C# 6 New Features Overview

Using C# 6 with Xamarin

Overview

This document introduces the new features of C# 6. It is fully supported by the mono compiler and developers can start using the new features across all the Xamarin target platforms.

This article includes brief snippets of the C# 6 code that illustrate basic use. The sample application is a command-line program that runs across all Xamarin target platforms and exercises the various features.

Requirements

Development Environment

Mac

- **Xamarin Studio for OS X** has partial support for C# 6: you can build and compile Xamarin apps using C# 6 features, however some syntax highlighting, formatting, and refactoring operations may not work as expected. Our Xamarin Studio "Roslyn" Preview has more robust C# 6.0 support.

Windows

- **Visual Studio 2015** has full support for C# 6. Earlier versions of Visual Studio (eg. 2013, 2012) will not support C# 6.

- **Xamarin Studio for Windows** does not currently support C# 6 features in the editor. This will be available in a future preview.

Compiler

The Mono C# 6 compiler is included in Mono 4.0 and later, which is freely available for download. Xamarin Studio automatically updates the Mono installation on your system.

Windows users must have Visual Studio 2015^ installed to compile C# 6 code (even if you choose Xamarin Studio for Windows as your IDE).

^ or *Microsoft Build Tools 2015* for command line compilation or build servers, for example.

Using C# 6

The C# 6 compiler is used in all recent versions of Xamarin Studio. Those using command-line compilers should confirm that `mcs --version` returns 4.0 or higher. Xamarin Studio users can check if they have Mono 4 (or newer) installed by referring to **About Xamarin Studio > Xamarin Studio > Show Details**.

Less Boilerplate

using static

Enumerations, and certain classes such as `System.Math`, are primarily holders of static values and functions. In C# 6, you can import all static members of a type with a single `using static` statement. Compare a typical trigonometric function in C# 5 and C# 6:

```
// Classic C#
class MyClass
{
    public static Tuple<double,double> SolarAngleOld(double
latitude, double declination, double hourAngle)
    {
        var tmp = Math.Sin (latitude) * Math.Sin (declination) +
Math.Cos (latitude) * Math.Cos (declination) * Math.Cos (hourAngle);
        return Tuple.Create (Math.Asin (tmp), Math.Acos (tmp));
    }
}

// C# 6
using static System.Math;

class MyClass
{
    public static Tuple<double, double> SolarAngleNew(double
latitude, double declination, double hourAngle)
    {
        var tmp = Asin (latitude) * Sin (declination) + Cos
(latitude) * Cos (declination) * Cos (hourAngle);
        return Tuple.Create (Asin (tmp), Acos (tmp));
    }
}
```

`using static` does not make public `const` fields, such as `Math.PI` and `Math.E`, directly accessible:

```
for (var angle = 0.0; angle <= Math.PI * 2.0; angle += Math.PI / 8)
... //PI is const, not static, so requires Math.PI
```

using static with Extension Methods

The `using static` facility operates a little differently with extension methods. Although extension methods are written using `static`, they don't make sense without an instance on which to operate. So when `using static` is used with a type that defines extension methods, the extension methods become available on their target type (the method's `this` type). For instance, `using static System.Linq.Enumerable` can be used to extend the API of `IEnumerable<T>` objects without bringing in all of the LINQ types:

```
using static System.Linq.Enumerable;
using static System.String;

class Program
{
    static void Main()
    {
        var values = new int[] { 1, 2, 3, 4 };
        var evenValues = values.Where (i => i % 2 == 0);
        System.Console.WriteLine (Join(",”, evenValues));
    }
}
```

The previous example demonstrates the difference in behavior: the extension method `Enumerable.Where` is associated with the array, while the static method `String.Join` can be called without reference to the `String` type.

nameof Expressions

Sometimes, you want to refer to the name you’ve given a variable or field. In C# 6, `nameof(someVariableOrFieldOrType)` will return the string `"someVariableOrFieldOrType"`. For instance, when throwing an `ArgumentException` you’re very likely to want to name which argument is invalid:

```
throw new ArgumentException ("Problem with " +
nameof(myInvalidArgument))
```

The chief advantage of `nameof` expressions is that they are type-checked and are compatible with tool-powered refactoring. The type-checking of `nameof` expressions is particularly welcome in situations where a `string` is used to dynamically associate types. For instance, in iOS a `string` is used to specify the type used to prototype `UITableViewCell` objects in a `UITableView`. `nameof` can assure this association does not fail due to a misspelling or sloppy refactoring:

```
public override UITableViewCell GetCell (UITableView tableView,
NSIndexPath indexPath)
{
    var cell = tableView.DequeueReusableCell (nameof(CellTypeA),
indexPath);
    cell.TextLabel.Text = objects [indexPath.Row].ToString ();
    return cell;
}
```

Although you can pass a qualified name to `nameof`, only the final element (after the last `.`) is returned. For instance, you can add a data binding in Xamarin.Forms:

```
var myReactiveInstance = new ReactiveType ();
var myLabelOld.BindingContext = myReactiveInstance;
var myLabelNew.BindingContext = myReactiveInstance;
var myLabelOld.SetBinding (Label.TextProperty, "StringField");
var myLabelNew.SetBinding (Label.TextProperty,
nameof(ReactiveType.StringField));
```

The two calls to `SetBinding` are passing identical values: `nameof(ReactiveType.StringField)` is `"StringField"`, not `"ReactiveType.StringField"` as you might initially expect.

Null-conditional Operator

Earlier updates to C# introduced the concepts of nullable types and the null-coalescing operator `??` to reduce the amount of boilerplate code when handling nullable values. C# 6 continues this theme with the "null-conditional operator" `?.`. When used on an object on the right-hand side of an expression, the null-conditional operator returns the member value if the object is not `null` and `null` otherwise:

```
var ss = new string[] { "Foo", null };
var length0 = ss [0]?.Length; // 3
var length1 = ss [1]?.Length; // null
var lengths = ss.Select (s => s?.Length ?? 0); //[3, 0]
```

(Both `length0` and `length1` are inferred to be of type `int?`)

The last line in the previous example shows the `?` null-conditional operator in combination with the `??` null-coalescing operator. The new C# 6 null-conditional operator returns `null` on the 2nd element in the array, at which point the null-coalescing operator kicks in and supplies a 0 to the `lengths` array (whether that's appropriate or not is, of course, problem-specific).

The null-conditional operator should tremendously reduce the amount of boilerplate null-checking necessary in many, many applications.

There are some limitations on the null-conditional operator due to ambiguities. You cannot immediately follow a `?` with a parenthesized argument list, as you might hope to do with a delegate:

```
SomeDelegate?("Some Argument") // Not allowed
```

However, `Invoke` can be used to separate the `?` from the argument list and is still a marked improvement over a `null`-checking block of boilerplate:

```
public event EventHandler HandoffOccurred;
public override bool ContinueUserActivity (UIApplication
application, NSUserActivity userActivity,
UIApplicationRestorationHandler completionHandler)
{
    HandoffOccurred?.Invoke (this, userActivity.UserInfo);
    return true;
}
```

String Interpolation

The `String.Format` function has traditionally used indices as placeholders in the format string, e.g., `String.Format("Expected: {0} Received: {1}.", expected, received)`. Of course, adding a new value has always involved an annoying little task of counting up arguments, renumbering placeholders, and inserting the new argument in the right sequence in the argument list.

C# 6's new string interpolation feature greatly improves upon `String.Format`. Now, you can directly name variables in a string prefixed with a `$`. For instance:

```
$"Expected: {expected} Received: {received}."
```

Variables are, of course, checked and a misspelled or non-available variable will cause a compiler error.

The placeholders do not need to be simple variables, they can be any expression. Within these placeholders, you can use quotation marks *without* escaping those quotations. For instance, note the `"s"` in the following:

```
var s = $"Timestamp: {DateTime.Now.ToString ("s",
System.Globalization.CultureInfo.InvariantCulture )}"
```

String interpolation supports the alignment and formatting syntax of `String.Format`. Just as you previously wrote `{index, alignment:format}`, in C# 6 you write `{placeholder, alignment:format}`:

```
using static System.Linq.Enumerable;
using System;

class Program
{
    static void Main ()
    {
        var values = new int[] { 1, 2, 3, 4, 12, 123456 };
        foreach (var s in values.Select (i => $"The value is {
i,10:N2}.")) {
            Console.WriteLine (s);
        }
  Console.WriteLine ($"Minimum is { values.Min(i => i):N2}.");
    }
}
```

results in:

```
The value is       1.00.
The value is       2.00.
The value is       3.00.
The value is       4.00.
The value is      12.00.
The value is 123,456.00.
Minimum is 1.00.
```

String interpolation is syntactic sugar for `String.Format`: it cannot be used with `@""` string literals and is not compatible with `const`, even if no placeholders are used:

```
const string s = $"Foo"; //Error : const requires value
```

In the common use-case of building function arguments with string interpolation, you still need to be careful about escaping, encoding, and culture issues. SQL and URL queries are, of course, critical to sanitize. As with `String.Format`, string interpolation uses the `CultureInfo.CurrentCulture`. Using `CultureInfo.InvariantCulture` is a little more wordy:

```
Thread.CurrentThread.CurrentCulture  = new CultureInfo ("de");
Console.WriteLine ($"Today is: {DateTime.Now}"); //"21.05.2015
13:52:51"
```

```
Console.WriteLine ($"Today is:
{DateTime.Now.ToString(CultureInfo.InvariantCulture)}");
//"05/21/2015 13:52:51"
```

Initialization

C# 6 provides a number of concise ways to specify properties, fields, and members.

Auto-property Initialization

Auto-properties can now be initialized in the same concise manner as fields. Immutable auto-properties can be written with only a getter:

```
class ToDo
{
    public DateTime Due { get; set; } = DateTime.Now.AddDays(1);
    public DateTime Created { get; } = DateTime.Now;
```

In the constructor, you can set the value of a getter-only auto-property:

```
class ToDo
{
    public DateTime Due { get; set; } = DateTime.Now.AddDays(1);
    public DateTime Created { get; } = DateTime.Now;
    public string Description { get; }

    public ToDo (string description)
    {
       this.Description = description; //Can assign (only in
constructor!)
    }
```

This initialization of auto-properties is both a general space-saving feature and a boon to developers wishing to emphasize immutability in their objects.

Index Initializers

C# 6 introduces index initializers, which allow you to set both the key and value in types that have an indexer. Typically, this is for `Dictionary`-style data structures:

```
partial void ActivateHandoffClicked (WatchKit.WKInterfaceButton
sender)
{
    var userInfo = new NSMutableDictionary {
        ["Created"] = NSDate.Now,
        ["Due"] = NSDate.Now.AddSeconds(60 * 60 * 24),
        ["Task"] = Description
    };
    UpdateUserActivity ("com.xamarin.ToDo.edit", userInfo, null);
    statusLabel.SetText ("Check phone");
}
```

Expression-bodied Function Members

Lambda functions have several benefits, one of which is simply saving space. Similarly, expression-bodied class members allow small functions to be expressed a little more succinctly than was possible in previous versions of C# 6.

Expression-bodied function members use the lambda arrow syntax rather than the traditional block syntax:

```
public override string ToString () => $"{FirstName} {LastName}";
```

Notice that the lambda-arrow syntax does not use an explicit `return`. For functions that return `void`, the expression must also be a statement:

```
public void Log(string message) => System.Console.WriteLine($"
{DateTime.Now.ToString ("s",
System.Globalization.CultureInfo.InvariantCulture )}: {message}");
```

Expression-bodied members are still subject to the rule that `async` is supported for methods but not properties:

```
//A method, so async is valid
public async Task DelayInSeconds(int seconds) => await
Task.Delay(seconds * 1000);
//The following property will not compile
public async Task<int> LeisureHours => await Task.FromResult<char>
(DateTime.Now.DayOfWeek.ToString().First()) == 'S' ? 16 : 5;
```

Exceptions

There's no two ways about it: exception-handling is hard to get right. New features in C# 6 make exception-handling more flexible and consistent.

Exception Filters

By definition, exceptions occur in unusual circumstances, and it can be very difficult to reason and code about *all* the ways an exception of a particular type might occur. C# 6 introduces the ability to guard an execution handler with a runtime-evaluated filter. This is done by adding a `when (bool)` pattern after the normal `catch(ExceptionType)` declaration. In the following, a filter distinguishes a parse error relating to the `date` parameter as opposed to other parsing errors.

```
public void ExceptionFilters(string aFloat, string date, string
anInt)
{
    try
    {
        var f = Double.Parse(aFloat);
        var d = DateTime.Parse(date);
        var n = Int32.Parse(anInt);
    } catch (FormatException e) when (e.Message.IndexOf("DateTime")
> -1) {
        Console.WriteLine ($"Problem parsing \"{nameof(date)}\"
argument");
    } catch (FormatException x) {
        Console.WriteLine ("Problem parsing some other argument");
    }
}
```

await in catch...finally…

The `async` capabilities introduced in C# 5 have been a game-changer for the language. In C# 5, `await` was not allowed in `catch` and `finally` blocks, an annoyance given the value of the `async/await` capability. C# 6 removes this limitation, allowing asynchronous results to be awaited consistently through the program as shown in the following snippet:

```
async void SomeMethod()
{
    try {
        //...etc...
    } catch (Exception x) {
        var diagnosticData = await GenerateDiagnosticsAsync (x);
        Logger.log (diagnosticData);
    } finally {
        await someObject.FinalizeAsync ();
    }
}
```

Summary

The C# language continues to evolve to make developers more productive while also promoting good practices and supporting tooling. This document has given an overview of the new language features in C# 6 and has briefly demonstrated how they are used.