TEKNOLOGISK
INSTITUT

Module 13

Advanced C# Language Features

TEKNOLOGISK
INSTITUT

What Are Nullable Types?

- Can assume the values of the underlying value type as well as `null`

```
int? i = 87;
int? j = null;
if (i.HasValue)
{
    int k = i.Value + j.GetValueOrDefault(42);
    Console.WriteLine(k);
}

int l = i.Value + (j ?? 42);    // if null return 42
```

- The `??` operator is an elegant shorthand

TEKNOLOGISK
INSTITUT

Characteristics of Nullable Types

- Make no mistake about it: Nullable types are **value types**!
- Only value types can be nullable!

- int? is actually defined as

```
Nullable<int> x = 42;
```

- This will become apparent when we discuss Generics later...

TEKNOLOGISK
INSTITUT

Null-conditional Operator

- C# 6.0 introduces a new null-conditional operator `?.`
  - Also known as the “Elvis operator” ☺
  - Works for reference types and nullable types

```
string s = null;

// ...

string tt = s?.ToUpper(); // Upper-case (or null if s == null)
```

- Right-hand side only evaluated if left-hand side is not null
  - Propagates null through expression

- Interacts brilliantly with the null-coalescing operator `??`

```
t = s?.ToUpper() ?? "No string";
```

TEKNOLOGISK INSTITUT

Interpolated Strings (string templates)

- Used to construct strings using expressions
  - $ keyword

```
int i = 10;
DateTime dt = new DateTime(2017, 1, 1);
string a = $"Test";                          // Test
string b = $"Test *{i}*";                    // Test *10*
string c = $"Test *{i:N2}*";                 // Test *10,00*
string d = $"Test {i} {dt:yyyy-MM-dd}";      // Test 10 2017-01-01
```

TEKNOLOGISK
INSTITUT

Defining Extension Methods

- *Extension methods* let you extend types with your own methods
  - Even if you don't have the source or the types are not yours

```
static class MyExtensions
{
    public static string ToMyTimestamp(this DateTime dt)
    {
        return dt.ToString("yyyy-MM-dd HH:mm:ss.fff");
    }
}
```

- Must be **static** and defined in a **static** class
- The first parameter contains `this` and determines the type being extended

- Extension methods can have any number of parameters

TEKNOLOGISK
INSTITUT

Invoking Extension Methods

- Extension methods can be invoked at the instance level

```
DateTime t = DateTime.Now;
Console.WriteLine(t.ToMyTimestamp());
```

- Alternatively, the method can be invoked statically

```
DateTime t = DateTime.Now;
Console.WriteLine(MyExtensions.ToMyTimestamp(t));
```

- Visual Studio has special IntelliSense for extension methods

TEKNOLOGISK
INSTITUT

Using Extension Methods

- The static class containing the extension methods must be in scope for the extension methods to be used
- Extension methods are indeed extending – not inheriting!
  - No access to private or protected members
  - All access is through the supplied parameter
- Can extend interfaces as well, but implementation must be provided

TEKNOLOGISK
INSTITUT

Creating Anonymous Types

- Combining implicitly typed variables with object initializer syntax provides an excellent shorthand for defining simple classes called *anonymous types*

```
var myEquipment = new
{
    Manufacturer = "Nintendo",
    Make = "Wii",
    Controllers = 4
};
Console.WriteLine("I have a {0} {1} with {2} controllers",
   myEquipment.Manufacturer,
   myEquipment.Make,
   myEquipment.Controllers);
```

- The compiler autogenerates an anonymous class for us to use
- This class inherits from `object`

- Members are read-only!

TEKNOLOGISK
INSTITUT

Equality of Anonymous Types

- Anonymous types come with their own overrides of `object` methods
  - `ToString()`
  - `Equals()`
  - `GetHashCode()`
- The `==` and `!=` operators are however not overloaded with `Equals()`!
  - The exact references are still compared

TEKNOLOGISK INSTITUT

Restrictions to Anonymous Types

- Anonymous types can be nested arbitrarily

```
var myFancyEquipment = new
{
    Manufacturer = "Microsoft",
    Make = "Xbox One",
    XboxLive = new { Name = "Komatoze", Membership = "Gold" }
};
```

- Some restrictions do apply to anonymous types
  - Type name is auto-generated and cannot be changed
  - Always derive directly from `object`
  - Fields and properties of anonymous types are always read-only
  - Anonymous types are implicitly sealed
  - No possibility of custom methods, operators, overrides, or events

TEKNOLOGISK
INSTITUT

Yield

- Used primarily to make iterations return values while iterating

```
public static IEnumerable<int> GetCollection1(int count)
{
    List<int> lst = new List<int>();
    for (int i = 0; i < count; i++)
    {
        lst.Add(rnd.Next(0, 100));
        Console.WriteLine("Index " + i);
    }
    return lst;
}
```

```
public static IEnumerable<int> GetCollection2(int count)
{
    for (int i = 0; i < count; i++)
    {
        int t = rnd.Next(0, 100);
        Console.WriteLine("Index " + i);
        yield return t;
    }
}
```

Yield

```
Console.WriteLine("1");
int i = 0;
foreach (var item in GetCollection1(5))
{
    if (i++ == 2)
        break;
    Console.WriteLine("Value " + item);
}
```

```
1
Index 0
Index 1
Index 2
Index 3
Index 4
Value 32
Value 60
2
Index 0
Value 29
Index 1
Value 57
Index 2
```

```
Console.WriteLine("2");
i = 0;
foreach (var item in GetCollection2(5))
{
    if (i++ == 2)
        break;
    Console.WriteLine("Value " + item);
}
```

Operator overload

- It's possible to overload all operators
  - Must be public and static
  - +, -, *, /, ==, !=, <, >

```
class Car
{
    public int Speed { get; set; }
    public string Color { get; set; }
    public string Make { get; set; }

    public static bool operator >(Car c1, Car c2)
    {
        return c1.Speed > c2.Speed;
    }

    public static bool operator <(Car c1, Car c2)
    {
        return c1.Speed < c2.Speed;
    }
}
```

```
Car c1 = new Car() { Speed = 100 };
Car c2 = new Car() { Speed = 150 };
Console.WriteLine(c1 > c2);     // false
Console.WriteLine(c1 < c2);     // true
```

TEKNOLOGISK
INSTITUT

Attributtes

- Associating declarative information with C# code
  - Decorate classes, methods, arguments etc. with meta data
  - Use reflection to retrieve information

```
[Serializable]
class Person {

    [Obsolete("Use Write()")]
    public void Print() { }


}
```

```
Person p = new Person();
System.Attribute[] attrs
    = System.Attribute.GetCustomAttributes(typeof(Person));
foreach (var item in attrs)
{
    Console.WriteLine(item.TypeId); // System.SerializableAttribute
}
```