

# Module 01

## "Advanced Types and Methods"



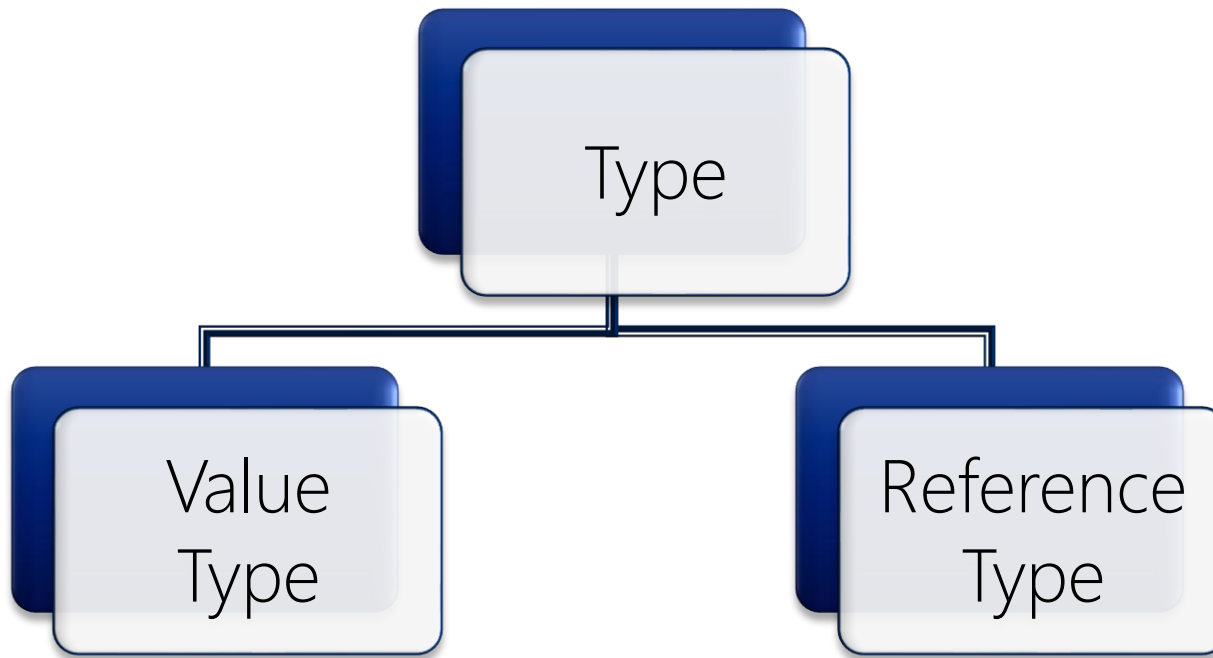
**TEKNOLOGISK**  
**INSTITUT**

# Agenda

- ▶ **Common Type System**
- ▶ Collections and Generics
- ▶ Iterators
- ▶ Anonymous Types
- ▶ Tuples and Other Types
- ▶ Methods
- ▶ Extension Methods
- ▶ Lab 1
- ▶ Discussion and Review

# Anatomy of the Common Type System

- ▶ Every variable has a type
- ▶ C# is statically typed



# Value Types vs. Reference Types

## Value Types

- ▶ Directly contain data
- ▶ Allocated on the stack
- ▶ Have to be initialized
- ▶ Each copy has its own data

## Reference Types

- ▶ Store references to data ("objects")
- ▶ Stored on the heap
- ▶ Has a default value of **null**
- ▶ Several references can refer to same data

# Examples of Types

## Value Types

- ▶ bool
- ▶ int
- ▶ float
- ▶ decimal
- ▶ **struct**
- ▶ enum
- ▶ DateTime
- ▶ ...

## Reference Types

- ▶ **class**
- ▶ string
- ▶ Array
- ▶ Exception
- ▶ ...

# Implicitly Typed Variables

- ▶ You can define local implicitly typed variables using the **var** keyword

```
var myInteger = 87;  
var myBoolean = true;  
var myString = "Hello, there...";
```

- ▶ The compiler infers the type of the local variable!
- ▶ Everything is still completely type-safe

```
var i = 87;  
i = 112;  
int j = i + 42;  
i = "Forbidden!";
```



- ▶ Must be assigned a value when declared

```
var myInteger;  
myInteger = 87;
```



# 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 k = i.Value + ( j ?? 42 );
```

- ▶ The ?? operator is an elegant shorthand



# Characteristics of Nullable Types

- ▶ Make no mistake about it:
  - Nullable types are **value types**!
- ▶ Only value types can be nullable!
- ▶ **int?** is actually generically defined as

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



# Recursive Types

- ▶ Classes are allowed to be recursive, i.e refer to instances of the same class
- ▶ A classic examples is the Linked List

```
public class Node
{
    public object Data { get; set; }
    public Node Next { get; set; }

    ...
}
```

- ▶ Note: Structs are **not** allowed to be recursive!



# Agenda

- ▶ Common Type System
- ▶ **Collections and Generics**
- ▶ Iterators
- ▶ Anonymous Types
- ▶ Tuples and Other Types
- ▶ Methods
- ▶ Extension Methods
- ▶ Lab 1
- ▶ Discussion and Review

# System.Collections.Stack

- Stack is a container with last-in, first-out behavior based on **object**

Member of <b>Stack</b>	Meaning
<b>Push()</b>	Adds an object to the top of the stack
<b>Pop()</b>	Removes the object at the top of the stack
<b>Peek()</b>	Returns the object at the top of the stack without removing it

```
Stack stack = new Stack();  
stack.Push( new Car( "Fred", 90 ) );  
stack.Push( new Car( "Mary", 100 ) );  
Car top = stack.Peek() as Car;  
Car removed = stack.Pop() as Car;  
foreach( Car c in stack )  
{  
    Console.WriteLine( c.PetName );  
}
```



# Annoying Problems

- ▶ You can insert anything into a **Stack**!

```
Stack stack = new Stack();  
stack.Push( new Car( "Fred", 90 ) );  
stack.Push( new Car( "Mary", 100 ) );  
stack.Push( "Hello, World" );  
stack.Push( 87 );
```



```
Car top = stack.Peek() as Car;  
Car removed = stack.Pop() as Car;
```



```
foreach( Car c in stack )  
{  
    Console.WriteLine( c.PetName );  
}
```



- ▶ The problem is that type-safety is missing



# Wouldn't It Be Nice If...

- ▶ ... we only needed to construct each type once?
- ▶ ... and it had no (un)boxing performance hit?

```
class Stack<T>
{
    public Stack { ... }
    public T Peek() { ... }
    public void Push( T t ) { ... }
    public T Pop() { ... }
    ...
}
```

- ▶ I.e. "generic" types!



# The Classes of the **System.Collections.Generic** Namespace

- ▶ Type-safe, reusable, and efficient collection classes

Class	Meaning
<b>List&lt;T&gt;</b>	Dynamically sized list of elements of type T
<b>Dictionary&lt;K,V&gt;</b>	Values of type V indexed by an element key of type K
<b>SortedDictionary&lt;K,V&gt;</b>	Values of type V indexed and sorted by keys of type K
<b>Queue&lt;T&gt;</b>	First-in, first-out queue of elements of type T
<b>Stack&lt;T&gt;</b>	Last-in, first-out queue of elements of type T
<b>HashSet&lt;T&gt;</b>	Set of elements of type T
<b>SortedSet&lt;T&gt;</b>	Sorted set of elements of type T

- ▶ These implement the generic interfaces on the previous slide
- ▶ Never use the non-generic collections!

# Using Generic Types

- ▶ Substitute T with a concrete type whenever it is used

```
List<int> list = new List<int>();  
list.Add( 42 );  
list.Add( 87 );  
list.Add( 112 );  
  
foreach( int i in list )  
{  
    Console.WriteLine( i );  
}
```

```
List<string> list = new  
List<string>();  
list.Add( "Hello" );  
list.Add( "World" );  
  
foreach( string s in list )  
{  
    Console.WriteLine( s );  
}
```



# Queue<T>

- ▶ **Queue<T>** is a type-safe container ensuring first-in, first-out behavior

Member of <b>Queue&lt;T&gt;</b>	Meaning
<b>Dequeue()</b>	Removes and returns the element at beginning of queue
<b>Enqueue()</b>	Adds an element to the end of queue
<b>Peek()</b>	Returns the element at the beginning

```
Queue<Car> queue = new Queue<Car>();
queue.Enqueue( new Car( "Fred", 90 ) );
queue.Enqueue( new Car( "Mary", 100 ) );
Car first = queue.Peek();
Car removed = queue.Dequeue();
foreach( Car c in queue )
{
    Console.WriteLine( c.PetName );
}
```





# Dictionary<K,V>

- ▶ **Dictionary<K,V>** is a container of values of type V indexed by an element key of type K

Member of <b>Dictionary&lt;K,V&gt;</b>	Meaning
<b>Add()</b>	Adds an key-value pair to the dictionary
<b>Remove()</b>	Removes the element with the specified key

- ▶ Iterate dictionaries by using **KeyValuePair<K,V>**

```
Dictionary<int, string> dict = new Dictionary<int, string>();  
dict.Add( 11, "Peter Graulund" );  
dict.Add( 7, "Stephan Petersen" );  
Console.WriteLine( "Number 11 is {0}", dict[ 11 ] );  
  
foreach( KeyValuePair<int, string> kv in dict )  
{  
    Console.WriteLine( "Player {0} is {1}", kv.Key, kv.Value );  
}
```

# HashSet<T>

- ▶ **HashSet<T>** is a set of values of type T

Member of <b>HashSet&lt;T&gt;</b>	Meaning
<b>Add()</b>	Adds an element to the set
<b>Remove()</b>	Removes the specified element in the set

- ▶ There is also a **SortedSet<T>**

- Needs **IComparer<T>**

```
HashSet<int> set = new HashSet<int>();  
set.Add( 42 );  
set.Add( 87 );  
set.Add( 42 );  
set.Remove( 42 );  
  
foreach( int i in set )  
{  
    Console.WriteLine( i );  
}
```



# Collection Initializers

- ▶ Collections can be conveniently initialized via *collection initializer syntax*

```
List<int> list = new List<int> { 42, 87, 112 };
```

```
List<string> list = new List<string> { "Hello", "World" };
```

```
SortedSet<int> set = new SortedSet<int> { 87, 42, 112, 176 };
```

- ▶ Note: Only works for those collection classes with an **Add()** method, i.e. not
  - Stack<T>
  - Queue<T>
  - LinkedList<T>
  - ...



# Agenda

- ▶ Common Type System
- ▶ Collections and Generics
- ▶ **Iterators**
- ▶ Anonymous Types
- ▶ Tuples and Other Types
- ▶ Methods
- ▶ Extension Methods
- ▶ Lab 1
- ▶ Discussion and Review

# The IEnumerable Interface

- ▶ The **IEnumerable** interface states that the items of a class can be enumerated

```
using System.Collections;

public interface IEnumerable
{
    IEnumerator GetEnumerator();
}
```

```
public interface IEnumerator
{
    bool MoveNext ();
    object Current { get; }
    void Reset ();
}
```

- ▶ The **IEnumerator** interface provides an enumerator mechanism for the class
- ▶ Both are built into the .NET Framework base classes in the **System.Collections** namespace
- ▶ Arrays and collection types implement **IEnumerable** out-of-the-box



# Implementing IEnumerable

- ▶ You can implement **IEnumerable** in your own types

```
public class Garage : IEnumerable
{
    private Car[] carArray = new Car[ 4 ];
    public Garage()
    {
        carArray[ 0 ] = new Car( "FeeFee", 200 );
        carArray[ 1 ] = new Car( "Clunker", 90 );
        carArray[ 2 ] = new Car( "Zippy", 30 );
        carArray[ 3 ] = new Car( "Fred", 30 );
    }

    public IEnumerator GetEnumerator() { ... }
}
```

```
Garage garage = new Garage();
foreach( Car c in garage )
{
    Console.WriteLine( c.PetName );
}
```



# Building Iterators with `yield`

- ▶ C# provides powerful mechanisms for creating *iterator methods*

```
public IEnumerator GetEnumerator()  
{  
    foreach( Car c in carArray )  
    {  
        yield  
    }  
}
```

```
public IEnumerator GetEnumerator()  
{  
    yield return carArray[ 0 ];  
    yield return carArray[ 1 ];  
    yield return  
    yield return  
}
```

```
public IEnumerator GetEnumerator()  
{  
    int i = 0;  
    while( true )  
    {  
        yield return carArray[ i++ ];  
        if( i == 4 ) { yield break; }  
    }  
}
```



# Named Iterators

- ▶ Multiple iterators can be built for a class with named iterators

```
public IEnumerable GetTheCars( bool returnReversed )
{
    if( returnReversed )
    {
        for( int i = carArray.Length; i != 0; i-- )
        { yield return carArray[i-1]; }
    }
    else
    {
        foreach( Car c in carArray ) { yield return c; }
    }
}
```

```
Garage garage = new Garage();
foreach( Car c in garage.GetTheCars( true ) )
{
    Console.WriteLine( c.PetName );
}
```





# Implementing `IEnumerable<T>`

- ▶ There are generic versions of `IEnumerable` and `IEnumerator`

```
public interface IEnumerable<out T> : IEnumerable
{
    IEnumerator<T> GetEnumerator();
}
```

```
public interface IEnumerator<T>
{
    bool MoveNext ();
    T Current { get; }
    void Reset ();
}
```

- ▶ *Note: Slight trick involved when implementing `IEnumerable<T>`*
- ▶ We will return to the "**out**" in Module 2

# Agenda

- ▶ Common Type System
- ▶ Collections and Generics
- ▶ Iterators
- ▶ **Anonymous Types**
- ▶ Tuples and Other Types
- ▶ Methods
- ▶ Extension Methods
- ▶ Lab 1
- ▶ 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 360",
    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



# Agenda

- ▶ Common Type System
- ▶ Collections and Generics
- ▶ Iterators
- ▶ Anonymous Types
- ▶ **Tuples and Other Types**
- ▶ Methods
- ▶ Extension Methods
- ▶ Lab 1
- ▶ Discussion and Review

# Tuples

- ▶ Tuples are immutable data values supporting 1 to 8 data items of any type named
  - **Item1, Item2, ...**
- ▶ Carry no semantic meaning

```
Tuple<int, int> point = Tuple.Create( 1, 3 );  
Console.WriteLine( "The point is {0}", point );
```

```
Tuple<string, bool, int> person =  
    Tuple.Create( "Anders Hejlsberg", true, 220 );
```

- ▶ Provides overridden methods
  - **ToString()**
  - **Equals()**
- ▶ Implements
  - **Comparable** (explicitly)







# Agenda

- ▶ Common Type System
- ▶ Collections and Generics
- ▶ Iterators
- ▶ Anonymous Types
- ▶ Tuples and Other Types
- ▶ **Methods**
- ▶ Extension Methods
- ▶ Lab 1
- ▶ Discussion and Review

# The Syntax of a Method

- ▶ The syntax of methods are

```
ReturnValue MethodName( arguments ) { MethodBody }
```

- ▶ All methods must exist inside of a class definition – no “global” methods!

```
class Program
{
    static void DoStuff( )
    {
        Console.WriteLine( 87 );
    }
}
```

```
class Calculator
{
    public int Add(int x, int y)
    {
        return x + y;
    }
}
```

# Implicit Typing in Methods

- ▶ The **var** keyword cannot be used as parameters or return value in methods

```
public var M( var x, var y )  
{  
    ...  
}
```

- ▶ But can be used locally inside the method body

```
int GetSomeInt()  
{  
    var ret = 87; ✓  
    return ret;  
}
```

# Passing Parameters by Value

- ▶ Define *formal* parameters within parentheses in method
  - Supply type and name for each parameter

```
static void Twice( int x )  
{  
    x = 2 * x;  
}
```

- ▶ Invoke method by supplying *actual* parameters in parentheses
  - The formal and actual parameter types and count must be compatible

```
int i = 42;  
Twice( i );  
Console.WriteLine( i );
```

- ▶ Parameter values are copied from actual to formal
- ▶ Changes made inside method has no effect outside method!



# The `ref` Modifier

- ▶ Reference parameters are references to memory locations, i.e. aliases for variables
- ▶ Use the **`ref`** modifier to pass variables by reference

```
static void Twice( ref int x )  
{  
    x = 2 * x;  
}
```

```
int i = 42;  
Twice( ref i );  
Console.WriteLine( i );
```

- ▶ Also use the **`ref`** keyword when invoking the method
- ▶ Parameter values are referred (or aliased)
- ▶ Changes made inside method has indeed effect outside method!
- ▶ Variable must be assigned before call



# The out Modifier

- ▶ Passing by reference consists of both “inputting” and “outputting”
- ▶ Use the **out** modifier when only outputting value

```
static void FillWithNumber( out int x )  
{  
    x = 87;  
}
```

```
int i;  
FillWithNumber( out i );  
Console.WriteLine( i );
```

- ▶ Also use the **out** keyword when invoking the method
- ▶ Parameter values are output
- ▶ Changes made inside method has indeed effect outside method!
- ▶ Variable does not have to be assigned before call



# The params Modifier

- ▶ Passing parameter lists of varying length by using the **params** modifier

```
static int Sum( params int[] values )  
{  
    int total = 0;  
    foreach( int i in values )  
    {  
        total += i;  
    }  
    return total;  
}
```

```
Console.WriteLine( Sum( 42, 87 ) );
```

- ▶ Actual parameters are then passed into the method by value as an array
- ▶ Only one **params** per method



# Optional Parameters

- ▶ Methods can have optional parameters by specifying their default values

```
static void M( int x, int y = 87, bool z = false )  
{  
    ...  
}
```

M(1, 2, true);	✓
M(1, 2); // Equivalent to M(1, 2, false)	✓
M(1); // Equivalent to M(1, 87, false)	✓
M(); // Illegal! x is required!	✗

- ▶ Optional parameters can be omitted when invoking the method
- ▶ Note: Optional parameters must appear last in parameter list
- ▶ Default values for optional parameters must be known at compile time!

```
static void N( bool b, DateTime dt = DateTime.Now )  
{  
    ...  
}
```





# Named Parameters

- ▶ Can pass parameter values using their *names* (as opposed to their *position*)

```
M(1, z: true); ✓ // z is passed by name  
M(x: 1, z: true); ✓ // x and z are both passed by name  
M(z: true, x: 1); ✓ // z and x are both passed by name (equivalent!)  
M(z: true, 1 ) ✗
```

- ▶ Note: Positional parameters must always appear before any named parameters when invoking methods!
- ▶ Named and optional parameters mix perfectly
- ▶ Syntax look horrible, but what is the alternative...? ☺



# Recursive Methods

- ▶ Methods can call itself either directly or indirectly.
- ▶ Such methods are said to be *recursive*

```
static int Factorial( int n )  
{  
    if( n <= 1 )  
    {  
        return 1;  
    }  
  
    return n * Factorial( n - 1 );  
}
```

```
Console.WriteLine( Factorial( 10 ) );
```

- ▶ Perfect for solving inductively defined problems
- ▶ Must have terminating base clause
- ▶ Use with care!



# Generic Methods

- ▶ You can define methods operating on generic types

```
void Swap<T>( ref T a, ref T b )  
{  
    T temp = a;  
    a = b;  
    b = temp;  
}  
  
int i = 42;  
int j = 87;  
Swap<int>( ref i, ref j );  
  
string s = "Hello";  
string t = "World";  
Swap<string>( ref s, ref t );
```

- ▶ Such methods cannot be defined inside generic classes or structs!
- ▶ T is "free" to match any type
  - Use **typeof(T)** to retrieve instantiated type
- ▶ The C# compiler will try to infer the generic types when omitted



# Caller Info Attributes

- ▶ C# 5.0 introduced three types of caller info attributes
  - [CallerMemberName]
  - [CallerFilePath]
  - [CallerLineNumber]

```
static void Log(  
    [CallerMemberName] string callerName = null,  
    [CallerFilePath] string callerFilePath = null,  
    [CallerLineNumber] int callerLine = -1 )  
{  
    ...  
}
```

- ▶ Applicable to default parameters
  - Compiler replaces values at compilation time
- ▶ In **System.Runtime.CompilerServices**



# Agenda

- ▶ Common Type System
- ▶ Collections and Generics
- ▶ Iterators
- ▶ Anonymous Types
- ▶ Tuples and Other Types
- ▶ Methods
- ▶ **Extension Methods**
- ▶ Lab 1
- ▶ 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





# Lab 1: Creating Advanced Types and Methods

- ▶ Labs 1.1 – 1.8



# Discussion and Review

- ▶ Common Type System
- ▶ Collections and Generics
- ▶ Iterators
- ▶ Anonymous Types
- ▶ Tuples and Other Types
- ▶ Methods
- ▶ Extension Methods



WINCUBATE

***Jesper Gulmann Henriksen***

PhD, MCT, MCSD, MCPD

Phone : +45 22 12 36 31

Email : [jgh@wincubate.net](mailto:jgh@wincubate.net)

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

Hasselvangelen 243

8355 Solbjerg

Denmark