Advanced C#

C# 3.0 features

InfoSupport

Agenda

1. General
    1. Partial Methods
    2. Initializers
    3. Anonymous Types
    4. Auto-Implemented Properties
    5. Extension Methods
    6. Lambda Expressions

General – Partial Methods

```
// tool-generated code
partial class Person
{
    public string Name {…}
    public string Address {…}

    private string _label;
    public string Label
    {
        get
        {
            MakeLabel();
            return _label;
        }
    }

    partial void MakeLabel();
}
```

```
// custom user code
partial class Person
{
    partial void MakeLabel()
    {
        _label =
          String.Format(
          "{0}\n\t{1}",
          Name, Address);
    }
}
```

General – Partial Methods

Why?

- To allow interaction between generated code and custom code in partial classes
- No need for inheritance and abstract methods or modifying the generated code
- The generated code can contain a method call to a partial method without having an implementation: the call will be ignored
- Partial method acts as a ‘light-weight event’

General – Partial Methods

Rules:

- Must be part of a *partial* class
- Must return *void*
- Cannot use *out* parameters
- Is always *private*
- May only have one implementing declaration

Agenda

1. General
    1. Partial Methods
    2. Initializers
    3. Anonymous Types
    4. Auto-Implemented Properties
    5. Extension Methods
    6. Lambda Expressions

General – Initializers

Example…

```
Person person =
        new Person{
                Name="Fred Adams",
                Address="15 Portobello Road, London"
        };

Person person =
        new Person("Fred Adams"){
                Address="15 Portobello Road, London"
        };
```

General – Initializers

… compiles to:

```
Person person =
        new Person();
person.Name = "Fred Adams";
person.Address = "15 Portobello Road, London";


Person person =
        new Person("Fred Adams");
person.Address = "15 Portobello Road, London";
```

General – Initializers

Why?

- To allow quick object initialization without having to write a lot of constructors
- Less coding

General – Initializers

Rules:

- The class must have an accessible (default) constructor
- Only properties and fields that are accessible and are not read-only, are available in the initializer

General – Collection Initializers

Example…

```
List<int> digits =

        new List<int> { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

General – Collection Initializers

… compiles to:

```
List<int> digits = new List<int>();

digits.Add(0);

digits.Add(1);

digits.Add(2);

digits.Add(3);

digits.Add(4);

digits.Add(5);

digits.Add(6);

digits.Add(7);

digits.Add(8);

digits.Add(9);
```

General – Collection Initializers

Rules:

- The collection class must have an Add method
- The collection class must implement IEnumerable

General – Dictionary Initializers

- Example

```
Dictionary<string, string> myDic = new
Dictionary<string, string>()
{
    ["key1"] = "value1",
    ["key2"] = "value2"
};
```

- Rules
  - It can be used on any collection supporting an indexer
  - No need to support Add

Agenda

1. General
    1. Partial Methods
    2. Initializers
    3. Anonymous Types
    4. Auto-Implemented Properties
    5. Extension Methods
    6. Lambda Expressions

General – Anonymous Types

Example

```
var b1 = new { Name = "John", Age = 40 };

Console.WriteLine("{0}: {1}", b1.Name, b1.Age);
```

```
Person person =
        new Person { Name = "Paul", Address = "Penny Lane" };
int age = 40;

var b2 = new {
                Name = "John",  // member assignment
                person.Address, // member access
                age             // simple name
             };

Console.WriteLine("name={0}, address={1}, age={2}",
                        b2.Name, b2.Address, b2.age);
```

General – Anonymous Types

Why?

- Without explicitly defining a type, the compiler creates a type based on the code that constructs an object
- Less coding
- Support for LINQ (later)

General – Anonymous Types

Rules (1):

- Only public, read-only, managed properties
- The names of the properties can be given explicitly or implicitly
- In code the type is always *var*
- The inferred anonymous class inherits directly from *object*
- Variables of type *var* cannot leave method scope except by casting to *object*

General – Anonymous Types

Rules (2):

- Two anonymous classes with exactly the same properties and order of the properties share their types
- Two objects of the same anonymous type are equal when all their properties are equal
  - The *Equals* method on the class will call *Equals* on the properties
  - Similarly the *GetHashCode* method on the class will use the *GetHashCode* method of the properties to generate a hash code

General – Anonymous Types (var)

More on var:

- Var does not mean *variant, object* or being late-bound
- The compiler will determine the actual type at compile time
- Can only be used for:
  1. local variables,
  2. *for*-loop initialization,
  3. *foreach*-loop initialization and in
  4. *using* statements.
- Be aware that using *var* decreases readability of the source code

Agenda

1. General
    1. Partial Methods
    2. Initializers
    3. Anonymous Types
    4. Auto-Implemented Properties
    5. Extension Methods
    6. Lambda Expressions

Auto-Implemented Properties

Example

```
partial class Person
{
    public string Name { get; set; }
    public string Address { get; set; }
    public DateTime Birthday { get; private set; }
}
```

Auto-Implemented Properties

Why?

- Instead of public fields; to prevent breaking the interface (on IL level) when changing from fields to properties
- Less coding

When not?

- When the name of the backing field needs to be specified (Serialization)
- When attributes are needed on the backing field
- When code has to be added to the *get* and/or *set*

Auto-Implemented Properties

Rules:

- Read-only and Write-only properties are not allowed
- Definite Assignment of *struct* types that have auto-implemented properties is only possible by having all user-defined constructors call the default constructor
  - Only then the properties will be Definitely Assigned and thus the *struct* too
  - The compiler will generate an error when omitted

Agenda

1. General
    1. Partial Methods
    2. Initializers
    3. Anonymous Types
    4. Auto-Implemented Properties
    5. Extension Methods
    6. Lambda Expressions

General – Extension Methods

Example

```
static class DoubleExtensions
{
        public static double Sin(this double number)
        {
                return Math.Sin(number);
        }
}
```

```
double p = 3.1415;

Console.WriteLine( p.Sin() );

// compiles to:
Console.WriteLine( DoubleExtensions.Sin(p) );
```

General – Extension Methods

Why?

- Adding logic to a class when:
  - The class' source code is not available for modification
  - The class cannot be inherited from
- No need to recompile the existing class

General – Extension Methods

Rules (1):

- Extension methods are always *static* methods on a *static* non-generic class
- The first parameter is of the type of the class that is being extended and is prefixed with *this*
- The extension method will only be available when the namespace of the extension class is in scope (using)

General – Extension Methods

Rules (2):

- When a method on the class exists (or is added later) with exactly the same signature as the extension, the extension method will not be called
- The extension method can not access private or protected members of the class that is being extended
- Extension methods are in effect static methods, so:
  - No late binding; extension methods are called based on compile-time type of an expression

General – Extension Methods

Powerful when used with interfaces!!

```
static class ICloneableExtensions
{
        public static object DeepClone(this ICloneable item)
        {
                return ... // something with reflection
        }
}
```

```
string copy = “Hello World”.DeepClone();


XmlDocument doc = …

XmlDocument docCopy = doc.DeepClone();
```

Agenda

1. General
    1. Partial Methods
    2. Initializers
    3. Anonymous Types
    4. Auto-Implemented Properties
    5. Extension Methods
    6. Lambda Expressions

General – Lambda Expressions

Example

```
List<Person> band =
        new List<Person> {
                new Person{ Name="John"},
                new Person{ Name="Paul"},
                new Person{ Name="George"},
                new Person{ Name="Ringo"}
        };


// C# 2.0: Anonymous Method:
IEnumerable<Person> orderedBand = band.OrderBy(
        delegate(Person person)
        {
                return person.Name.Length;
        }
);
```

General – Lambda Expressions

Example

```
List<Person> band =
        new List<Person> {
                new Person{ Name="John"},
                new Person{ Name="Paul"},
                new Person{ Name="George"},
                new Person{ Name="Ringo"}
        };


// C# 3.0: Lambda Expression:
IEnumerable<Person> orderedBand = band.OrderBy(
        person => person.Name.Length
);
```

General – Lambda Expressions

Why?

- Less coding
- More readable LINQ queries (later)

General – Lambda Expressions

Rules (1):

- Use lambda expressions to create delegates (and expression tree types, later)
- Syntax: `(input parameters) => expression`
- The expression can be a block statement
- The parameter types can be omitted when the compiler can infer the parameter types
  - Based on the body of the lambda expression and/or the expected delegate the compiler infers the types of the parameters and the resulting type

General – Lambda Expressions

Rules (2):

- A lambda expression can refer to variables that are in the outer scope and will prevent the outer variables to be garbage collected
- A lambda expression can not use a *ref* or *out* parameter of the enclosing method
- A *return* statement in the lambda expression will exit the lambda expression; it will not exit the enclosing method

Review

1. General
    1. Partial Methods
    2. Initializers
    3. Anonymous Types
    4. Auto-Implemented Properties
    5. Extension Methods
    6. Lambda Expressions

Questions & Answers

General

- Labs