C# Basic

Type is a general term referring to a member of the set {class, interface, structure, enumeration, delegate}).

Unlike many other languages, in C# it is not possible to create global functions or global points of data. Rather, all data members and all methods must be contained within a type definition.

C# is a ***case-sensitive programming language***. therefore, Main is not the same as main. All C# keywords are lowercase (e.g., public, lock, class, dynamic), while namespaces, types, and member names begin (by convention) with an initial capital letter and have capitalized the first letter of any embedded words (e.g., Console.WriteLine, System.Data.SqlClient).

Every executable C# application must contain a class defining a **Main()** method, which is used to signify the entry point of the application. The class that defines the Main() method is termed the application object. The signature of Main() is adorned with the static keyword. In addition to that, Main() method has a single parameter, which happens to be an array of strings (string[] args). Although you are not currently bothering to process this array, this parameter may contain any number of incoming command-line arguments. Finally, this Main() method has been set up with a void return value, meaning you do not explicitly define a return value using the return keyword before exiting the method scope. The logic of **Program** is within Main(). Here, you make use of the Console class, which is defined within the System namespace. Among its set of members is the static WriteLine(), which, as you might assume, sends a text string and carriage return to the standard output. You also make a call to Console.ReadLine() to ensure the command prompt launched by the Visual Studio IDE remains visible during a debugging session until you press the Enter key. *(If you did not add this line, your application would terminate immediately during a debugging session and you could not read the output!).*

Variations on the Main() method:

```
// int return type, array of strings as the parameter.
static int Main(string[] args)
{
        // Must return a value before exiting!
        return 0;
}

// No return type, no parameters.
static void Main()
{
}

// int return type, no parameters.
static int Main()
{
        // Must return a value before exiting!
        return 0;
}
```

The Main() method may also be defined as public as opposed to private, which is assumed if you do not supply a specific access modifier. ***Visual studio automatically defines a program's Main() method as implicitly private.***

With the release of C# 7.1, the Main() method can now be asynchronous:

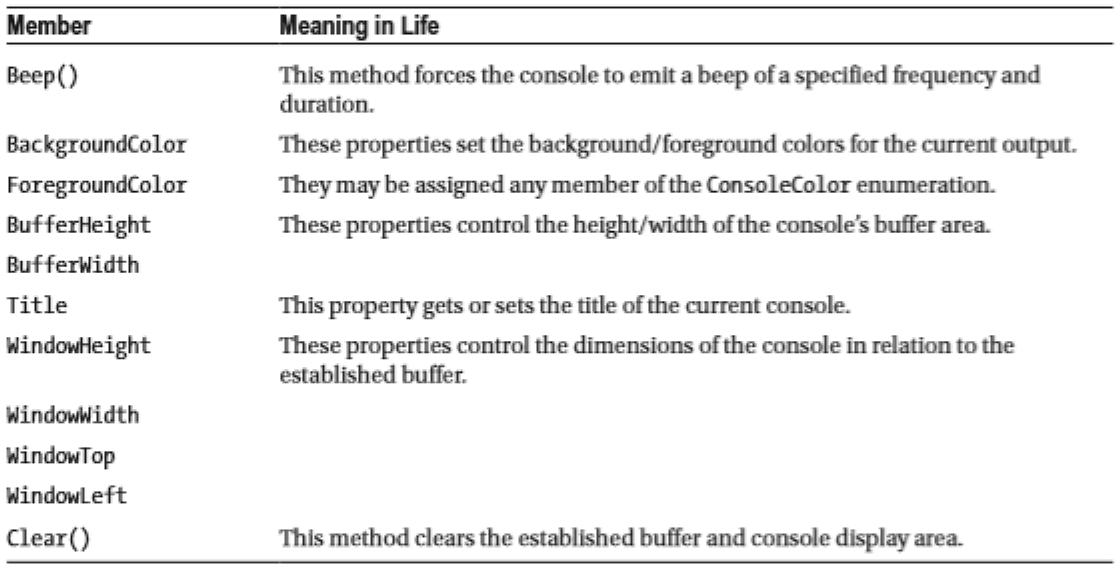
```
static Task Main()
static Task<int> Main()
static Task Main(string[])
static Task<int> Main(string[])
```

By convention, returning the value 0 indicates the program has terminated successfully, while another value (such as -1) represents an error condition (*be aware that the value 0 is automatically returned, even if you construct a Main() method prototyped to return void*).

On the Windows operating system, an application's return value is stored within a system environment variable named %ERRORLEVEL%. If you were to create an application that programmatically launches another executable you can obtain the value of %ERRORLEVEL% using the static System.Diagnostics.Process.ExitCode property.

System.Console Class

*The **Console class** encapsulates input, output, and error-stream manipulations for console-based applications.*



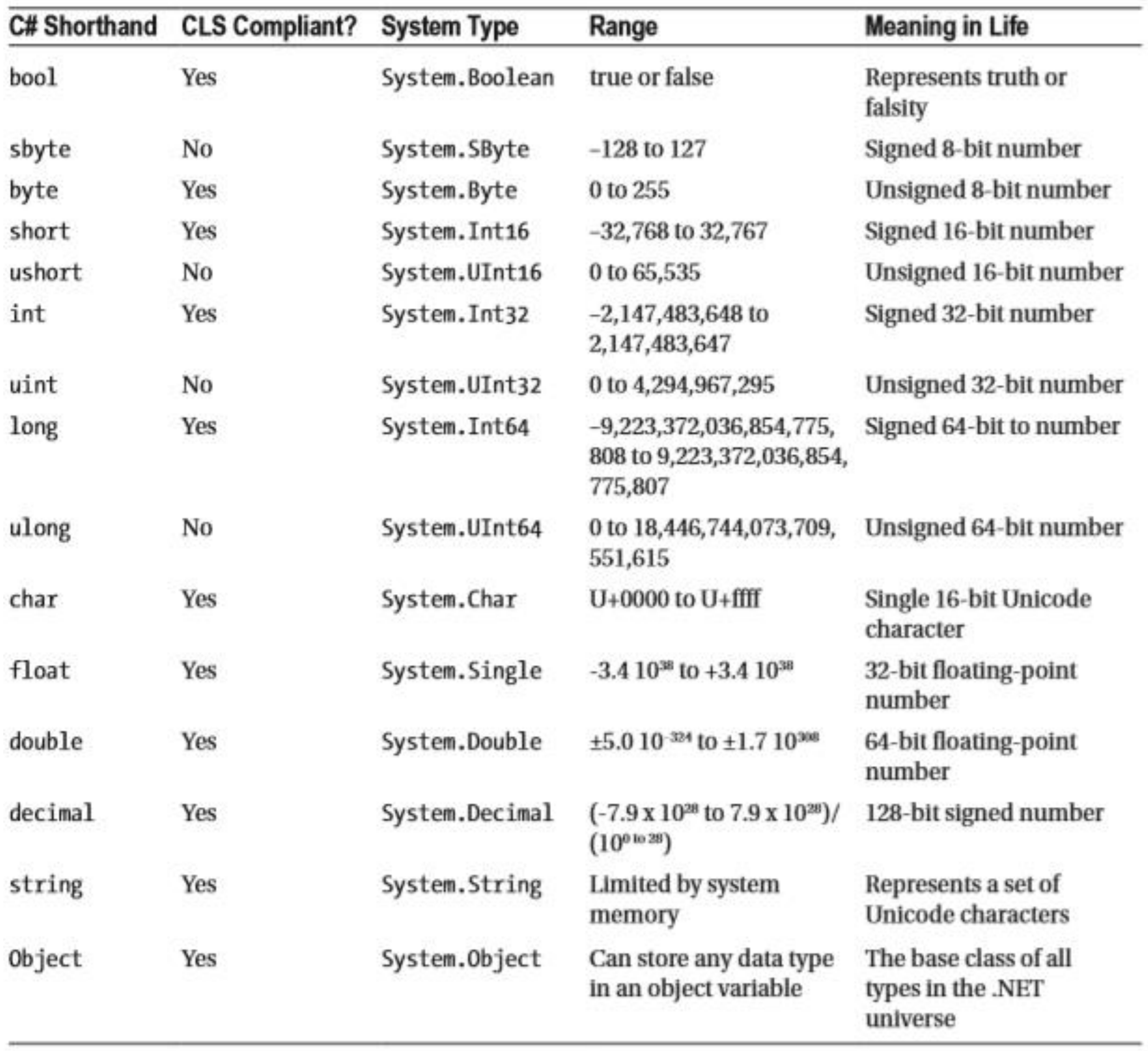
Member	Meaning in Life
Beep()	This method forces the console to emit a beep of a specified frequency and duration.
BackgroundColor	These properties set the background/foreground colors for the current output.
ForegroundColor	They may be assigned any member of the ConsoleColor enumeration.
BufferHeight	These properties control the height/width of the console's buffer area.
BufferWidth	
Title	This property gets or sets the title of the current console.
WindowHeight	These properties control the dimensions of the console in relation to the established buffer.
WindowWidth	
WindowTop	
WindowLeft	
Clear()	This method clears the established buffer and console display area.

string.Format() method

The string.Format() method returns a new string object, which is formatted according to the provided flags. After this point, you are free to use the textual data.

System Data Types and Corresponding C# Keywords

C# data type keywords are actually shorthand notations for full-blown types in the System namespace. The following table lists each system data type, its range, the corresponding C# keyword, and the type's compliance with the Common Language Specification (CLS).



C# Shorthand	CLS Compliant?	System Type	Range	Meaning in Life
bool	Yes	System.Boolean	true or false	Represents truth or falsity
sbyte	No	System.SByte	–128 to 127	Signed 8-bit number
byte	Yes	System.Byte	0 to 255	Unsigned 8-bit number
short	Yes	System.Int16	–32,768 to 32,767	Signed 16-bit number
ushort	No	System.UInt16	0 to 65,535	Unsigned 16-bit number
int	Yes	System.Int32	–2,147,483,648 to 2,147,483,647	Signed 32-bit number
uint	No	System.UInt32	0 to 4,294,967,295	Unsigned 32-bit number
long	Yes	System.Int64	–9,223,372,036,854,775, 808 to 9,223,372,036,854, 775,807	Signed 64-bit to number
ulong	No	System.UInt64	0 to 18,446,744,073,709, 551,615	Unsigned 64-bit number
char	Yes	System.Char	U+0000 to U+ffff	Single 16-bit Unicode character
float	Yes	System.Single	$-3.4\ 10^{38}$ to $+3.4\ 10^{38}$	32-bit floating-point number
double	Yes	System.Double	$\pm5.0\ 10^{-324}$ to $\pm1.7\ 10^{308}$	64-bit floating-point number
decimal	Yes	System.Decimal	$(-7.9 \times 10^{28}$ to $7.9 \times 10^{28})/(10^{0 \text{ to } 28})$	128-bit signed number
string	Yes	System.String	Limited by system memory	Represents a set of Unicode characters
Object	Yes	System.Object	Can store any data type in an object variable	The base class of all types in the .NET universe

Recall that if you expose non-CLs-compliant data from your programs, other .net languages might not be able to make use of it.

It is good practice to assign an initial value to your local data points at the time of declaration. You may do so on a single line or by separating the declaration and assignment into two code statements. It is also permissible to declare multiple variables of the same underlying type on a single line of code, as in the following three bool variables:

```
bool b1 = true, b2 = false, b3 = b1;
```

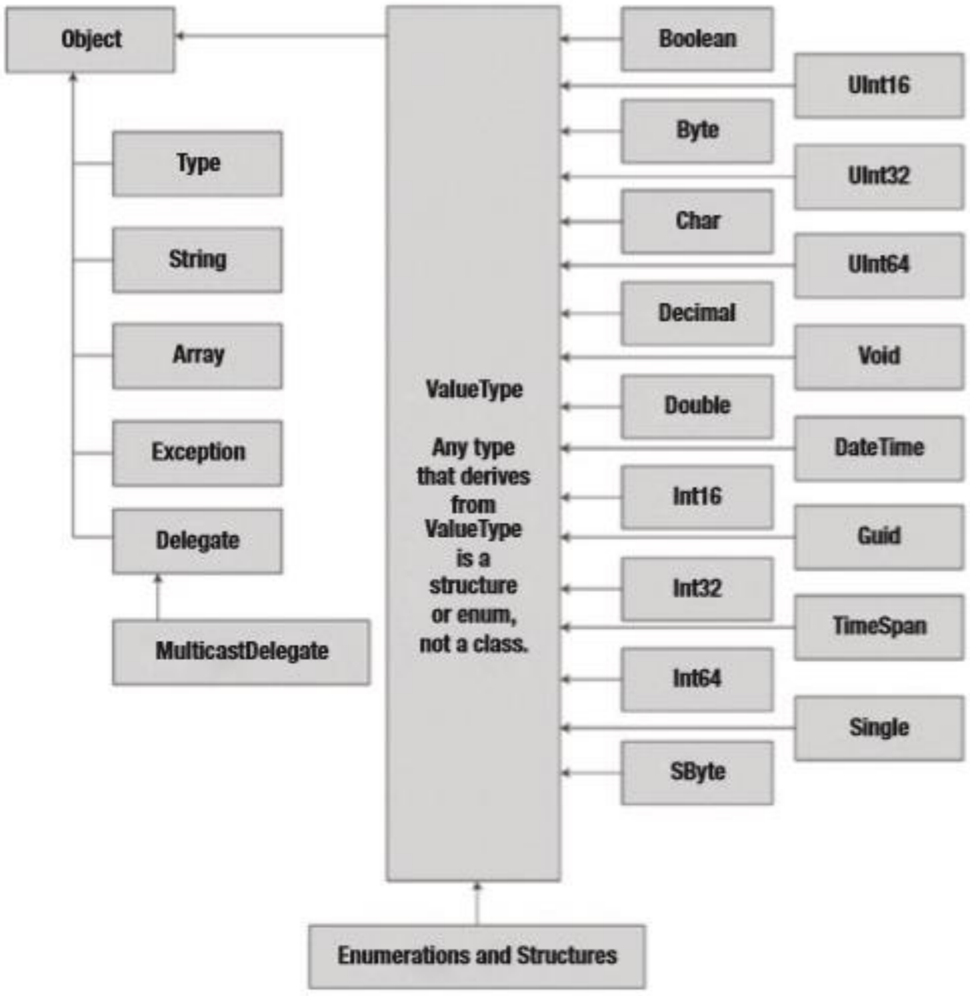
Default literal

The **default** literal is a feature in C# that *allows for assigning a variable the default value for its data type*. This works for standard data types as well as custom classes and generic types.

All intrinsic data types support what is known as a default constructor. This feature allows you to create a variable using the new keyword, which automatically sets the variable to its default value.

- ***bool variables are set to false.***
- ***Numeric data is set to 0 (or 0.0 in the case of floating-point data types).***
- ***char variables are set to a single empty character.***
- ***BigInteger variables are set to 0.***
- ***DateTime variables are set to 1/1/0001 12:00:00 AM.***
- ***Object references (including strings) are set to null.***

The Data Type Class Hierarchy



Each type ultimately derives from System.Object, which defines a set of methods common to all types in the .NET base class libraries. Also note that many ***numerical data types derive from a class named System.ValueType.*** Descendants of ValueType are automatically allocated on the **stack** and, therefore, have a predictable lifetime and are quite efficient. On the other hand, types that do not have System.ValueType in their inheritance chain are not allocated on the stack but on the garbage-collected **heap**.

The numerical types of .NET support **MaxValue** and **MinValue** properties that provide information regarding the range a given type can store. In addition to the MinValue/MaxValue properties, for example, the System.Double type allows you to obtain the values for *epsilon* and *infinity:*

```
Console.WriteLine (int.MaxValue);
Console.WriteLine (double.MinValue);
Console.WriteLine (double.Epsilon);
```

System.Boolean

- Represents the Boolean value true as a string.

```
public static readonly string TrueString;

  Console.WriteLine(bool.TrueString);
```

- Represents the Boolean value false as a string.

```
public static readonly string FalseString;

  Console.WriteLine(bool.FalseString);
```

These fields are read-only.

System.Char

The System.Char type provides you with a great deal of functionality beyond the ability to hold a single point of character data. You are able to determine whether a given character is numerical, alphabetical, a point of punctuation, or whatnot.

```
char character = 'A';
Console.WriteLine(char.IsDigit(character));
Console.WriteLine(char.IsLetter(character));
Console.WriteLine(char.IsWhiteSpace(character));
```

etc.

Parsing Values from String Data

The .NET data types provide the ability to generate a variable of their underlying type given a textual equivalent (e.g., parsing).

```
bool b = bool.Parse("True");
double d = double.Parse("99.884");
int i = int.Parse("8");
```

One issue with the preceding code is that an exception will be thrown if the string cannot be cleanly converted to the correct data type. For example, the following will fail at runtime:

```
bool b = bool.Parse("Hello");
```

The preferable solution is the **TryParse()** statement, which takes an out parameter and returns a bool if the parsing was successful. if a string can be converted to the requested datatype, the TryParse() method returns true and assigns the parsed value to the variable passed into the method. If the value cannot be parsed, the variable is assigned its default value, and the TryParse() method returns false.

```
string value = "Hello";
if (double.TryParse(value, out double d))
{
        Console.WriteLine("Value of d: {0}", d);
}
```

System.DateTime and System.TimeSpan

The DateTime type contains data that represents a specific date (month, day, year) and time value, both of which may be formatted in a variety of ways using the supplied members. The TimeSpan structure allows you to easily define and transform units of time using various members.

```
// This constructor takes (year, month, day).
DateTime dt = new DateTime(2015, 10, 17);
TimeSpan ts = new TimeSpan(4, 30, 0);
Console.WriteLine(ts);
Console.WriteLine(ts.Subtract(new TimeSpan(0, 15, 0)));
```

System.Numerics.dll Assembly

The System.Numerics namespace defines a structure named **BigInteger**. As its name implies, the *BigInteger data type can be used when you need to represent humongous numerical values, which are not constrained by a fixed upper or lower limit.* After you assign a value to a *BigInteger variable, you cannot change it, as the data is **immutable***. However, the BigInteger class defines a number of members that will return new BigInteger objects based on your data modifications. You can create a BigInteger variable using the new operator. Within the constructor, you can specify a numerical value, including floating-point data. However, when you define a literal whole number (such as 500), the runtime will default the data type to an int. Likewise, literal floating-point data (such as 55.333) will default to a double. If you do find the need to define a massive numerical value, your first step is to reference the System.Numerics.dll assembly into your project.

Digit Separators

Sometimes when assigning large numbers to a numeric variable, there are more digits than the eye can keep track of. C# 7 introduces the underscore (_) as a digit separator (for integer, long, decimal, or double data types). Here's an example of using the new digit separator:

```
Console.WriteLine(123_456);      // 123456
Console.WriteLine(123_456.1234F);      // 123456.1234f
```

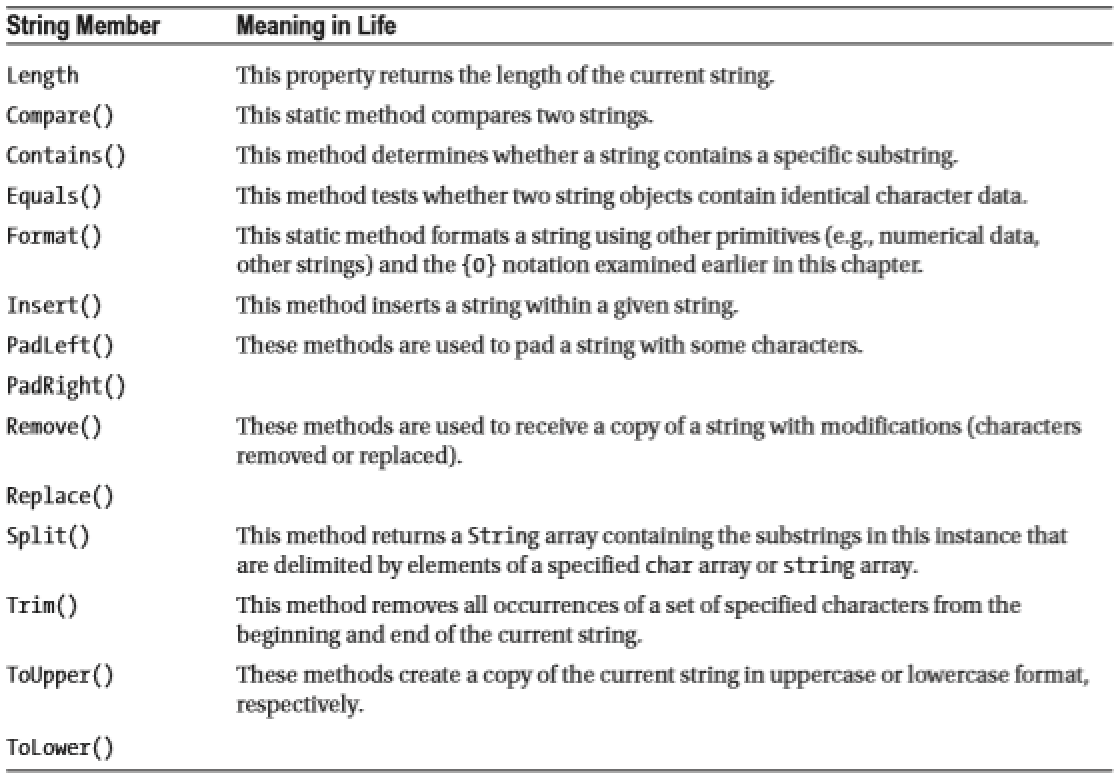
Binary Literals

C# introduces a new literal for binary values, for example, for creating bit masks. Now, binary numbers can be written as you would expect. Here's an example:

```
Console.WriteLine(0b0100_0000);   // 64
```

System.String

System.String provides a number of methods you would expect from such a utility class, including methods that return the length of the character data, find substrings within the current string, and convert to and from uppercase/lowercase.



String Member	Meaning in Life
`Length`	This property returns the length of the current string.
`Compare()`	This static method compares two strings.
`Contains()`	This method determines whether a string contains a specific substring.
`Equals()`	This method tests whether two string objects contain identical character data.
`Format()`	This static method formats a string using other primitives (e.g., numerical data, other strings) and the `{0}` notation examined earlier in this chapter.
`Insert()`	This method inserts a string within a given string.
`PadLeft()`	These methods are used to pad a string with some characters.
`PadRight()`	
`Remove()`	These methods are used to receive a copy of a string with modifications (characters removed or replaced).
`Replace()`	
`Split()`	This method returns a `String` array containing the substrings in this instance that are delimited by elements of a specified `char` array or `string` array.
`Trim()`	This method removes all occurrences of a set of specified characters from the beginning and end of the current string.
`ToUpper()`	These methods create a copy of the current string in uppercase or lowercase format, respectively.
`ToLower()`	

String variables can be connected to build larger strings via the C# + (as well as +=) operator. As you might know, this technique is formally termed **string concatenation**.

```
string s2 = "aaa";
string s3 = s2 + "123";
```

You might to know that *the C# + symbol is processed by the compiler to emit a call to the static String.Concat() method*. Given this, it is possible to perform string concatenation by calling String.Concat() directly.

Escape Characters

As in other C-based languages, C# string literals may contain various escape characters, which qualify how the character data should be printed to the output stream.

- \' Inserts a single quote into a string literal
- \" Inserts a double quote into a string literal
- \\ Inserts a backslash into a string literal. This can be quite helpful when defining file or network paths.
- \a Triggers a system alert (beep). For console programs, this can be an audio clue to the user.
- \n Inserts a new line (on Windows platforms).
- \r Inserts a carriage return.
- \t Inserts a horizontal tab into the string literal.

When you prefix a string literal with the @ symbol, you have created what is termed a **verbatim string**. Using verbatim strings, you disable the processing of a literal's escape characters and print out a string as is. This can be most useful when working with strings representing directory and network paths.

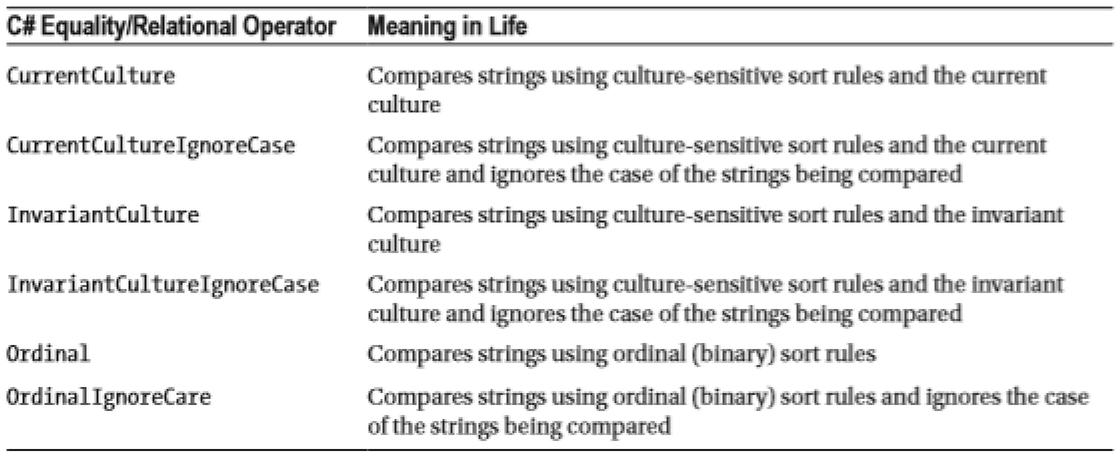
```
Console.WriteLine(@"C:\MyApp\bin\Debug");
```

Strings and Equality

As we know a reference type is an object allocated on the garbage-collected managed heap. *By default, when you perform a test for equality on reference types (via the C# == and != operators), you will be returned true if the references are pointing to the same object in memory.* However, even though the **string data type is indeed a reference type, the equality operators have been redefined to compare the values of string objects, not the object in memory to which they refer.**

The C# equality operators by default perform a case-sensitive, culture-insensitive, character-bycharacter equality test on string objects. Therefore, "Hello!" is not equal to "HELLO!", which is also different from "hello!". Also, keeping the connection between string and System.String in mind, notice that you are able to test for equality using the Equals() method of String as well as the baked-in equality operators.

As mentioned, the string equality operators (Compare(), Equals(), and ==) as well as the IndexOf() function are by default case-sensitive and culture-insensitive. This can cause a problem if your program doesn't care about case. One way to overcome this is to convert everything to uppercase or lowercase and then compare. This makes a copy of each string with all lowercase letters. It's probably not an issue in most cases but could be a performance hit with a significantly large string.



C# Equality/Relational Operator	Meaning in Life
`CurrentCulture`	Compares strings using culture-sensitive sort rules and the current culture
`CurrentCultureIgnoreCase`	Compares strings using culture-sensitive sort rules and the current culture and ignores the case of the strings being compared
`InvariantCulture`	Compares strings using culture-sensitive sort rules and the invariant culture
`InvariantCultureIgnoreCase`	Compares strings using culture-sensitive sort rules and the invariant culture and ignores the case of the strings being compared
`Ordinal`	Compares strings using ordinal (binary) sort rules
`OrdinalIgnoreCare`	Compares strings using ordinal (binary) sort rules and ignores the case of the strings being compared

Strings are Immutable

One of the interesting aspects of System.String is that after you assign a string object with its initial value, the character data cannot be changed. At first glance, this might seem like a flat-out lie, given that you are always reassigning strings to new values and because the System.String type defines a number of methods that appear to modify the character data in one way or another. However, if you look more closely at what is happening behind the scenes, you will notice the methods of the string type are, in fact, returning you a new string object in a modified format.

```
string s1 = "This is my string.";
Console.WriteLine(s1);                    //  This is my string
string upperString = s1.ToUpper();
Console.WriteLine(upperString);           //  THIS IS MY STRING
Console.WriteLine(s1);                    //  This is my string
```

In this example the original string object (s1) is not uppercased when calling ToUpper(). Rather, you are returned a copy of the string in a modified format.

If you are building an application that makes heavy use of frequently changing textual data (such as a word processing program), it would be a bad idea to represent the word processing data using string objects, as you will most certainly (and often indirectly) end up making unnecessary copies of string data.

System.Text.StringBuilder Type

What is unique about the **StringBuilder** is that when you call members of this type*, you are directly modifying the object's internal character data (making it more efficient), not obtaining a copy of the data in a modified format*. When you create an instance of the StringBuilder, you can supply the object's initial startup values via one of many constructors.

By default, a StringBuilder is only able to initially hold a string of 16 characters or fewer (but will expand automatically if necessary); however, this default starting value can be changed via an additional constructor argument. If you append more characters than the specified limit, the StringBuilder object will copy its data into a new instance and grow the buffer by the specified limit.

Implicitly and explicitly cast of types
Type conversions

Implicit cast

```
byte myByte = 0;
int myInt = 200;
myByte = myInt;
```

explicit cast

```
byte myByte = 0;
int myInt = 200000;
byte myByte2 = (byte)(myByte + myInt);
```

This code compiles, however, the result of the addition is completely incorrect. An explicit cast allows you to force the compiler to apply a narrowing conversion, even when doing so may result in a loss of data. So, tha answer will be 64.

Checked and Unchecked Keywords

If you are building an application where loss of data is always unacceptable, C# provides the checked and unchecked keywords to ensure data loss does not escape undetected.

```
byte myByte = 0;
int myInt = 200000;
byte sum = checked((byte)(myByte  + myInt));
Console.WriteLine(sum);
```

The value of sum has been wrapped within the scope of the **checked** keyword. Because the sum is greater than a byte, this triggers a runtime exception.

```
unchecked
{
        byte sum = (byte)(b1 + b2);
        Console.WriteLine(sum);
}
```

So, to summarize the C# checked and unchecked keywords, remember that the default behavior of the .NET runtime is to ignore arithmetic overflow/underflow. When you want to selectively handle discrete statements, make use of the checked keyword. If you want to trap overflow errors throughout your application, enable the /checked flag. Finally, the unchecked keyword can be used if you have a block of code where overflow is acceptable (and thus should not trigger a runtime exception).

implicitly typing of variables (var keyword)

```
int myInt = 0;
bool myBool = true;
string myString = "Time, marches on...";
```

C# language does provide for implicitly typing of local variables using the **var** keyword. The var keyword can be used in place of specifying a specific data type (such as int, bool, or string). When you do so, the compiler will automatically infer the underlying data type based on the initial value used to initialize the local data point.

```
var myInt = 0;
var myBool = true;
var myString = "Time, marches on...";
```

Strictly speaking, var is not a C# keyword. it is permissible to declare variables, parameters, and fields named var without compile-time errors. however, when the var token is used as a data type, it is contextually treated as a keyword by the compiler.

Restrictions on Implicitly Typed Variables

There are various restrictions regarding the use of the **var keyword**.

- ***implicit typing applies only to local variables in a method or property scope. It is illegal to use the var keyword to define return values, parameters, or field data of a custom type.***

- ***local variables declared with the var keyword must be assigned an initial value at the exact time of declaration and cannot be assigned the initial value of null. This restriction should make sense, given that the compiler cannot infer what sort of type in memory the variable would be pointing to based only on null.***

- ***It is permissible, however, to assign an inferred local variable to null after its initial assignment (provided it is a reference type).***

```
// OK, if SportsCar is a reference type!
var myCar = new SportsCar();
myCar = null;
```

- ***Furthermore, it is permissible to assign the value of an implicitly typed local variable to the value of other variables, implicitly typed or not.***

```
// Also OK!
var myInt = 0;
var anotherInt = myInt;
string myString = "Wake up!";
var myData = myString;
```

- ***Also, it is permissible to return an implicitly typed local variable to the caller, provided the method return type is the same underlying type as the var-defined data point.***

```
static int GetAnInt()
{
        var retVal = 9;
        return retVal;
}
```

Implicit Typed Data Is Strongly Typed Data

Rather, type inference keeps the strongly typed aspect of the C# language and affects only the declaration of variables at compile time. After that, the data point is treated as if it were declared with that type; assigning a value of a different type into that variable will result in a compile-time error.

In fact, it could be argued that the only time you would make use of the var keyword is when defining data returned from a LINQ query. Remember, if you know you need an int, just declare an int! Overuse of implicit typing (via the var keyword) is considered by most developers to be poor style in production code.