Oliver Sturm

C# 2012 - Best Practices

Oliver Sturm
oliver@oliversturm.com

http://www.oliversturm.com

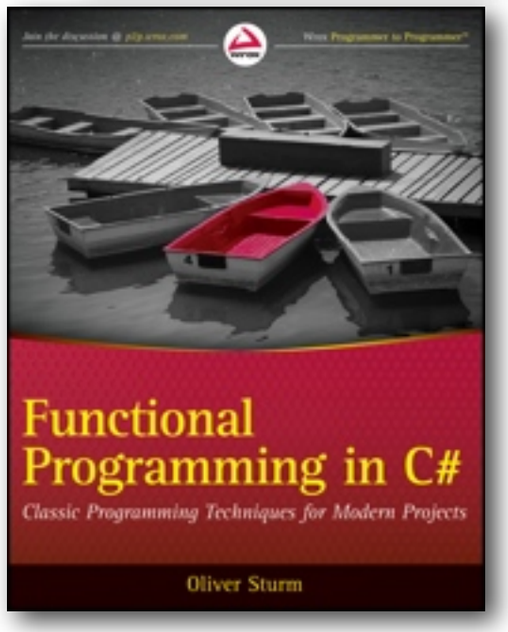
thinktecture Associate

Oliver Sturm (@olivers)

- **Consultant and Trainer, Author**
- **Associate Consultant at thinktecture**

- **.NET Application System Architecture**
  - User Interfaces
  - Data Handling / Data Access Architectures
  - Programming Languages
  - DevExpress Component/Framework Products

- **Microsoft MVP for C#**
- **INETA Europe Speaker**



- **Services: http://www.oliversturm.com**
- **Blog: http://www.sturmnet.org/blog**

- **oliver@oliversturm.com**

Agenda

- C#
- Its features
  - especially the ones you really should know, but might have missed somehow
- Why they are good/useful/plain cool or not
  - my take, your take
- And of course how they work.

(Some) C# Features in 1.0

Implementation inheritance

Classes

Interfaces

Exception handling

Delegates

using()

Events

Loop constructs

Explicit/implicit interface implementation

Boxing/unboxing

Comparison constructs

C# Features > 1.0

2.0	3.0	4.0	5.0
Nullable Types	var	Optional Parameters	Async/Await
Null-coalescing operator	Extension Methods	Named Parameters	
Access modifiers for properties	Auto-implemented properties	Co-/Contravariance (in/out)	
Aliases for assembly refs	Initializers	“dynamic”	
Iterators / yield return…	Anonymous Types		
Generics	Partial Methods		
Static Classes	Lambda Expressions		
Partial Classes	Expression Trees		
Anonymous Methods	Query Expressions		
Co-/Contravariance			
Closures			

C# Features > 1.0

2.0	3.0	4.0	5.0
Nullable Types	var	Optional Parameters	Async/Await
Null-coalescing operator	Extension Methods	Named Parameters	
Access modifiers for properties	Auto-implemented properties	Co-/Contravariance (in/out)	
Aliases for assembly refs	Initializers	"dynamic"	
Iterators / yield return…	Anonymous Types		
Generics	Partial Methods		
Static Classes	Lambda Expressions		
Partial Classes	Expression Trees		
Anonymous Methods	Query Expressions		
Co-/Contravariance			
Closures			

The important stuff

In no particular order:

- Iterators
- Closures
- Expression Trees
- dynamic
- async/await
- Co-/Contravariance
-

C# 1.0: Delegates

```
public delegate int TakeIntReturnIntDelegate(int p);

static int Square(int x) {
  return x * x;
}

static void DoSomethingWithDelegate(
  TakeIntReturnIntDelegate del) {
  Console.WriteLine(del(5) + del(8));
}
```

C# 1.0: Exception handling

```
try {
  throw new Exception("Something went wrong");
}
catch (Exception e) {
  Console.WriteLine("Error: {0}", e.Message);
  throw;
}
finally {
  Console.WriteLine("Cleaning up");
}
```

Code smells:

- no exception parameter with “catch”
- wrong “throw”
- no “throw”
- standard exceptions for everything

C# 1.0: Using

```
using (MyType t = new MyType( )) {
  t.DoSomething( );
}

...

class MyType : IDisposable {
  public void DoSomething( ) {
    Console.WriteLine("Doing something");
  }
  void IDisposable.Dispose( ) {
    Console.WriteLine("Disposing myself");
  }
}
```

C# 1.0: Explicit/implicit interface implementation

```
public class ExplicitImplementation : IMyInterface {
  int IMyInterface.GetVal(string p) {
    return 42;
  }

  DateTime IMyInterface.Date {
    get { return DateTime.Today; }
  }
}
```

C# 2.0: Nullable Types

```
int? i = null;
int? j = 5;
```

- General idea: unify special case
- Occasional use, depending on code being written
- DBNull covers related case

C# 2.0: Null-coalescing Operator

```
string hello = "Hello ";
string name = null;
string unknown = "unknown user";

string message = hello + (name ?? unknown);
Console.WriteLine(message);
```

- Shortens inline null checks
- Some common cases not covered:

```
string msg = “Hello” + (person != null ? person.Name :  “unknown”);
```

- Not compatible with people who hate ternary expressions :-)

C# 2.0: Property access modifiers

```
private string textVal;
public string TextVal {
  get { return textVal; }
  protected set {
    textVal = value;
  }
}
```

- Configure access to property getter/setter separately
- Sounds like it makes sense, but I can't remember having used it, ever

C# 2.0: Aliases for assembly refs

- Referencing several assemblies with identical FQNs becomes possible

  Compiler:
  `/r:MyAlias=myassembly.dll`

  Code:
  `extern alias MyAlias;`

- Important feature to have for conflict situations. Never used it myself though.

C# 2.0: Generics

```
new Dictionary<ITuple<Type, Type>,
  Dictionary<ITuple<Func<P, R>,
    Func<Func<P, R>, Func<P, R>>>, Func<P, R>>>( )
```

- Type system abstraction for "types that work with other types"
- CLR based implementation
- Combines type safety with flexibility
- Can have performance advantages for boxed types
- Constraints available
- Limitations around operations performed on the "unknown" types

C# 2.0: Anonymous Methods

```
TakeIntReturnIntDelegate method1 =
  delegate(int v) {
    return v + 100;
  };

DoSomethingWithDelegate(method1);

DoSomethingWithDelegate(delegate(int x) { return x / 2; });
```

- Inline, unnamed methods
- Simplify delegate implementation
- "Nested method" functionality for C#

C# 2.0: Iterators

```
public static IEnumerable<int> EndlessListFunction( ) {
  int val = 0;
  while (true)
    yield return val++;
}
```

- Simplify implementation of IEnumerable/ IEnumerator
- Important building block for later LINQ APIs etc.

C# 2.0: Static Classes

```
public static class MyClass {
  public static void DoIt( ) {
    Console.WriteLine("I'm doing it!");
  }
}
```

- Entire classes can be marked “static”
- Can only contain “static” members
- Marking a class “static” is a flag of intention

C# 2.0: Partial Classes

```
public partial class DatabaseMapping {
   ...
```

- There can be several "parts" that declare one class together
- Useful in conjunction with code generation
- VS designer takes advantage
- Not just two parts allowed
- Conflicts are **not** allowed
- Careful when deriving - one base class only!

C# 2.0: Co-/Contravariance for delegates

```
delegate void AcceptBirdDelegate(Bird bird);
delegate Animal GetAnimalDelegate( );
```

- Replacing types from signature declarations with related types
- Contravariance for parameters, Covariance for return types

C# 2.0: Closures

```
int baseVal = 10;

Func<int, int> add = delegate(int val) {
  return baseVal + val;
};
```

- Capturing variables from outer scopes for later use
- Basis: clever code generation

C# 3.0: var

```
var i = 5;
var s = "Hello";
var d = 1.5;

var numbers = new int[] { 1, 2, 3 };
var orders = new Dictionary<string, Order>( );
```

- Compiler infers types from values assigned to variables
- C# is statically typed just like always!
- Some sort of code discipline recommended

C# 3.0: Extension Methods

```
public static string Concatenate(
  this IEnumerable<string> strings, string separator) {
  ...
}
```

- Make utility methods easier to find
- Chain up calls in a more natural order

C# 3.0: Auto-implemented properties

```
public class Person {
  public string Name { get; set; }
  public int Age { get; private set; }
}
```

- Simplified syntax for standard property implementation pattern with backing store variable
- Watch out for future maintenance issues!

C# 3.0: Initializers

```
Person person = new Person {
  Name = "Willy",
  Age = 42
};
```

- Setting property values on creation without special constructors

C# 3.0: Anonymous Types

```
var person = new {
  Name = "Willy",
  Age = 33
};
```

- Constructing types on the fly
- Problem: very restricted in their use

C# 3.0: Partial Methods

```
partial void PartialMethod( ) {
  Console.WriteLine("Partial Method executed");
}
```

- Extension of the Partial Class idea
- Mainly for implementors of code generation
- Highly efficient alternative to events or complex OO structures

C# 3.0: Lambda Expressions

```
x => x * x * x
```

- Simplified syntax for anonymous methods, with lots of variations
- Types inferred in many cases
- Expression-body expressions compatible with Expression Trees

C# 3.0: Expression Trees

```
Expression<Func<double, double>> expression = e => e * 2 + 1;
```

- "Code is Data" for C#
- Analysis and construction of code expressions at runtime
- Basis of "translation" functionality in LINQ

C# 3.0: Query Expressions

```
var query =
  from c in customers
  where c.City == "Madrid"
  select c.CompanyName;
```

- C# syntax on top of LINQ mechanisms
- Nicer looking in-code queries
- Syntax somewhat restrictive

C# 4.0: Optional Parameters

```
static void DoSomething(int x, int y = 20) {
  ...
}
```

- Default values for parameters
- Important for automation scenarios
- Potential for confusion when combined with overloading, inheritance, named parameters, ...

C# 4.0: Named Parameters

```
DoSomething(y: 100, x: 13);
```

- Pass parameters by name
- The order doesn't matter
- You can skip parameters
- Very useful for automation scenarios
- Good feature to have, but it's a code smell if your APIs are only usable with its help

C# 4.0: Variance (in/out)

```
var birdcage = new Cage<Bird>() as ICage<Bird>;
var animalcage = (ICage<Animal>) birdcage;
var eagle = new Eagle( );
((IMyComparer<Eagle>) birdcage).EqualTo(eagle);
```

- As usual, variance makes things work "like they should"
- "in" and "out" keywords allowed on interfaces and delegates
- Many standard .NET interfaces decorated with these keywords in .NET 4.0

C# 4.0: dynamic

```
dynamic i = 42;
dynamic s = "Hi there";

i.DoSomethingImpossible();
```

- Keyword “dynamic” results in code generation to interface with DLR
- Simplifies work with dynamic languages, COM Interop, etc

C# 5.0: async/await

```
var atomData = await webclient.DownloadStringTaskAsync(url);
```

- Implements "TAP", "task based asynchronous pattern"
- Keywords await and async
- await means "run this, queue remaining code as a continuation, then return to caller" -
  NOT: "wait here"
- async means "I'm going to use await" -
  NOT: this method runs asynchronously

Thank you

Please feel free to contact me about the content anytime.

oliver@oliversturm.com