

C# Notes for Professionals

Chapter 19: DateTime Methods

Section 19.1: DateTime Formatting

Standard DateTime Formatting:
DateTimeFormatter specifies a set of specifiers for simple date and time formatting. Every specifier is a particular DateTimeFormatter format pattern.

```
//Create datetime  
DateTime dt = new DateTime(2016, 08, 01, 10, 59, 23, 200);  
  
var t = String.Format("{0:t}", dt); // "10:59 AM"  
var d = String.Format("{0:d}", dt); // "08/01/2016"  
var f = String.Format("{0:f3}", dt); // "Monday, August 1, 2016 10:59:23 PM"  
var F = String.Format("{0:F3}", dt); // "Monday, August 1, 2016 10:59:23 PM"  
var g = String.Format("{0:g}", dt); // "10/01/16 10:59 PM"  
var G = String.Format("{0:G}", dt); // "Monday, August 1, 2016 10:59:23 PM"  
var y = String.Format("{0:y}", dt); // "16/08/01"  
var Y = String.Format("{0:Y}", dt); // "2016/08/01 10:59:23"  
var u = String.Format("{0:u}", dt); // "2016-08-01T10:59:23Z"  
var U = String.Format("{0:U}", dt); // "2016-08-01T10:59:23Z"  
var universalTime = dt.ToString("u");
```

Custom DateTime Formatting

There are following custom format specifiers:

- {t} (type)
- {1} (month)
- {d} (day)
- {h} (hour 12)
- {i} (hour 24)
- {m} (minute)
- {s} (second)
- {F} (second fraction, trailing zeroes are trimmed)
- {M} (AM or AM)
- {z} (time zone)

```
string foo = "some string";  
var year = new DateTime(1990, 01, 01);  
var month = new DateTime(1990, 01, 01, 00, 00, 00);  
var day = new DateTime(1990, 01, 01, 00, 00, 00);  
var hour = new DateTime(1990, 01, 01, 12, 00, 00);  
var minute = new DateTime(1990, 01, 01, 12, 30, 00);  
var second = new DateTime(1990, 01, 01, 12, 30, 45);  
var fraction = new DateTime(1990, 01, 01, 12, 30, 45, 123456);  
var fraction2 = new DateTime(1990, 01, 01, 12, 30, 45, 123456789);  
var period = new DateTime(1990, 01, 01, 12, 30, 45, 123456789, DateTimeKind.Utc);  
var zone = new DateTime(1990, 01, 01, 12, 30, 45, 123456789, DateTimeKind.Utc);
```

You can use also data separator / (slash) and time separator : (colon).

C# Notes for Professionals

Chapter 34: Anonymous types

Section 34.1: Anonymous vs dynamic

Anonymous types allow the creation of objects without having to explicitly define their types ahead of time, while maintaining static type checking.

```
var anon = new { Value = 1 };  
Console.WriteLine(anon.Id); // compile time error
```

Conversely, **dynamic** has dynamic type checking, opting for runtime errors, instead of compile-time errors.

```
dynamic val = "you";  
Console.WriteLine(val.Id); // compiles, but throws runtime error
```

Section 34.2: Creating an anonymous type

Since anonymous types are not named, variables of those types must be implicitly typed (**var**).

```
var anon = new { Foo = 1, Bar = 2 };  
int anon.Foo == 1;  
int anon.Bar == 2;
```

If the member names are not specified, they are set to the name of the property/variable used to initialize the object.

```
int foo = 1;  
int bar = 2;  
var anon = new { foo, bar };  
// anon.Foo == 1  
// anon.Bar == 2
```

Note that names can only be omitted when the expression in the anonymous type declaration is a simple property access; for method calls or more complex expressions, a property name must be specified.

```
string foo = "some string";  
var anon = new { foo.Length == 10 ? "short string" : "long string" };  
// compiler error - Invalid anonymous type member declarator.  
var anon = new { Description = foo.Length == 10 ? "short string" : "long string" };  
// OK
```

Section 34.3: Anonymous type equality

Anonymous type equality is given by the **EqualityComparer** instance method. Two objects are equal if they have the same and equal values (through **IEqualityComparer<T>.Equals(T, T)** for every property).

```
var anon = new { Foo = 1, Bar = 2 };  
var anon2 = new { Foo = 1, Bar = 2 };  
var anon3 = new { Foo = 1, Bar = 10 };  
var anon4 = new { Bar = 10, Foo = 1 };  
var anon5 = new { Bar = 2, Foo = 1 };  
// anon.Equals(anon2) == true  
// anon.Equals(anon3) == false
```

C# Notes for Professionals

Chapter 98: Networking

Section 98.1: Basic TCP Communication Client

This code example creates a TCP client, sends "Hello World" over the socket connection, and then writes the server response to the console before closing the connection.

```
int port = 1990;  
int timeout = 2000;
```

```
// Declare Variables  
string host = "stackoverflow.com";  
int port = 1990;  
int timeout = 2000;  
  
// Create TCP client and connect  
using (var client = new TcpClient(host, port))  
{  
    client.Connect();  
    NetworkStream stream = client.GetStream();  
  
    // Write a message over the socket  
    string message = "Hello World";  
    byte[] dataToSend = System.Text.Encoding.ASCII.GetBytes(message);  
    stream.Write(dataToSend, 0, dataToSend.Length);  
}
```

```
// Read server response  
byte[] responseData = new byte[1024];  
int bytes = stream.Read(responseData, 0, responseData.Length);
```

```
message = System.Text.Encoding.ASCII.GetString(responseData, 0, bytes);
```

```
Console.WriteLine("Response from {0}: {1}", host, message);  
// The client and stream will close as control exits the using block (Dispose() but safer than  
// calling Close());
```

Section 98.2: Download a file from a web server

Downloading a file from the internet is a very common task required by almost every application you're likely to build.

To accomplish this, you can use the **System.Net.WebClient** class. The simplest use of this, using the "using" pattern, is shown below:

```
using (var webClient = new WebClient())  
{  
    webClient.DownloadFile("http://www.server.com/file.txt", "C:\file.txt");
```

What this example does is it uses "using" to make sure that your web client is cleaned up correctly when finished, and simply transfers the named resource from the URL in the first parameter to the named file on your local hard drive in the second parameter.

The first parameter is of type **System.Uri**, the second parameter is of type **System.String**.

You can also use this function in an **async** form, so that it goes off and performs the download in the background, while your application gets on with something else, using the call in the way is of major importance in modern applications, as it helps to keep your user interface responsive.

When you use the **Async** methods, you can hook up event handlers that allow you to monitor the progress, so that

700+ pages
of professional hints and tricks

Contents

About	1
Chapter 1: Getting started with C# Language	2
Section 1.1: Creating a new console application (Visual Studio)	2
Section 1.2: Creating a new project in Visual Studio (console application) and Running it in Debug mode	4
Section 1.3: Creating a new program using .NET Core	7
Section 1.4: Creating a new program using Mono	9
Section 1.5: Creating a new query using LinqPad	9
Section 1.6: Creating a new project using Xamarin Studio	12
Chapter 2: Literals	18
Section 2.1: uint literals	18
Section 2.2: int literals	18
Section 2.3: sbyte literals	18
Section 2.4: decimal literals	18
Section 2.5: double literals	18
Section 2.6: float literals	18
Section 2.7: long literals	18
Section 2.8: ulong literal	18
Section 2.9: string literals	19
Section 2.10: char literals	19
Section 2.11: byte literals	19
Section 2.12: short literal	19
Section 2.13: ushort literal	19
Section 2.14: bool literals	19
Chapter 3: Operators	20
Section 3.1: Overloadable Operators	20
Section 3.2: Overloading equality operators	21
Section 3.3: Relational Operators	22
Section 3.4: Implicit Cast and Explicit Cast Operators	24
Section 3.5: Short-circuiting Operators	25
Section 3.6: ?: Ternary Operator	26
Section 3.7: ?. (Null Conditional Operator)	27
Section 3.8: "Exclusive or" Operator	27
Section 3.9: default Operator	28
Section 3.10: Assignment operator '='	28
Section 3.11: sizeof	28
Section 3.12: ?? Null-Coalescing Operator	29
Section 3.13: Bit-Shifting Operators	29
Section 3.14: => Lambda operator	29
Section 3.15: Class Member Operators: Null Conditional Member Access	31
Section 3.16: Class Member Operators: Null Conditional Indexing	31
Section 3.17: Postfix and Prefix increment and decrement	31
Section 3.18: typeof	32
Section 3.19: Binary operators with assignment	32
Section 3.20: nameof Operator	32
Section 3.21: Class Member Operators: Member Access	33
Section 3.22: Class Member Operators: Function Invocation	33

Section 3.23: Class Member Operators: Aggregate Object Indexing	33
Chapter 4: Conditional Statements	34
Section 4.1: If-Else Statement	34
Section 4.2: If statement conditions are standard boolean expressions and values	34
Section 4.3: If-Else If-Else Statement	35
Chapter 5: Equality Operator	36
Section 5.1: Equality kinds in c# and equality operator	36
Chapter 6: Equals and GetHashCode	37
Section 6.1: Writing a good GetHashCode override	37
Section 6.2: Default Equals behavior	37
Section 6.3: Override Equals and GetHashCode on custom types	38
Section 6.4: Equals and GetHashCode in IEqualityComparer	39
Chapter 7: Null-Coalescing Operator	41
Section 7.1: Basic usage	41
Section 7.2: Null fall-through and chaining	41
Section 7.3: Null coalescing operator with method calls	42
Section 7.4: Use existing or create new	43
Section 7.5: Lazy properties initialization with null coalescing operator	43
Chapter 8: Null-conditional Operators	44
Section 8.1: Null-Conditional Operator	44
Section 8.2: The Null-Conditional Index	44
Section 8.3: Avoiding NullReferenceExceptions	45
Section 8.4: Null-conditional Operator can be used with Extension Method	45
Chapter 9: nameof Operator	47
Section 9.1: Basic usage: Printing a variable name	47
Section 9.2: Raising PropertyChanged event	47
Section 9.3: Argument Checking and Guard Clauses	48
Section 9.4: Strongly typed MVC action links	48
Section 9.5: Handling PropertyChanged events	49
Section 9.6: Applied to a generic type parameter	49
Section 9.7: Printing a parameter name	50
Section 9.8: Applied to qualified identifiers	50
Chapter 10: Verbatim Strings	51
Section 10.1: Interpolated Verbatim Strings	51
Section 10.2: Escaping Double Quotes	51
Section 10.3: Verbatim strings instruct the compiler to not use character escapes	51
Section 10.4: Multiline Strings	52
Chapter 11: Common String Operations	53
Section 11.1: Formatting a string	53
Section 11.2: Correctly reversing a string	53
Section 11.3: Padding a string to a fixed length	54
Section 11.4: Getting x characters from the right side of a string	55
Section 11.5: Checking for empty String using String.IsNullOrEmpty() and String.IsNullOrWhiteSpace()	56
Section 11.6: Trimming Unwanted Characters Off the Start and/or End of Strings	57
Section 11.7: Convert Decimal Number to Binary,Octal and Hexadecimal Format	57
Section 11.8: Construct a string from Array	57
Section 11.9: Formatting using ToString	58
Section 11.10: Splitting a String by another string	59

Section 11.11: Splitting a String by specific character	59
Section 11.12: Getting Substrings of a given string	59
Section 11.13: Determine whether a string begins with a given sequence	59
Section 11.14: Getting a char at specific index and enumerating the string	59
Section 11.15: Joining an array of strings into a new one	60
Section 11.16: Replacing a string within a string	60
Section 11.17: Changing the case of characters within a String	60
Section 11.18: Concatenate an array of strings into a single string	61
Section 11.19: String Concatenation	61
Chapter 12: String.Format	62
Section 12.1: Since C# 6.0	62
Section 12.2: Places where String.Format is 'embedded' in the framework	62
Section 12.3: Create a custom format provider	62
Section 12.4: Date Formatting	63
Section 12.5: Currency Formatting	64
Section 12.6: Using custom number format	65
Section 12.7: Align left/ right, pad with spaces	65
Section 12.8: Numeric formats	66
Section 12.9: ToString()	66
Section 12.10: Escaping curly brackets inside a String.Format() expression	67
Section 12.11: Relationship with ToString()	67
Chapter 13: String Concatenate	68
Section 13.1: + Operator	68
Section 13.2: Concatenate strings using System.Text.StringBuilder	68
Section 13.3: Concat string array elements using String.Join	68
Section 13.4: Concatenation of two strings using \$	69
Chapter 14: String Manipulation	70
Section 14.1: Replacing a string within a string	70
Section 14.2: Finding a string within a string	70
Section 14.3: Removing (Trimming) white-space from a string	70
Section 14.4: Splitting a string using a delimiter	71
Section 14.5: Concatenate an array of strings into a single string	71
Section 14.6: String Concatenation	71
Section 14.7: Changing the case of characters within a String	71
Chapter 15: String Interpolation	73
Section 15.1: Format dates in strings	73
Section 15.2: Padding the output	73
Section 15.3: Expressions	74
Section 15.4: Formatting numbers in strings	74
Section 15.5: Simple Usage	75
Chapter 16: String Escape Sequences	76
Section 16.1: Escaping special symbols in string literals	76
Section 16.2: Unicode character escape sequences	76
Section 16.3: Escaping special symbols in character literals	76
Section 16.4: Using escape sequences in identifiers	76
Section 16.5: Unrecognized escape sequences produce compile-time errors	77
Chapter 17: StringBuilder	78
Section 17.1: What a StringBuilder is and when to use one	78
Section 17.2: Use StringBuilder to create string from a large number of records	79

Chapter 18: Regex Parsing	80
Section 18.1: Single match	80
Section 18.2: Multiple matches	80
Chapter 19: DateTime Methods	81
Section 19.1: DateTime Formatting	81
Section 19.2: DateTime.AddDays(Double)	82
Section 19.3: DateTime.AddHours(Double)	82
Section 19.4: DateTime.Parse(String)	82
Section 19.5: DateTime.TryParse(String, DateTime)	82
Section 19.6: DateTime.AddMilliseconds(Double)	83
Section 19.7: DateTime.Compare(DateTime t1, DateTime t2)	83
Section 19.8: DateTime.DaysInMonth(Int32,Int32)	83
Section 19.9: DateTime.AddYears(Int32)	84
Section 19.10: Pure functions warning when dealing with DateTime	84
Section 19.11: DateTime.TryParseExact(String, String, IFormatProvider, DateTimeStyles, DateTime)	84
.....	84
Section 19.12: DateTime.Add(TimeSpan)	86
Section 19.13: Parse and TryParse with culture info	86
Section 19.14: DateTime as initializer in for-loop	87
Section 19.15: DateTime.ParseExact(String, String, IFormatProvider)	87
Section 19.16: DateTime ToString, ToShortDateString, ToLongDateString and ToString formatted	88
Section 19.17: Current Date	88
Chapter 20: Arrays	89
Section 20.1: Declaring an array	89
Section 20.2: Initializing an array filled with a repeated non-default value	89
Section 20.3: Copying arrays	90
Section 20.4: Comparing arrays for equality	90
Section 20.5: Multi-dimensional arrays	91
Section 20.6: Getting and setting array values	91
Section 20.7: Iterate over an array	91
Section 20.8: Creating an array of sequential numbers	92
Section 20.9: Jagged arrays	92
Section 20.10: Array covariance	94
Section 20.11: Arrays as IEnumerable<> instances	94
Section 20.12: Checking if one array contains another array	94
Chapter 21: O(n) Algorithm for circular rotation of an array	96
Section 21.1: Example of a generic method that rotates an array by a given shift	96
Chapter 22: Enum	98
Section 22.1: Enum basics	98
Section 22.2: Enum as flags	99
Section 22.3: Using << notation for flags	101
Section 22.4: Test flags-style enum values with bitwise logic	101
Section 22.5: Add and remove values from flagged enum	102
Section 22.6: Enum to string and back	102
Section 22.7: Enums can have unexpected values	103
Section 22.8: Default value for enum == ZERO	103
Section 22.9: Adding additional description information to an enum value	104
Section 22.10: Get all the members values of an enum	105
Section 22.11: Bitwise Manipulation using enums	105
Chapter 23: Tuples	106

[Click here to download full PDF material](#)