

Module 13

"Advanced C# Language Features"



TEKNOLOGISK
INSTITUT

Agenda

- ▶ **Indexers**
- ▶ Extension Methods
- ▶ Anonymous Types
- ▶ Lab 13
- ▶ Discussion and Review

Defining Indexers

- ▶ You can create “array-like” indexing of your own classes using *indexers*

```
class Garage
{
    private List<Car> list;
    ...
    public Car this[ int index ]
    {
        get { return list[ index ]; }
        set { list[ index ] = value; }
    }
}
```

```
Garage garage = new Garage();
Console.WriteLine( garage[1] );
garage[1] = new Car("Goofy", 87);
foreach( Car car in garage )
{
    Console.WriteLine( car );
}
```

- ▶ This is basically the syntax of a special property named **this** but with square brackets used instead of parentheses



Indexing Objects Using Strings

- ▶ You can create indexers on your own types with any indexing type – not just integers!

```
public Car this[ string index ]  
{  
    get { return list.Find( c => c.PetName == index ); }  
    set {  
        int i = list.FindIndex( c => c.PetName == index );  
        if( i >= 0 ) { list[ i ] = value; }  
        else { list.Add( value ); }  
    }  
}
```

```
Garage garage = new Garage();  
Console.WriteLine( garage[ "Zippy" ] );  
garage[ "Goofy" ] = new Car( "Goofy", 87 );
```

- ▶ Note that indexers can be overloaded in the same manner as methods!



Variations on Indexers

- ▶ Indexers can be multi-dimensional

```
class GridWrapper : IEnumerable
{
    private int[ , ] grid = new int[ 3, 3 ]
    public int this[ int row, int column ]
    {
        get { return grid[ row, column ]; }
        set { grid[ row, column ] = value; }
    }
}
```

```
GridWrapper gw = ...;
gw[ 0, 0 ] = 87;

foreach( int i in gw )
{
    Console.WriteLine( i );
}
```

- ▶ Indexers can be members of interfaces

```
public interface IMyStringContainer<T>
{
    string this[ T index ] { get; set; }
}
```

- ▶ Indexers can be virtual and generic



Agenda

- ▶ Indexers
- ▶ **Extension Methods**
- ▶ Anonymous Types
- ▶ Lab 13
- ▶ Discussion and Review

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



Invoking Extension Methods

- ▶ Extension methods can be invoked at the instance level

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

- ▶ Alternatively, the method can be invoked statically

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

- ▶ Visual Studio has special IntelliSense for extension methods



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

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

- ▶ Can extend interfaces as well, but implementation must be provided



Agenda

- ▶ Indexers
- ▶ Extension Methods
- ▶ **Anonymous Types**
- ▶ Lab 13
- ▶ Discussion and Review

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!



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



Restrictions to Anonymous Types

- ▶ Anonymous types can be nested arbitrarily

```
var myFancyEquipment = new
{
    Manufacturer = "Microsoft",
    Make = "Xbox One",
    XboxLive = new { Name = "Komatoze",
                     Membership = MembershipType.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





Quiz: Advanced C# Language Features – Right or Wrong?

```
class Garage
{
    public Car this[ int i, int j ] { ... }
    ...
}
```

```
static class DateTimeExtensions
{
    public static string ToMyTimestamp( DateTime dt ) { ... }
}
```

```
static class PersonExtensions
{
    public static string GetName( this Person p, bool upperCase ) { ... }
}
```

```
Car myCar = new { Color = "Navy", Make = "Saab", CurrentSpeed = 55 };
```



Lab 13: Advanced C# Language Features



Discussion and Review

- ▶ Indexers
- ▶ Extension Methods
- ▶ Anonymous Types



WINCUBATE

Jesper Gulmann Henriksen

PhD, MCT, MCSD, MCPD

Phone : +45 22 12 36 31

Email : jgh@wincubate.net

WWW : <http://www.wincubate.net>

Hasselvangelen 243

8355 Solbjerg

Denmark