Core .NET

BY JON SKEET, REVISED AND UPDATED BY JEFFRY MORRIS

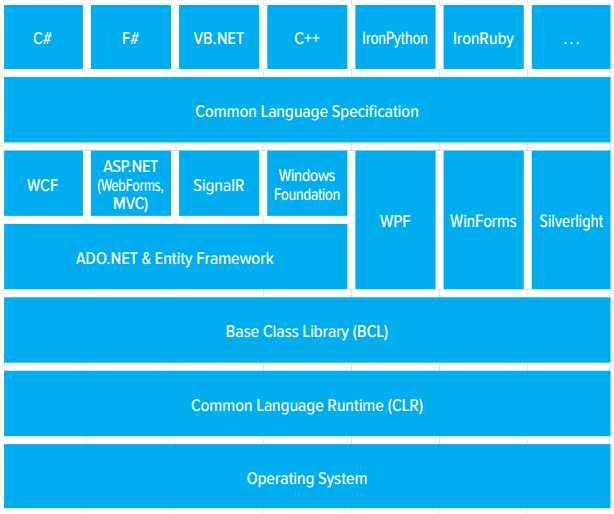
ABOUT .NET

The first version of the .NET Framework was released more than 10 years ago. Over the years, the framework and the ecosystem have evolved drastically. Today, the .NET Framework is available for development on microcontrollers, smartphones, PCs, servers, and so on. Next to being platform independent, the .NET Framework is also language agnostic. Developers can use C#, VB.NET, F#, Python, C++, or even mix programming languages.

Because the .NET Framework now supports a magnitude of platforms, there is no longer a single .NET Framework and different variants exist. Each variant is supporting a slightly different subset of the BCL (Base Class Library).

.NET VARIANT	DESCRIPTION
.NET Framework	The base .NET Framework. For more information: bit.ly/1pGsOHK
.NET Micro Framework	A subset of the .NET Framework for resource-constrained devices with at least 256 KB of flash and 64 KB of RAM. It contains a limited BCL. For more information: bit.ly/1n2iOyz
.NET for Windows Store Apps	A subset of the .NET Framework enabling developers to create Windows Store apps. For more information: bit.ly/1Jrxu48
Silverlight	A subset of the .NET Framework to develop rich internet applications. This subset has become obsolete because Microsoft halted the development of Silverlight. For more information: bit.ly/1TCe50G
Xamarin.Android	An implementation of .NET for Android devices based on the open source Mono framework . Xamarin.Android provides an extended subset of the Silverlight BCL. For more information: bit.ly/1O8BB1Q
Xamarin.iOS	An implementation of .NET for iOS devices based on the open-source Mono framework. It is lacking support for reflection and dynamic code, due to limitations imposed by Apple. For more information: bit.ly/1MQTkZy

Over the years, various frameworks have appeared for the .NET Framework, keeping it up with the latest industry trends: WCF, SignalR, ASP.NET MVC & Web API, WF, WPF, Entity Framework, etc.



It is not possible anymore to cover the entire framework even in a single book, so this Refcard will teach you the important basics you need to know, making it applicable for whatever kind of project you are working on, whether it's ASP.NET, WCF, Windows Phone, Xamarin, Azure, etc.

COMMON .NET TYPES

C# ALIAS	.NET TYPE	SIZE
`object`	`System.Object`	The normal overhead for all reference types is 8 bytes
`string`	`System.String`	14 + 2*(length in characters) bytes
`bool`	`System.Boolean`	1 byte
`byte`	`System.Byte`	1 byte
`sbyte`	`System.SByte`	1 byte

C# ALIAS	.NET TYPE	SIZE
short	System.Int16	2 bytes
ushort	System.UInt16	2 bytes
int	System.Int32	4 bytes
uint	System.UInt32	4 bytes
long	System.Int64	8 bytes
ulong	System.UInt64	8 bytes
float	System.Single	4 bytes (accurate to 7 significant digits)
double	System.Double	8 bytes (accurate to 15 significant digits)
decimal	System.Decimal	16 bytes (accurate to 28 significant digits)
char	System.Char	2 bytes

Apart from `Object` and `String`, all the types above are value types.

When choosing between the three fractional number types (`Single`, `Double`, and `Decimal`):

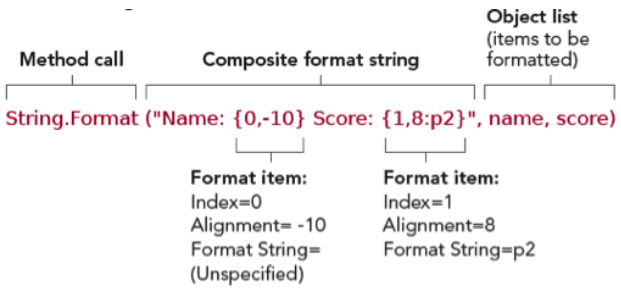
- For financial calculations (i.e. when dealing with money), use Decimal.
- For scientific calculations (i.e. when dealing with physical quantities with theoretically infinite precision, such as weights), use Single or Double.

The `Decimal` type is better suited for quantities that occur in absolutely accurate amounts that can be expressed as decimals. For example, 0.1 can be expressed exactly as a `Decimal` but not as a `Double`. For more information, read pobox.com/~skeet/csharp/decimal.html and pobox.com/~skeet/csharp/floatingpoint.html.

FORMATTING STRINGS

One common task that always has me reaching for MSDN is working out how to format numbers, dates, and times as strings. There are two ways of formatting in .NET: you can call `ToString` directly on the item you wish to format, passing in just a format string or using composite formatting with a call to `String.Format` to format more than one item at a time, or mix data and other text. In either case you can usually specify an `IFormatProvider` (such as `CultureInfo`) to help with internationalization. Many other methods in the .NET Framework also work with composite format strings, such as `Console.WriteLine` and `StringBuilder.AppendFormat`.

Composite format strings consist of normal text and format items containing an index and optionally an alignment and a format string. The following figure shows a sample of using composite format string, with each element labeled.



When the alignment isn't specified you omit the comma; when the format string isn't specified you omit the colon. Every format item must have an index as this says which of the following arguments to format. Arguments can be used any number of times, and in any order. In general, the alignment is used to specify a minimum width—if this is negative, the result is padded with spaces to the right; if it's positive, the result is padded with spaces to the left. The effect of the format string depends on the type of item being formatted. To include a brace as normal text in a composite format string (instead of it indicating the beginning or end of a format item), just double it. For example, the result of `String.Format("{{0}}")` is `"{0}"`.

NUMERIC FORMAT STRINGS

Numbers can be formatted in two ways: with standard or custom format strings. The standard ones allow some flexibility in terms of the precision and style, but the custom ones can be used for very specific formats.

Standard numeric format strings

Standard numeric format strings all take the form of a single letter (the format specifier) and then optionally one or two digits (the precision). For example, a format string of `N5` has `N` as the format specifier and 5 as the precision. The exact meaning of the precision depends on the format specifier, as described in the following table. If no precision is specified, a suitable default is used based on the current `IFormatProvider`.

FORMAT	DESCRIPTION	PRECISION	EXAMPLES (WITH US ENGLISH IFORMATPROVIDER)
C or c	Currency—exact format is specified by `NumberFormatInfo`	Number of decimal places	`123.4567, "c"` => `"$123.46"` `123.4567, "c3"` => `"$123.457"`
D or d	Decimal (integer types only)	Minimum number of digits	`123, "d5"` => `"00123"` `123, "d2"` => `"123"`
E or e	Scientific—used to express very large or small numbers in exponential format	Number of digits after the decimal point	`123456, "e2"` => `"1.23e+005"` `123456, "E4"` => `"1.2345E+005"`
F or f	Fixed point	Number of decimal places	`123.456, "f2"` => `"123.46"` `123.4, "f3"` => `"123.400"`

G or g	General—chooses fixed or scientific notation based on number type and precision	Depends on exact format used (see bit.ly/1R9jy1g for details)	`123.4, "g2"` => `"1.2e+02"`
			`123.4, "g6"` => `"123.4"`
			`123.400m, "g"` => `"123.400"`

N or n	Number—decimal form including thousands indicators (e.g. commas)	Number of decimal places	`1234.567, "n2"` => `"1,234.57"`
P or p	Percentage—number is multiplied by 100 and percentage sign is applied	Number of decimal places	`0.1234, "p1"` => `"12.3 %"`

FORMAT	DESCRIPTION	PRECISION	EXAMPLES (WITH US ENGLISH IFORMATPROVIDER)
R or r	Round-trip—if you later parse the result, you're guaranteed to get the original number	Ignored	`0.12345, "r" => 0.12345`
X or x	Hexadecimal (integer types only)—the case of the result is the same as the case of the format specifier	Minimum number of digits	`123, "x" => "7b"` `123, "X4" => "007B"`

Custom numeric format strings

To format numbers in a custom fashion, you provide a pattern to the formatter, consisting of format specifiers as shown in the following table.

FORMAT SPECIFIER	NAME	DESCRIPTION
`0`	Zero placeholder	Always formatted as 0 or a digit from the original number.
`#`	Decimal placeholder	Formatted as a digit when it's a significant digit in the number, or omitted otherwise.
`.`	Decimal point	Formatted as the decimal point for the current `IFormatProvider`.
`,`	Thousands separator and number scaling specifier	When used between digit or zero placeholders, formatted as the group separator for the current `IFormatProvider`. When it's used directly before a decimal point (or an implicit decimal point) each comma effectively means "divide by a thousand."
`%`	Percentage placeholder	Formatted as the percent symbol for the current `IFormatProvider`, and also multiplies the number by 100.
`‰ (\u2030)`	Per mille placeholder	Similar to the percentage placeholder, but the number is multiplied by 1000 instead of 100, and the per mille symbol for the culture is used instead of the percent symbol.
`E0, e0, E+0, e+0, E-0, or e-0`	Scientific notation	Formats the number with scientific (exponential) notation. The number of 0s indicates the minimum number of digits to use when expressing the exponent. For `E+0` and `e+0`, the exponent's sign is always expressed; otherwise it's only expressed for negative exponents.
`" or '`	Quoting for literals	Text between quotes is formatted exactly as it appears in the format string (i.e. it's not interpreted as a format pattern).
`;`	Section separator	A format string can consist of up to three sections, separated by semi-colons. If only a single section is present, it is used for all numbers. If two sections are present, the first is used for positive numbers and zero; the second is used for negative numbers. If three sections are present, they are used for positive, negative, and zero numbers, respectively.
`\c`	Single-character escape	Escapes a single character (i.e. the character c is displayed verbatim).

The following table shows examples of custom numeric format strings when formatted with a US English format provider.

NUMBER	FORMAT STRING	OUTPUT	NOTES
123	`####.00#`	123.00	0 forces a digit; # doesn't
12345.6789	`####.00#`	12345.679	Value is rounded to 3 decimal places
1234	`0,0.#`	1,234	Decimal point is omitted when not required
1234	`0,.####`	1.234	Value has been divided by 1000
0.35	`0.00%`	35.00%	Value has been multiplied by 100
0.0234	`0.0\u2030`	23.4‰	
0.1234	`0.00E0`	1.23E-1	Exponent specified with single digit
1234	`0.00e00`	1.23e03	Exponent is specified with two digits, but sign is omitted
1234	`##'text0'###`	1text0234	The texto part is not parsed as a format pattern
12.34	`0.0;000.00; 'zero'`	12.3	First section is used
-12.34	`0.0;000.00; 'zero'`	012.34	Second section is used
0	`0.0;000.00; 'zero'`	zero	Third section is used

Date and time format strings

Dates and times tend to have more cultural sensitivity than numbers—the ordering of years, months, and days in dates varies between cultures, as do the names of months, and so forth. As with numbers, .NET allows both standard and custom format strings for dates and times.

Standard date and time format strings

Standard date and time format strings are always a single character. Any format string that is longer (including whitespace) is interpreted as a custom format string. The round-trip (o or O), RFC1123 (r or R), sortable (s), and universal sortable (u) format specifiers are culturally invariant—in other words, they will produce the same output whichever `IFormatProvider` is used. The following table lists all of the standard date and time format specifiers.

FORMAT SPECIFIER	DESCRIPTION	EXAMPLE (US ENGLISH)
`d`	Short date pattern	5/30/2008
`D`	Long date pattern	Friday, May 30, 2008
`f`	Full date/time pattern (short time)	Friday, May 30, 2008 8:40 PM
`F`	Full date/time pattern (long time)	Friday, May 30, 2008 8:40:36 PM
`g`	General date/time pattern (short time)	5/30/2008 8:40 PM
`G`	General date/time pattern (long time)	5/30/2008 8:40:36 PM
`M or m`	Month day pattern	May 30

FORMAT SPECIFIER	DESCRIPTION	EXAMPLE (US ENGLISH)
O or o	Round-trip pattern	2008-05-30T20:40:36.8460000+01:00
R or r	RFC1123 pattern (assumes UTC: caller must convert)	Fri, 30 May 2008 19:40:36 GMT
s	Sortable date pattern (ISO 8601 compliant)	2008-05-30T20:40:36
t	Short time pattern	8:40 PM
T	Long time pattern	8:40:36 PM
u	Universal sortable date pattern (Assumes UTC: caller must convert)	2008-05-30 19:40:36Z
U	Universal full date/time pattern (Format automatically converts to UTC)	Friday, May 30, 2008 7:40:36 PM
Y or y	Year month pattern	May, 2008

Custom date and time format strings

As with numbers, custom date and time format strings form patterns which are used to build up the result. Many of the format specifiers act differently depending on the number of times they're repeated. For example, "d" is used to indicate the day—for a date falling on a Friday and the 5th day of the month, "d" (in a custom format string) would produce "5", "dd" would produce "05", "ddd" would produce "Fri" and "dddd" would produce "Friday" (in US English—other cultures will vary). The following table shows each of the custom date and time format specifiers, describing how their meanings change depending on repetition.

FORMAT SPECIFIER	MEANING	NOTES AND VARIANCE BY REPETITION
d, dd, ddd, dddd	Day	d 1-31 dd: 01-31 ddd: Abbreviated day name (e.g. Fri) dddd: Full day name (e.g. Friday)
f, ff ... fffffff	Fractions of a second	f: Tenths of a second ff: Hundredths of a second The specified precision is always used, with insignificant zeroes included if necessary
F, FF ... FFFFFFF	Fractions of a second	Same as f ... fffffff except insignificant zeroes are omitted
g	Period or era	For example, "A.D."
h, hh	Hour in 12 hour format	h: 1-12 hh: 01-12
H, HH	Hour in 24 hour format	H: 0-23 HH: 00-23
K	Time zone offset	For example, +01:00; outputs Z for UTC values
m, mm	Minute	m: 0-59 mm: 00-59
M ... MMMM	Month	M: 1-12 MM: 01-12 MMM: Abbreviated month name (e.g. Jan) MMMM: Full day name (eg January)
s, ss	Seconds	s: 0-59 ss: 00-59
t, tt	AM/PM designator	t: First character only (e.g. "A" or "P") tt: Full designator (e.g. "AM" or "PM")
y ... yyyyy	Year	y: 0-99 (least significant two digits are used) yy: 00-99 (least significant two digits are used) yyy: 000-9999 (three or four digits as necessary) yyyy: 0000-9999 yyyyy: 00000-99999
z ... zzz	Offset from UTC (of local operating system)	z: -12 to +13, single or double digit zz: -12 to +13, always double digit (e.g. +05) zzz: -12:00 to +13:00, hours and minutes
:	Time separator	Culture-specific symbol used to separate hours from minutes, etc.
/	Date separator	Culture-specific symbol used to separate months from days, etc.
'	Quoting for literals	Text between two apostrophes is displayed.
%c	Single custom format specifier	Uses c as a custom format specifier; used to force a pattern to be interpreted as a custom instead of a standard format specifier.
\c	Single-character escape	Escapes a single character (i.e. the character c is displayed verbatim).

Typically only years within the range 1-9999 can be represented, but there are some exceptions due to cultural variations

The following table shows examples of custom date and time format strings when formatted with a US English format provider. The date and time in question is the same one used to demonstrate the standard format strings.

FORMAT STRING	OUTPUT	NOTES
yyyy/MM/dd'T'HH:mm:ss.fff	2008/05/30T20:40:36.846	T is quoted for clarity only—T is not a format specifier, so would have been output anyway
d MMMM yy h:mm tt	30 May 08 8:40 PM	12 hour clock, single digit used

FORMAT STRING	OUTPUT	NOTES
`HH:mm:sszzz`	20:40:36+01:00	24 hour clock, always two digits
`yyyy g`	2008 A.D.	Rarely used—era is usually implicit
`yyyyMMddHHmmssfff`	20080530204036846	Not very readable, but easily sortable—handy for log filenames; consider using UTC though

WORKING WITH DATES AND TIMES

The support in .NET for dates and times has changed significantly over time. It's never simple to do this properly (particularly taking time zones, internationalization, leap years, and other idiosyncrasies into account) but the support has definitely improved.

`DateTime` and `TimeZone` have been in the .NET Framework since version 1.0. `DateTime` simply stores the number of ticks since midnight on January 1st, 1 A.D.—where a tick is 100ns. This structure was improved in .NET 2.0 to allow more sensible time zone handling, but it's still not entirely satisfactory. It's useful when you don't care about time zones, but newer alternatives have been introduced. `TimeZone` is sadly restricted to retrieving the time zone of the local machine.

.NET 2.0SP1 (which is part of .NET 3.0SP1 and .NET 3.5) introduced `DateTimeOffset` which is effectively a `DateTime` with an additional Offset property representing the difference between the local time and UTC. This unambiguously identifies an instant in time. However, it's not inherently aware of time zones—if you add six months, the result will have the same Offset even if the "logical" answer would be different due to Daylight Savings Time.

.NET 3.5 introduced `TimeZoneInfo`, which is a much more powerful class for representing time zones than `TimeZone`—the latter is now effectively deprecated. `TimeZoneInfo` allows you access to all the time zones that the system knows about, as well as creating your own. It also contains historical data (depending on your operating system) instead of assuming that every year has the same rules for any particular time zone.

Tips

- If you're using .NET 2.0SP1 and higher, then consider `DateTimeOffset` to be the "default" date and time type. Some databases are easier to work with using `DateTime`.
- It is usually a good idea to use a UTC representation as much as possible, unless you really need to preserve an original time zone. Convert to local dates and times for display purposes.
- If you need to preserve the original time zone instead of just the offset at a single point in time, and keep the relevant `TimeZoneInfo`.
- There are situations where the time zone is irrelevant, primarily when either just the date or just the time is important. Identify these situations early and make sure you don't apply time zone offsets.
- In almost all commonly used formats:
  - 10:00:00.000+05:00 means "the local time is 10am; in UTC it's 5am"
  - 10:00:00.000-05:00 means "the local time is 10am; in UTC it's 3pm"
- `DateTimeOffset.Offset` is positive if the local time is later than UTC, and negative if the local time is earlier than UTC. In other words, Local = UTC + Offset.

In addition to the types described above, the `Calendar` and `DateTimeFormatInfo` classes in the `System.Globalization` namespace are important when parsing or formatting dates and times. However, their involvement is usually reasonably well hidden from developers.

TEXT ENCODINGS

Internationalization (commonly abbreviated to i18n) is another really thorny topic. Guy Smith-Ferrier's book, .NET Internationalization (Addison-Wesley Professional, 2006) is probably the definitive guide. However, before you even consider what resources you will display to the user, make sure you can accurately move textual data around—which means you need to know about encodings.

Whenever you use a string in .NET, it uses Unicode for its internal representation. Unicode is a standard way of converting characters ('a', 'b', 'c', etc.) into numbers (97, 98, 99 respectively, in this case). Each number is a 16-bit unsigned integer—in other words, it's in the range 0-65535. You need to be careful how you read your text to start with, and how you output it. This almost always involves converting between the textual representation (your string) and a binary representation (plain bytes)—either in memory or to disk, or across a network. This is where different encodings represent characters differently.

There are actually more than 65536 characters in Unicode, so some have to be stored as pairs of surrogate characters. Most of the time you don't need to worry about this—most useful characters are in the Basic Multilingual Plane (BMP).

The `System.Text.Encoding` class is at the heart of .NET's encoding functionality. Various classes are derived from it, but you rarely need to access them directly. Instead, properties of the `Encoding` class provide instances for various common encodings. Others (such as ones using Windows code pages) are obtained by calling the relevant `Encoding` constructor. The following table describes the encodings you're most likely to come across.

NAME	HOW TO CREATE	DESCRIPTION
UTF-8	`Encoding.UTF8`	The most common multi-byte representation, where ASCII characters are always represented as single bytes, but other characters can take more—up to 3 bytes for a character within the BMP. This is usually the encoding used by .NET if you don't specify one (for instance, when creating a `StreamReader`). When in doubt, UTF-8 is a good choice of encoding.
System default	`Encoding.Default`	This is the default encoding for your operating system—which is not the same as it being the default for .NET APIs! It's typically a Windows code page—1252 is the most common value for Western Europe and the US, for example.

NAME	HOW TO CREATE	DESCRIPTION
UTF-16	`Encoding.Unicode`, `Encoding.BigEndianUnicode`	UTF-16 represents each character in a .NET string as 2 bytes, whatever its value. `Encoding.Unicode` is little-endian, as opposed to `.BigEndianUnicode`.
ASCII	`Encoding.ASCII`	ASCII contains Unicode values 0-127. It does not include any accented or "special" characters. "Extended ASCII" is an ambiguous term usually used to describe one of the Windows code pages.
Windows code page	`Encoding.GetEncoding(page)`	If you need a Windows code page encod¬ing other than the default, use `Encoding.GetEncoding(Int32)`.
ISO-8859-1 ISO-Latin-1	`Encoding.GetEncod-ing(28591)`	Windows code page 28591 is also known as ISO-Latin-1 or ISO-8859-1, which is reasonably common outside Windows.
UTF-7	`Encoding.UTF7`	This is almost solely used in email, and you're unlikely to need to use it. I only mention it because many people think they've got UTF7-encoded text when it's actually a different encoding entirely.

ASYNCHRONOUS OPERATIONS

Writing good multi-threaded code is complex. It is relatively simple to understand the basics of threads and parallel code, but in practice keeping track of what's going on can be extremely tricky. For most developers and in most applications it's recommended to avoid using threads directly. It's much better to use one of the higher levels of abstractions for asynchronous operations Microsoft has introduced over the years into .NET.

PROGRAMMING MODEL OR PATTERN	DESCRIPTION
Asynchronous Programming Model (APM)	An asynchronous operation that is implemented as two methods, by convention named `BeginOperationName` and `EndOperationName`. The `BeginOperationName` returns an `IAsyncResult` object and the `EndOperationName` requires the `IAsyncResult` in input. After calling the `BeginOperationName`, the developer is free to call other methods on the current thread while the operation is executing in the background. In order to get the result of the operation, `EndOperationName` must be called. For example: `FileStream.BeginRead` and `FileStream.EndRead`.
Event-based asynchronous pattern (EAP)	An asynchronous operation that is implemented as a combination of a method and an event, by convention the method is named `OperationNameAsync` and the event `OperationNameCompleted`. When you call the method `OperationNameAsync`, the operation will be executed in the background on a separate thread. The call will not block the current thread. When the operation is done the event handler will be called. The result of the operation is passed through as the `AsyncCompletedEventArgs` parameter of the event. For example: `PictureBox.LoadAsync` and `PictureBox.LoadCompleted`.
Task-based Asynchronous Pattern (TAP)	The TAP or Task-based Asynchronous Pattern is based off the Task Parallel Library (TPL) and use the async and await keywords and method naming conventions to make it easier to consume and write asynchronous code. When TAP methods are available for a given API, they are the recommended asynchronous programming model over the APM and EAP methods if they exist. For example: `HttpClient.GetAsync`.

The Task Parallel Library (TPL) has been introduced in .NET 4.0 and is now the recommended asynchronous programming model. The TPL uses tasks as abstraction for the parallelism. Each task represents an asynchronous operation. They provide a higher level of abstraction than threads. The TPL provides a rich set of APIs for waiting, cancellation, continuation, robust exception handling, etc.

The TPL is responsible for determining and adjusting the number of threads required for maximum efficiency to execute all the scheduled tasks.

The TPL supports Data Parallelism, Task Parallelism, and Dataflow components. Data Parallelism is for scenarios where the operation is performed concurrently on elements within a collection or array using the familiar for and foreach constructs. Task Parallelism includes the familiar TAP methods and represent asynchronous operations on a Task as an abstraction. The Dataflow Library contains components that allow for concurrent applications to be built which need to communicate with one another asynchronously or process data as it becomes available.

Microsoft has updated the existing frameworks such as WCF, ASP.NET MVC to support the TPL.

In .NET 4.5, Microsoft has added the async and await keywords to the C# programming language to further simplify working with the TPL. Methods marked with the async keyword will run asynchronously. Code within these methods will wait on the results of other async methods by using the await keyword before a method call.

If you want to use async/await in a .NET 4.0, Silverlight 4 or Windows Phone 7.5 application, it is possible because Microsoft has back-ported the necessary support. But you will need to add the NuGet package `Microsoft.Bcl.Async` to your project. In Windows 8, all operations that potentially may require longer then 50ms are asynchronous by default and rely on the TPL, which makes it practically impossible to develop a Windows 8 application without understanding how async/await functions.

WRITING AN ASYNC METHOD

Writing an `async` method is not complex. The method must be marked with the `async` keyword and the return type must be `void`, `Task`, or `Task<TResult>`. The name of an `async` method should end with "Async", although this is by convention and not mandatory.

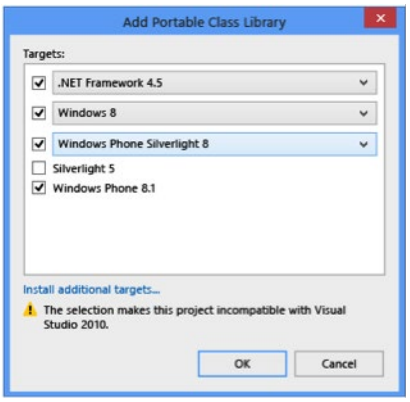
RETURN TYPE	DESCRIPTION
`void`	An `async void` method is not recommended. Because such a method does not return a `Task`, the callers have no easy way to know when the operation was finished. Furthermore, all thrown exceptions within the `async void` method are lost, because the uncaught exceptions of async methods are stored on the `Task` object (which `async void` methods do not have).
`Task`	The `async Task` method is the asynchronous equivalent of the traditional `void` method. The operation does not provide a return value, but callers can be notified when the operation is finished and can handle the uncaught exceptions thrown from within the `async` method.
`Task<TResult>`	The `async Task<TResult>` method is similar to the above one, but the difference is that a `Task<TResult>` can return a value of type `TResult` through the `Task` object. By using the `await` keyword the return value is extracted from the `Task` object.

```
// Mark the method as asynchronous
private async Task<int> CalculateDZonePageSizeAsync()
{
    var client = new HttpClient();
    var task = await client.GetStringAsync("http:…");

    // Do some synchronous stuff while
    // the HTTP request is ongoing
    DoSomeStuff();

    // Wait on the result of the HTTP Request
    var content = await task;

    // Return the result, the caller could
    // already continue executing because
    // the method is asynchronous
    return content.Length;
}
```

PORTABLE CLASS LIBRARIES

The .NET Framework now runs on various platforms. Unfortunately these different platforms often use different subsets of the Base Class Libraries (BCL) and therefore they are using a different .NET Core Library Profile. Each platform can only use libraries targeting its library profile, making it so that third party frameworks would require separate class library projects for each platform.

Microsoft introduced the Portable Class Library specification to resolve this issue. Portable Class Libraries are assemblies that work on multiple platforms (with different .NET core library profiles). Portable Class Libraries are supported in .NET 4.0+ and Xamarin. A lot of popular Microsoft and third party frameworks are now PCLs (HttpCLient, Json.NET, OxyPlot, MVVM Light, etc.).



Portable Class Libraries are a specific project type in Visual Studio. When you create such a project, Visual Studio will display a dialog to select the targeted platforms. The choice of the platforms will impact the supported features.

ABOUT THE AUTHOR

JEFF MORRIS is a Senior Software Engineer at Couchbase, Inc. and maintains the Open Source Couchbase .NET SDK. Jeff has over 14 years' experience developing and architecting systems under a variety of platforms and technology; but primarily focusing on Microsoft solutions.

RESOURCES

- .NET Home
- .NET Documentation
- .NET GitHub
- .NET Framework and SDK Downloads

DZone

DZone communities deliver over 6 million pages each month to more than 3.3 million software developers, architects and decision makers. DZone offers something for everyone, including news, tutorials, cheat sheets, research guides, feature articles, source code and more.

"DZone is a developer's dream," says PC Magazine.

DZONE, INC.
150 PRESTON EXECUTIVE DR.
CARY, NC 27513

888.678.0399
919.678.0300

REFCARDZ FEEDBACK WELCOME
refcardz@dzone.com

SPONSORSHIP OPPORTUNITIES
sales@dzone.com

ISBN-13: 978-1-936502-77-6
ISBN-10: 1-936502-77-1
50795
9 781936 502776

VERSION 1.0