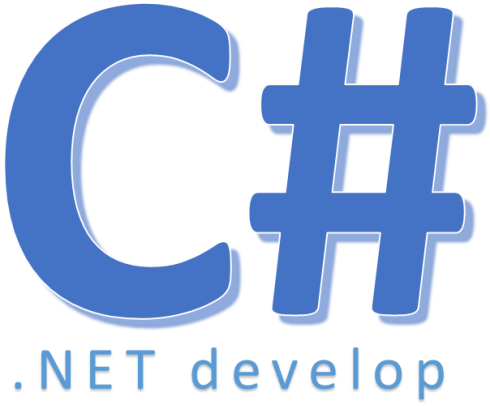
Fachhochschule Nordwestschweiz
Hochschule für Technik

C# ADVANCED

OPERATOR OVERLOADING, EXTENSION METHODS & MORE



Martin Kropp, Yves Senn
University of Applied Sciences Northwestern Switzerland

Learning Targets

- You
  - know and can explain the concepts Extension Methods, Operator Overloading, yield, and nullable types
  - can compare these concepts with corresponding concepts in the Java language
  - can apply the above concepts correctly for software development

Content

- Nullable Types
- Operator Overloading
- Extension Methods
- yield Keyword

More about

Reference- & Value-Types

Nullable types

`?` makes value-types nullable:

```
int a = null;
int? b = null;

b = 7;
if (b.HasValue)
    Console.WriteLine(b.Value);
```

Null-coalescing operator

- ?? is a binary operator that is part of a conditional expression

```
class Person
{
    public string Name { get; set; }
}
```

```
var p = new Person();

// so far
string name = (p.Name != null) ? p.Name : "Unknown";


// more compact using null coalescing operator
string name = p.Name ?? "Unknown";
```

More about null-coalescing operator

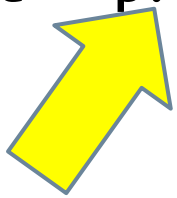
- Concatenate

```
// more compact using null coalescing operator
string name = someValue ?? someValue2 ?? "Unknown";
```

- Combine with nullable types

```
int? i = 3;
int x = i ?? 0;

// test if p is null included
Person p = new Person();
string name = p?.Name ?? "Unknown";
```



Nameof and null-coalescing operator

You can use the nameof operator to make the argument-checking code more maintainable:

```
public string Name
{
        get => name;
        set => name =
                value ??
                throw new ArgumentNullException(
                        nameof(value),
                        $"{nameof(Name)} cannot be null"
        );
}
```

Operator overloading

Example: Rational numbers

The “classic” approach:

```
var r1 = new Rational(1,2);
var r2 = new Rational(2,1);

var r3 = r1.AddRational(r2);

double d = Rational.ConvertToDouble(r3);
```

You write a special method

You write a static converter

What if you could write?

```
var r1 = new Rational(1,2);
var r2 = new Rational(2,3);

var r3 = r1 + r2;         // operator overloading
double d = (double) r3; // explicit conversion
Rational r2 = 2;          // implicit conversion
```

Operator overloading

Operator overloading allows you to define **`static`** custom operator implementations for various operators:

```
var c1 = new MyClass();
var c2 = new MyClass();
var c3 = c1 + c2;
```

Supported types for overloading:

- Unary operators: `+, -, !, ~, ++, --`
- Binary operators: `+, -, *, /, %, &, |, ^, <<, >>`
- Comparison operators: `==, !=, <, >, <=, >=`
- Implicit overloading: `+=, -=, &&, ||`
- Conversion operators: `implicit, explicit`

Example: Rational numbers

Let's start with this class:

```
public struct Rational
{
    public Rational(int n, int d) { … }

    public int Numerator   { get {…} }
    public int Denominator { get {…} }

    public override string ToString() { … }
}
```

Nothing special so far:

```
var r = new Rational(1,2);
var s = r.ToString();
```

Example: Rational numbers

Overloading operators:

```
public struct Rational
{
  …
  public static Rational operator* (Rational lhs, Rational rhs)
  {
     // here goes the implementation
     return new Rational(lhs.Numerator * rhs.Numerator,
                         lhs.Denominator * rhs.Denominator);
  }

  public static Rational operator+ (Rational lhs, Rational rhs)
  {
     …
  }
}

Rational r3 = r1 * r2;
r3 *= r2; // *= is provided for free, if you implement operator *
```

At least one parameter must be of enclosing type!

Equality operators

== and != operators:

```
public static bool operator== (Rational lhs, Rational rhs)
public static bool operator!= (Rational lhs, Rational rhs)
```

If value equality is needed, also override `Equals()` and GetHashCode() for consistency:

```
public override bool Equals(object o)

public override int GetHashCode()
```

```
if (r1.Equals(r2)) { … }

if (r1 == r2) { … }
```

```
if (!r1.Equals(r2)) { … }

if (r1 != r2)  { … }
```

Implicit type conversions

Handy for *lossless* conversions:

```
public struct Rational
{
    …
    public static implicit operator Rational(int i)
    {
        return new Rational(i,1);
    }
}
```

Example: From `int` to `Rational`

```
Rational r = 2;
```

Explicit type conversions

Beware: *lossy* conversions/exceptions:

```
public struct Rational
{
    …
    public static explicit operator double(Rational r)
    {
        return r.Numerator / (double)r.Denominator;
    }
}
```

Example: From `Rational` to `double`

```
var r = new Rational(2,3);
double d = (double)r;
```

Implicit vs. Explicit

	`explicit`	`implicit`
Usage	Typecast necessary → User actively forces conversion	Happens auto-magically → User might not be aware of conversion
Allowed effects	Might be lossy (rounding errors, one-way conversions, …)	Don't surprise users by losing information

Examples	`double` → `float`
	`float` → `int`
	`Rational` → `double`
		`float` → `double`
		`int` → `long`
		`int` → `Rational`

Worksheet – Part 1

Extension methods

- Adding methods to your own types is trivial
- Adding methods to existing types is hard
  - Subclassing is not always sensible
  - Static methods are in separate classes

Example:

Adding a method to the `string` class

Extension methods

Traditional way: A static method, passing arguments

```
public static class StringExtensions
{
    public static string Without(string text, char ch)
    {
        return string.Join("", text.Split(ch));
    }
}
```

Looks somehow cumbersome

Calling the static method:

```
var text = "Hxellxo";
Console.WriteLine(StringExtensions.Without(text, 'x'));
```

Extension methods

Why not like this?

Looks much more straightforward

```
var text = "Hexllxox";
var result = text.Without('x');
```

Allows you to add new methods to existing types

- *without* creating a new derived type
- *without* recompilation of existing code
- *without* modifying the original type

How to Do: Extension methods

Common Convention

```
public static class StringExtensions
{
    public static String Without(this string text, char ch)
    {
        return string.Join("", text.Split(ch));
    }
}
```

First param specifies which type you are extending

Call the method as if it was directly defined in `string`:

```
var s = "Hexllxox";
var result = s.Without('x');
```

or even

```
var result = "Hexllxox".Without('x');
```

Worksheet – Part 2

yield Keyword - Motivation

Method that incrementally computes and returns a sequence of values:

```
public static IEnumerable<int> GenerateNumbersClassic(int num)
{
    List<int> list = new List<int>();
    for (var i = 0; i < num; i++)
        list.Add(i);

    return list;
}
```

Quite some boilerplate code

```
static void Main()
{
    foreach (var a in GenerateNumbersClassic(10))
        Console.WriteLine(a);
}
```

yield Keyword

Method that incrementally computes and returns a sequence of values:

```
public static IEnumerable<int> GenerateNumbers(int num)
{
    for (var i = 0; i < num; i++)
        yield return i;
}
```

Much smaller

```
static void Main()
{
    foreach (var a in GenerateNumbers(10))
        Console.WriteLine(a);
}
```

yield Keyword

```
public void Consumer()
{
   foreach (var i in CreateSetOfIntegers())
      Console.WriteLine(i.ToString());
}

public IEnumerable<int> CreateSetOfIntegers()
{
   yield return 1;
   yield return 2;
   yield return 4;
   yield return 8;
   yield return 16;
   yield return 16777216;
}
```

yield & extension methods


```
static class IntExtensions
{
    public static IEnumerable<int> To(this int first, int last)
    {
        for (var i = first; i <= last; i++)
            yield return i;
    }
}


class Program
{
    static void Main()
    {
        foreach (var i in 3.To(10))
            Console.WriteLine(i);
    }
}
```

yield constraints

- Containing method **must declare** a `IEnumerator` or `IEnumerable` **return type**
- `unsafe` is not allowed
- No `ref` or `out` parameters
- `yield` cannot appear in anonymous methods

Worksheet – Part 3