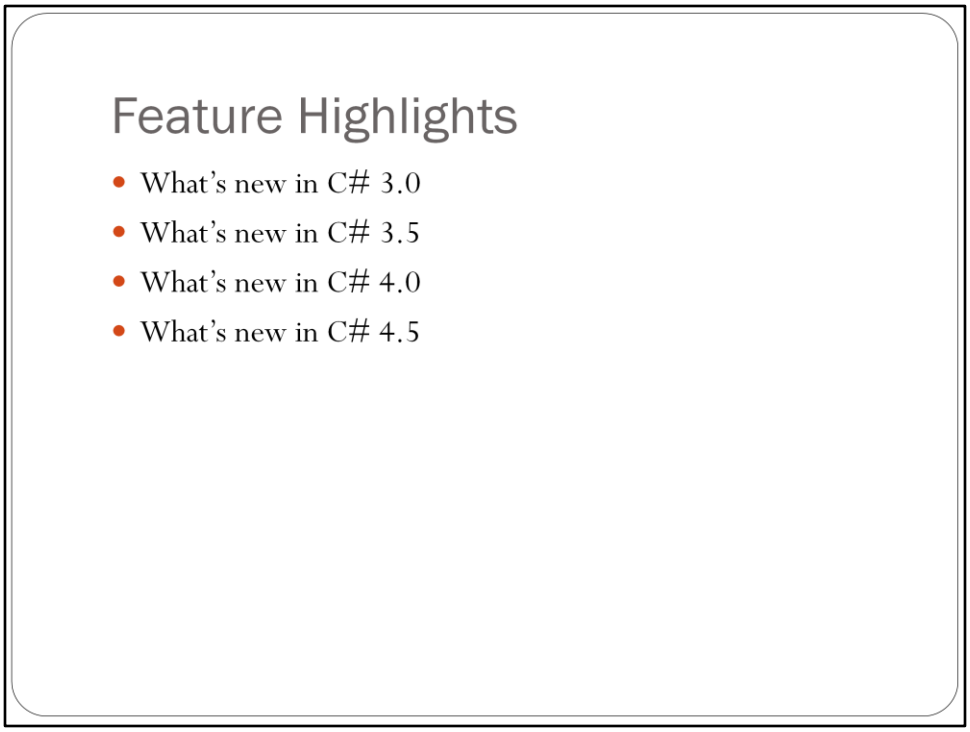
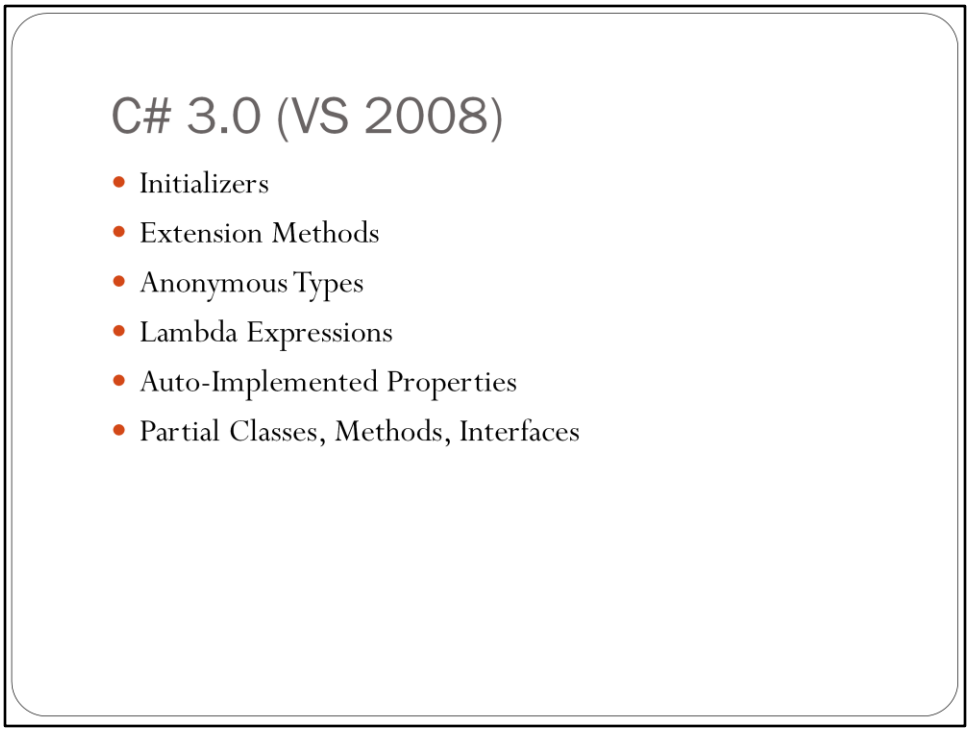
Introducing new C# features in the last (few) versions of C#. I went back as far as the Language Integrated Query (LINQ) feature, which I believe is a very significant aggregate feature set. I decided not to go back as far as generics.

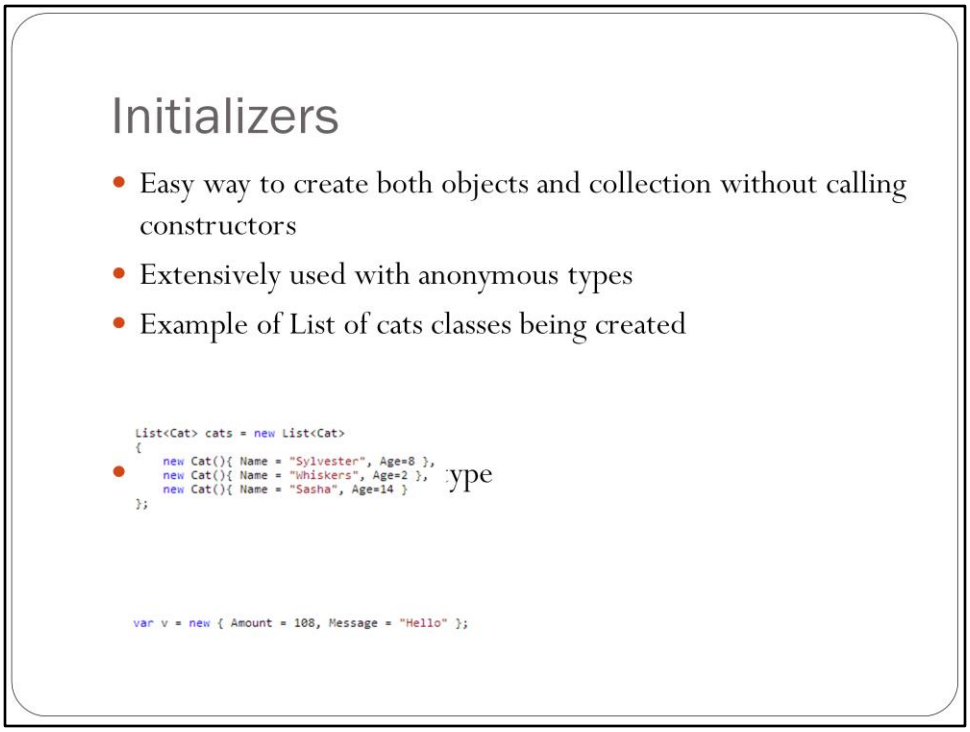


What's New Visual C# 2.0 : http://msdn.microsoft.com/en-US/library/7cz8t42e(v=vs.80).aspx
What's New in Visual C# 2008: http://msdn.microsoft.com/en-us/library/bb383815(v=vs.90).aspx
What's New in Visual C# 2010: http://msdn.microsoft.com/en-us/library/bb383815(v=vs.100).aspx
What's New in Visual C# 2012: http://msdn.microsoft.com/en-us/library/hh156499.aspx

These are the versions of both .NET and C# that I'm going to cover in this presentation. In the next few slides you will see highlights of new C# features introduced in each of these versions that I think are significant. This is not an exhaustive list because it would include too much information to cover in 20 minutes.



All the features you see here listed are nice features in and of themselves, but it's important to point out that they all make up the introduction of a major aggregate feature called LINQ that is introduced in the next version C# 3.5.



Initializers are a compact way to create both objects and collections without their specific constructors, only default constructors. It's also used a lot for anonymous types since they don't have constructors anyway. The example shown is using both object initializers and collection initializers. It creates a new list of three cats instead of calling the Add method of the list API to add three cats. Initializers are only used during the creation of the object. Each new cat is created by specifying the name and age. You can use any combinations of the properties to set in the class if you wish.

The anonymous type create a anonymous type "v" with properties Amount and Message and they are set with the values 108 and Hello respectively.

Extension Methods

- A way to extend an existing class's behavior without inheritance
- Adds methods by using a static method, but can be invoked by using instance method syntax
- Very common in LINQ
- Must be in the same namespace for methods to show up
- First parameter of the static method must be the type that you are extending specified by the *this* modifier
- Example:

```
namespace ExtensionMethods
{
    public static class MyExtensions
    {
        public static int WordCount(this String str)
        {
            return str.Split(new char[] { ' ', '.', '?' }, StringSplitOptions.RemoveEmptyEntries).Length;
        }
    }
}
```

Extension methods are used a lot in LINQ to add query methods to existing classes, specifically collection classes. Some examples of these methods are *Where, Find, OrderBy, etc.* Extension methods are an easy way to add functionality to existing classes without having to inherit from them. Essentially you're decorating an existing class with additional features. All you have to do is create a static method where the first parameter is the type you're extending such as String (see example) and proceed it with the *this* modifier. You then, in that method, can use the type's information and do what you need to do. The example adds a new method called *WordCount* to the String class. The only requirement is to make sure that the extension method is used in the same namespace where you want to use it. It will then show up as a method whenever you're using instances of the String class.

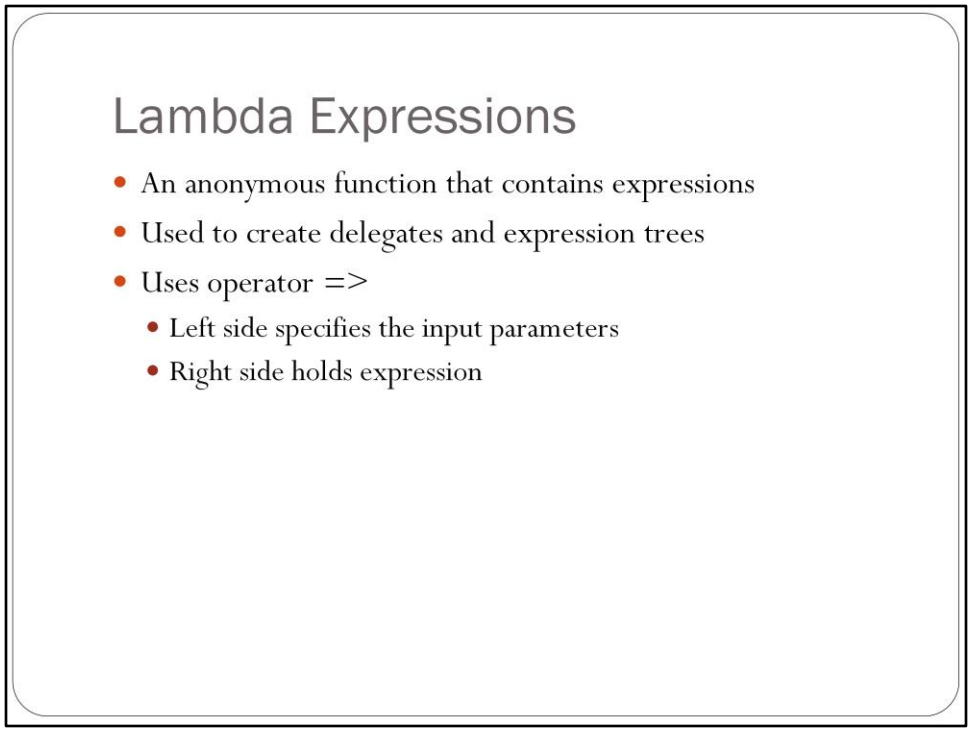
Anonymous Types

- "On the fly" types
- Must be typed with keyword *var*
- Structured types. Types with properties but not methods
- Example:
- Created by using new operator with the object initializer syntax

```
var v = new { Amount = 108, Message = "Hello" };
```

- Reference type inheriting directly from *Object*

Anonymous types are used throughout the language for different reasons. It's mostly used by LINQ. It's a way to save a lot of code by not having to create a concrete type when you only need a type to temporarily by used to hold a few properties.



Lambdas are another way to write delegates, sometimes known as function pointers. It's a much more compact way to write these and can be performed inline. They are extremely popular in LINQ where you can specify a Lambda expression to specify a predicate to query a collection or even a database. The syntax looks more declarative and/or functional than imperative.

Auto-Implemented Properties

- Provides quicker and easier way to implement properties
- No need to have private variables with wrappers

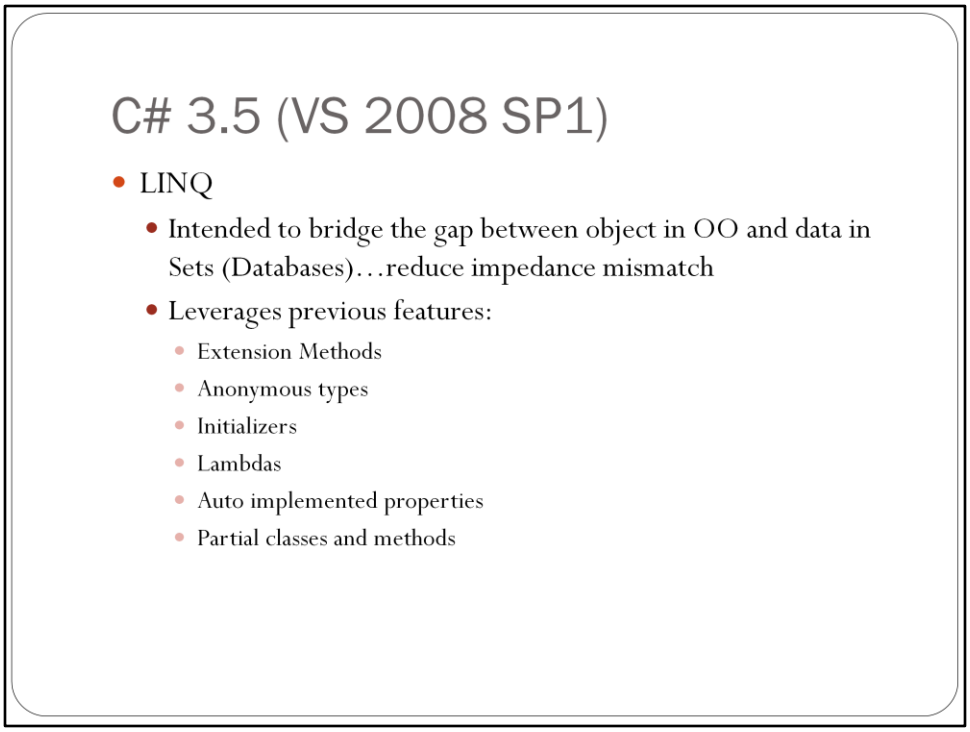
```
// This class is mutable. Its data can be modified from // outside the class. class Customer
{
    // Auto-Impl Properties for trivial get and set public double TotalPurchases { get; set; }
    public string Name { get; set; }
    public int CustomerID { get; set; }

    // Constructor public Customer(double purchases, string name, int ID)
    {
        TotalPurchases = purchases;
        Name = name;
        CustomerID = ID;
    }
    // Methods public string GetContactInfo() {return "ContactInfo";}
    public string GetTransactionHistory() {return "History";}

    // .. Additional methods, events, etc.
}
```

Properties are extremely common in OO design. The idea is to have private member variables with public wrappers so outside callers don't have direct access. It's boiler plate code and in most cases you can substitute auto implemented properties instead. The syntax is simple. Instead of specifying the private member values in the getters and setters, you leave the getters and setters blank…per se. See the example where the Name and Customer ID properties don't have corresponding private variables. They use { get; set; } instead. The compiler handles the rest.

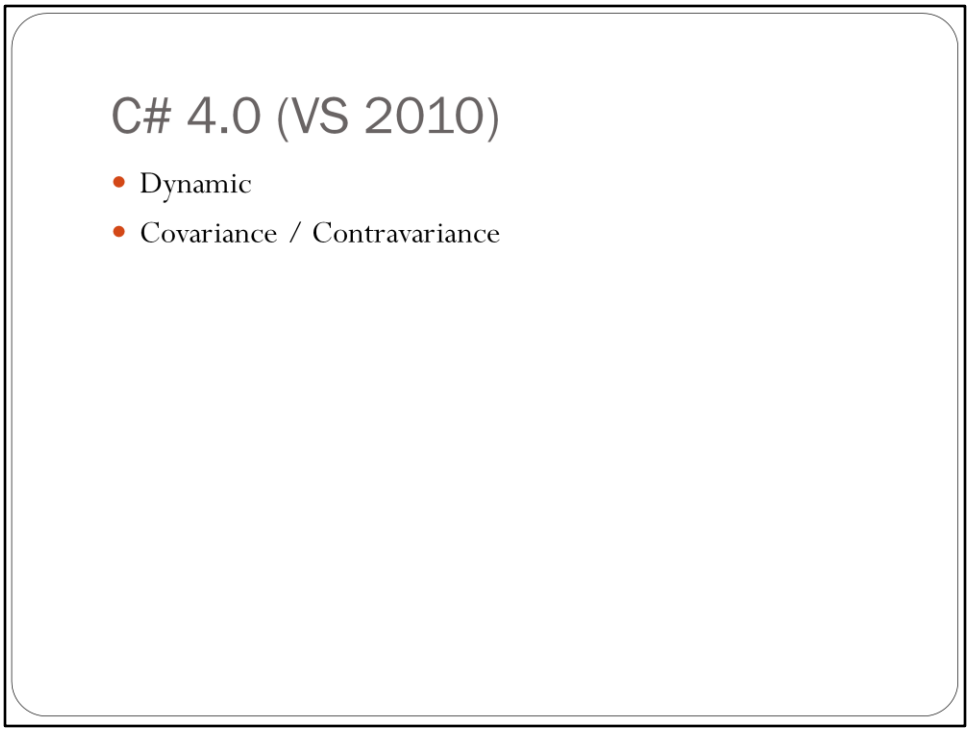
Partial Classes, Methods, and Interfaces

- Not really a language feature, well sort of
- Used a lot with auto generated code
- A way to separate classes, methods, and interface into two different code files.
- During compilation, it combines the different code files into a single class, method, or interface.

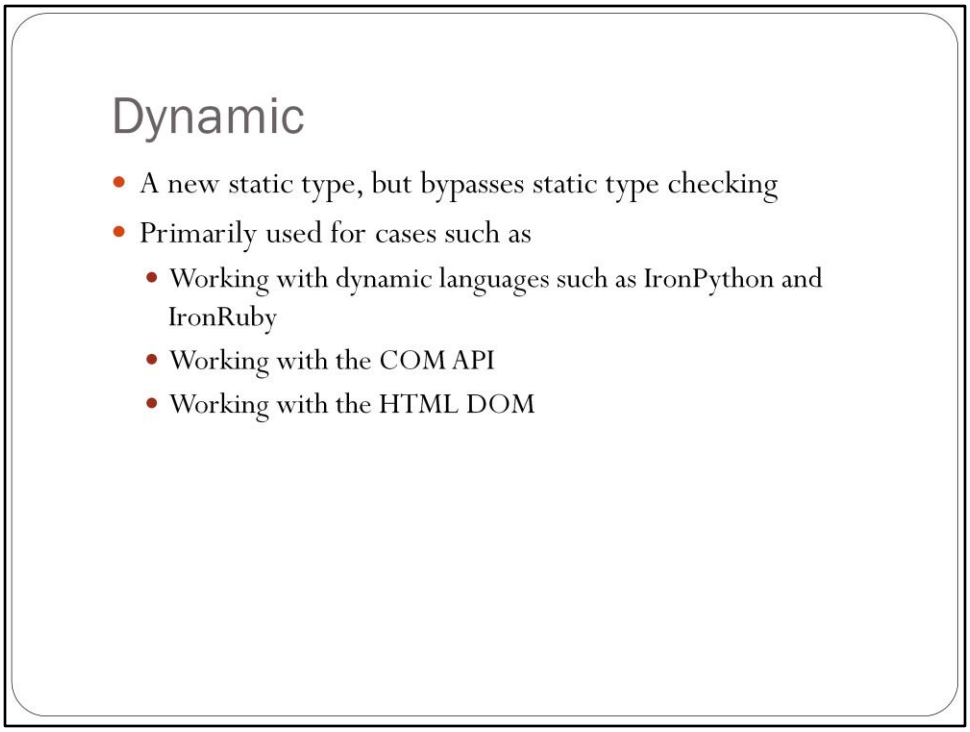
This is used extensively when the IDE creates a lot of auto generated code. You would create a partial class, method, or interface when you need to “hang” something off the auto generated code. Two common examples are designer classes where you have a WYSIWYG tool that generates code in the background and a ORM Tool that might generated a lot of classes based on the database design.



Language integrated query (LINQ) is probably the most important C# feature introduced since it's inception. It allows you to write a more function (declarative) code that is easy to read, with fewer lines of code while still being strongly typed so you can take advantage of things like refactoring.
"Language-Integrated Query (LINQ) is a set of features introduced in Visual Studio 2008 that extends powerful query capabilities to the language syntax of C#" - LINQ (Language-Integrated Query)(http://msdn.microsoft.com/en-us/library/vstudio/bb397926.aspx)
Querying using LINQ doesn't have to be a database. You can query against any sort of set information that implements IEnumerable<T>. That is, pretty much anything in the language that is made up of "things." If it doesn't implement this interface then there are extension methods that allow you to convert inline.



Not much was added in C# 4 other than the “Dynamic” keyword which let’s you do some late binding typing and the Covariance and Contravariance which has to do with derived types. Other features includes in this release include some MS Office enhancements and some type improvements but I won’t focus on them here.



Dynamic is not used much in C# because it's a strongly typed language. However you can in some instances use it in cases where you need loosely type, late binding behavior. Generally this is when you are working with other APIs. IronRuby and IronPython are not strongly typed and are built on top of the dynamic language runtime (DLR). C# can interact with the DLR and languages by using the Dynamic keyword. Other APIs that it's usually working are the Office runtime components as well as the COM APIs.

Covariance

- Preserves assignment compatibility
- Allows a more derived type to be converted to a less derived type for type parameters (generics)
- Example:

```
// Assignment compatibility.
string str = "test";
// An object of a more derived type is assigned to an object of a less derived type.
object obj = str;

// Covariance.
IEnumerable<string> strings = new List<string>();
// An object that is instantiated with a more derived type argument
// is assigned to an object instantiated with a less derived type argument.
// Assignment compatibility is preserved.
IEnumerable<object> objects = strings;
```

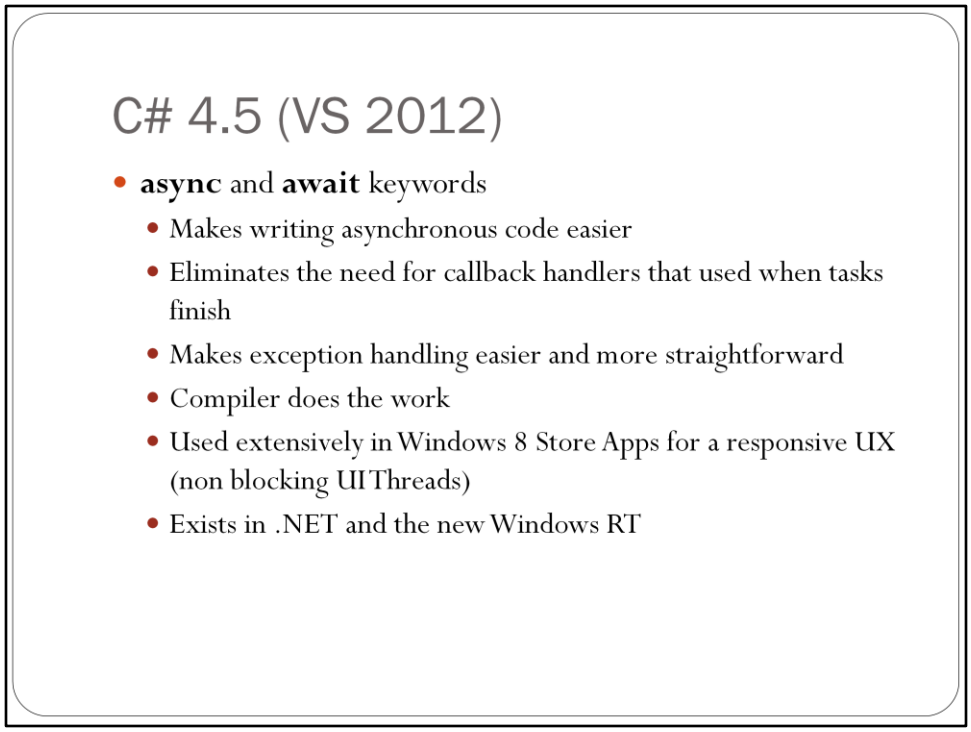
Covariance is sort of the opposite of Contravarience. It's used in generics where type parameters are passed. If you have a generic with a type parameter that is less derived from the one used then assignment compatibility is preserved. In the example above, the type *object*, which the most basic type is used to create a list of objects. But a list of *strings* is passed in that list of objects. Even though string is a more derived type, it works.

Contravariance

- Reserves assignment compatibility
- An object that's created from a less derived type that's assigned to an object of a more derived type has assignment compatibility reversed.
- Example

```
// Assignment compatibility.
string str = "test";
// An object of a more derived type is assigned to an object of a less derived type.
object obj = str;

// Contravariance.
// Assume that the following method is in the class:
// static void SetObject(object o) { }
Action<object> actObject = SetObject;
// An object that is instantiated with a less derived type argument
// is assigned to an object instantiated with a more derived type argument.
// Assignment compatibility is reversed.
Action<string> actString = actObject;
```

Contravarience allows a less derived type to be used as a type parameter in a delegate.



The goal of the Async and Wait keywords is to make asynchronous programming easier.

async

- New keyword in C# 4.5
- Applied to a method
- Returns Task or Task<TResult>
  - Represents ongoing work of the async method
- Task contains information for the caller of the async method such as the status, ID, and the results of the method

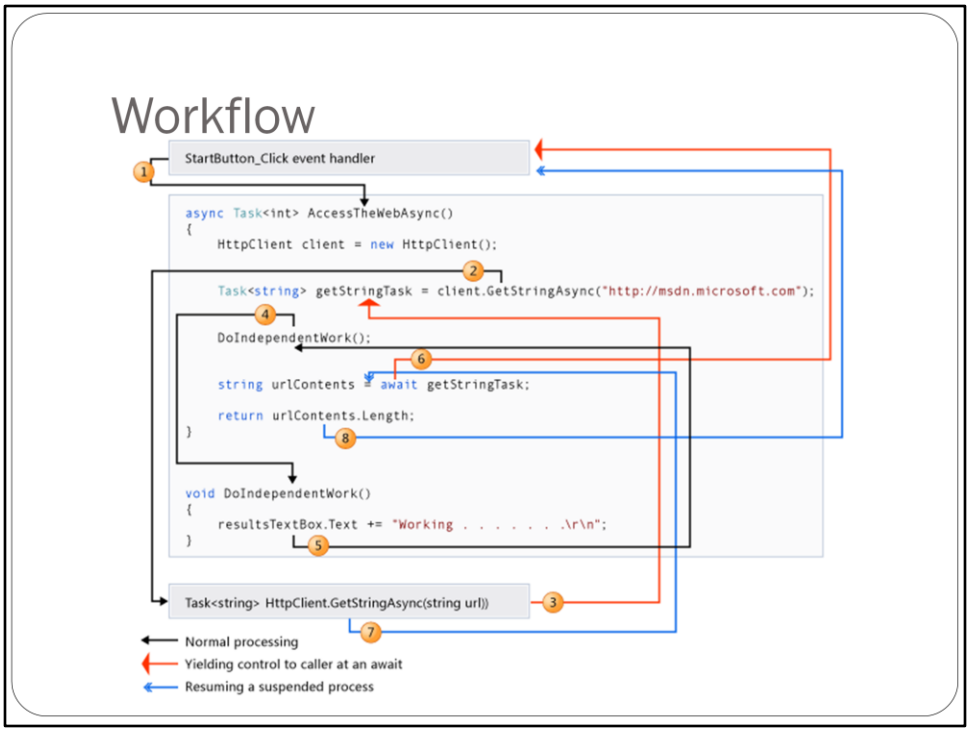
await

- Is applied to the returned Task of the async method
- Suspends execution of the method until complete (control is returned to call in the meantime)

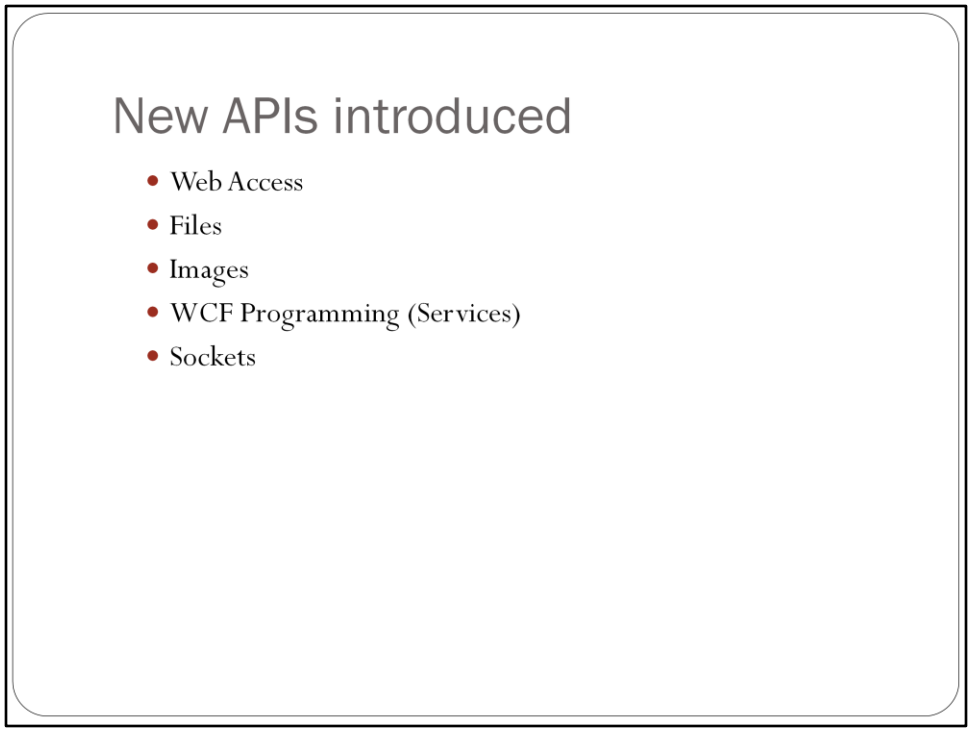
Example

```
// Three things to note in the signature:
//  - The method has an async modifier.
//  - The return type is Task or Task<T>. (See "Return Types" section.)
//    Here, it is Task<int> because the return statement returns an integer.
//  - The method name ends in "Async."
async Task<int> AccessTheWebAsync()
{
    // You need to add a reference to System.Net.Http to declare client.
    HttpClient client = new HttpClient();

    // GetStringAsync returns a Task<string>. That means that when you await the
    // task you'll get a string (urlContents).
    Task<string> getStringTask = client.GetStringAsync("http://msdn.microsoft.com");

    // You can do work here that doesn't rely on the string from GetStringAsync.
    DoIndependentWork();

    // The await operator suspends AccessTheWebAsync.
    //  - AccessTheWebAsync can't continue until getStringTask is complete.
    //  - Meanwhile, control returns to the caller of AccessTheWebAsync.
    //  - Control resumes here when getStringTask is complete.
    //  - The await operator then retrieves the string result from getStringTask.
    string urlContents = await getStringTask;

    // The return statement specifies an integer result.
    // Any methods that are awaiting AccessTheWebAsync retrieve the length value.
    return urlContents.Length;
}
```

This example is taken directly from http://msdn.microsoft.com/en-us/library/hh191443.aspx and is source code.



This example is taken directly from http://msdn.microsoft.com/en-us/library/hh191443.aspx and illustrates a the workflow of what happens during a call.



The await and async keywords were introduced in these APIs.