Contents

C# Keywords

9/5/2018 • 2 minutes to read • Edit Online

Keywords are predefined, reserved identifiers that have special meanings to the compiler. They cannot be used as identifiers in your program unless they include `@` as a prefix. For example, `@if` is a valid identifier, but `if` is not because `if` is a keyword.

The first table in this topic lists keywords that are reserved identifiers in any part of a C# program. The second table in this topic lists the contextual keywords in C#. Contextual keywords have special meaning only in a limited program context and can be used as identifiers outside that context. Generally, as new keywords are added to the C# language, they are added as contextual keywords in order to avoid breaking programs written in earlier versions.

abstract	as	base	bool
break	byte	case	catch
char	checked	class	const
continue	decimal	default	delegate
do	double	else	enum
event	explicit	extern	false
finally	fixed	float	for
foreach	goto	if	implicit
in	int	interface	internal
is	lock	long	namespace
new	null	object	operator
out	override	params	private
protected	public	readonly	ref
return	sbyte	sealed	short
sizeof	stackalloc	static	string
struct	switch	this	throw
true	try	typeof	uint
ulong	unchecked	unsafe	ushort

using	using static	virtual	void
volatile	while		

Contextual Keywords

A contextual keyword is used to provide a specific meaning in the code, but it is not a reserved word in C#. Some contextual keywords, such as `partial` and `where`, have special meanings in two or more contexts.

add	alias	ascending
async	await	by
descending	dynamic	equals
from	get	global
group	into	join
let	nameof	on
orderby	partial (type)	partial (method)
remove	select	set
value	var	when (filter condition)
where (generic type constraint)	where (query clause)	yield

See Also

- C# Reference
- C# Programming Guide

Types (C# Reference)

8/25/2018 • 2 minutes to read • Edit Online

The C# typing system contains the following categories:

- Value types
- Reference types
- Pointer types

Variables that are value types store data, and those that are reference types store references to the actual data. Reference types are also referred to as objects. Pointer types can be used only in unsafe mode.

It is possible to convert a value type to a reference type, and back again to a value type, by using boxing and unboxing. With the exception of a boxed value type, you cannot convert a reference type to a value type.

This section also introduces void.

Value types are also nullable, which means they can store an additional non-value state. For more information, see Nullable Types.

See Also

- C# Reference
- C# Programming Guide
- C# Keywords
- Reference Tables for Types
- Casting and Type Conversions
- Types

Value Types (C# Reference)

8/25/2018 • 2 minutes to read • Edit Online

The value types consist of two main categories:

- Structs
- Enumerations

Structs fall into these categories:

- Numeric types
  - Integral types
  - Floating-point types
- bool
- User defined structs.

Main Features of Value Types

Variables that are based on value types directly contain values. Assigning one value type variable to another copies the contained value. This differs from the assignment of reference type variables, which copies a reference to the object but not the object itself.

All value types are derived implicitly from the System.ValueType.

Unlike with reference types, you cannot derive a new type from a value type. However, like reference types, structs can implement interfaces.

Unlike reference types, a value type cannot contain the `null` value. However, the nullable types feature does allow for value types to be assigned to `null`.

Each value type has an implicit default constructor that initializes the default value of that type. For information about default values of value types, see Default Values Table.

Main Features of Simple Types

All of the simple types -- those integral to the C# language -- are aliases of the .NET Framework System types. For example, int is an alias of System.Int32. For a complete list of aliases, see Built-In Types Table.

Constant expressions, whose operands are all simple type constants, are evaluated at compilation time.

Simple types can be initialized by using literals. For example, 'A' is a literal of the type `char` and 2001 is a literal of the type `int`.

Initializing Value Types

Local variables in C# must be initialized before they are used. For example, you might declare a local variable without initialization as in the following example:

```
int myInt;
```

You cannot use it before you initialize it. You can initialize it using the following statement:

```
myInt = new int();  // Invoke default constructor for int type.
```

This statement is equivalent to the following statement:

```
myInt = 0;         // Assign an initial value, 0 in this example.
```

You can, of course, have the declaration and the initialization in the same statement as in the following examples:

```
int myInt = new int();
```

–or–

```
int myInt = 0;
```

Using the new operator calls the default constructor of the specific type and assigns the default value to the variable. In the preceding example, the default constructor assigned the value `0` to `myInt`. For more information about values assigned by calling default constructors, see Default Values Table.

With user-defined types, use new to invoke the default constructor. For example, the following statement invokes the default constructor of the `Point` struct:

```
Point p = new Point(); // Invoke default constructor for the struct.
```

After this call, the struct is considered to be definitely assigned; that is, all its members are initialized to their default values.

For more information about the new operator, see new.

For information about formatting the output of numeric types, see Formatting Numeric Results Table.

See Also

- C# Reference
- C# Programming Guide
- C# Keywords
- Types
- Reference Tables for Types
- Reference Types
- Nullable types

bool (C# Reference)

8/25/2018 • 2 minutes to read • Edit Online

The `bool` keyword is an alias of System.Boolean. It is used to declare variables to store the Boolean values, true and false.

> **NOTE**
>
> If you require a Boolean variable that can also have a value of `null`, use `bool?`. For more information, see Nullable Types.

Literals

You can assign a Boolean value to a `bool` variable. You can also assign an expression that evaluates to `bool` to a `bool` variable.

```
public class BoolTest
{
    static void Main()
    {
        bool b = true;

        // WriteLine automatically converts the value of b to text.
        Console.WriteLine(b);

        int days = DateTime.Now.DayOfYear;


        // Assign the result of a boolean expression to b.
        b = (days % 2 == 0);

        // Branch depending on whether b is true or false.
        if (b)
        {
            Console.WriteLine("days is an even number");
        }
        else
        {
            Console.WriteLine("days is an odd number");
        }
    }
}
/* Output:
  True
  days is an <even/odd> number
*/
```

The default value of a `bool` variable is `false`. The default value of a `bool?` variable is `null`.

Conversions

In C++, a value of type `bool` can be converted to a value of type `int`; in other words, `false` is equivalent to zero and `true` is equivalent to nonzero values. In C#, there is no conversion between the `bool` type and other types. For example, the following `if` statement is invalid in C#:

```
int x = 123;

// if (x)   // Error: "Cannot implicitly convert type 'int' to 'bool'"
{
    Console.Write("The value of x is nonzero.");
}
```

To test a variable of the type `int`, you have to explicitly compare it to a value, such as zero, as follows:

```

if (x != 0)   // The C# way
{
    Console.Write("The value of x is nonzero.");
}
```

Example

In this example, you enter a character from the keyboard and the program checks if the input character is a letter. If it is a letter, it checks if it is lowercase or uppercase. These checks are performed with the IsLetter, and IsLower, both of which return the `bool` type:

```
public class BoolKeyTest
{
    static void Main()
    {
        Console.Write("Enter a character: ");
        char c = (char)Console.Read();
        if (Char.IsLetter(c))
        {
            if (Char.IsLower(c))
            {
                Console.WriteLine("The character is lowercase.");
            }
            else
            {
                Console.WriteLine("The character is uppercase.");
            }
        }
        else
        {
            Console.WriteLine("Not an alphabetic character.");
        }
    }
}
/* Sample Output:
    Enter a character: X
    The character is uppercase.

    Enter a character: x
    The character is lowercase.

    Enter a character: 2
    The character is not an alphabetic character.
 */
```

C# language specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- C# Reference
- C# Programming Guide
- C# Keywords
- Integral Types Table
- Built-In Types Table
- Implicit Numeric Conversions Table
- Explicit Numeric Conversions Table

byte (C# Reference)

8/25/2018 • 2 minutes to read • Edit Online

`byte` denotes an integral type that stores values as indicated in the following table.

TYPE	RANGE	SIZE	.NET TYPE
`byte`	0 to 255	Unsigned 8-bit integer	System.Byte

Literals

You can declare and initialize a `byte` variable by assigning a decimal literal, a hexadecimal literal, or (starting with C# 7.0) a binary literal to it. If the integer literal is outside the range of `byte` (that is, if it is less than Byte.MinValue or greater than Byte.MaxValue), a compilation error occurs.

In the following example, integers equal to 201 that are represented as decimal, hexadecimal, and binary literals are implicitly converted from int to `byte` values.

```
byte byteValue1 = 201;
Console.WriteLine(byteValue1);

byte byteValue2 = 0x00C9;
Console.WriteLine(byteValue2);

byte byteValue3 = 0b1100_1001;
Console.WriteLine(byteValue3);
// The example displays the following output:
//          201
//          201
//          201
```

> **NOTE**
>
> You use the prefix `0x` or `0X` to denote a hexadecimal literal and the prefix `0b` or `0B` to denote a binary literal. Decimal literals have no prefix.

Starting with C# 7.0, a couple of features have been added to enhance readability.

- C# 7.0 allows the usage of the underscore character, `_`, as a digit separator.
- C# 7.2 allows `_` to be used as a digit separator for a binary or hexadecimal literal, after the prefix. A decimal literal isn't permitted to have a leading underscore.

Some examples are shown below.

```
byte byteValue4 = 0b1100_1001;
Console.WriteLine(byteValue4);

byte byteValue5 = 0b_1100_1001;
Console.WriteLine(byteValue5);       // C# 7.2 onwards
// The example displays the following output:
//          201
//          201
```

Conversions

There is a predefined implicit conversion from `byte` to short, ushort, int, uint, long, ulong, float, double, or decimal.

You cannot implicitly convert non-literal numeric types of larger storage size to `byte`. For more information on the storage sizes of integral types, see Integral Types Table. Consider, for example, the following two `byte` variables `x` and `y`:

```
byte x = 10, y = 20;
```

The following assignment statement will produce a compilation error, because the arithmetic expression on the right-hand side of the assignment operator evaluates to `int` by default.

```
// Error: conversion from int to byte:
byte z = x + y;
```

To fix this problem, use a cast:

```
// OK: explicit conversion:
byte z = (byte)(x + y);
```

It is possible though, to use the following statements where the destination variable has the same storage size or a larger storage size:

```
int x = 10, y = 20;
int m = x + y;
long n = x + y;
```

Also, there is no implicit conversion from floating-point types to `byte`. For example, the following statement generates a compiler error unless an explicit cast is used:

```
// Error: no implicit conversion from double:
byte x = 3.0;
// OK: explicit conversion:
byte y = (byte)3.0;
```

When calling overloaded methods, a cast must be used. Consider, for example, the following overloaded methods that use `byte` and int parameters:

```
public static void SampleMethod(int i) {}
public static void SampleMethod(byte b) {}
```

Using the `byte` cast guarantees that the correct type is called, for example:

```
// Calling the method with the int parameter:
SampleMethod(5);
// Calling the method with the byte parameter:
SampleMethod((byte)5);
```

For information on arithmetic expressions with mixed floating-point types and integral types, see float and double.

For more information on implicit numeric conversion rules, see the Implicit Numeric Conversions Table.

C# Language Specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See Also

- Byte
- C# Reference
- C# Programming Guide
- C# Keywords
- Integral Types Table
- Built-In Types Table
- Implicit Numeric Conversions Table
- Explicit Numeric Conversions Table

char (C# Reference)

8/25/2018 • 2 minutes to read • Edit Online

The `char` keyword is used to declare an instance of the System.Char structure that the .NET Framework uses to represent a Unicode character. The value of a `Char` object is a 16-bit numeric (ordinal) value.

Unicode characters are used to represent most of the written languages throughout the world.

TYPE	RANGE	SIZE	.NET TYPE
`char`	U+0000 to U+FFFF	Unicode 16-bit character	System.Char

Literals

Constants of the `char` type can be written as character literals, hexadecimal escape sequence, or Unicode representation. You can also cast the integral character codes. In the following example four `char` variables are initialized with the same character `X`:

```
char[] chars = new char[4];

chars[0] = 'X';        // Character literal
chars[1] = '\x0058';   // Hexadecimal
chars[2] = (char)88;   // Cast from integral type
chars[3] = '\u0058';   // Unicode

foreach (char c in chars)
{
    Console.Write(c + " ");
}
// Output: X X X X
```

Conversions

A `char` can be implicitly converted to ushort, int, uint, long, ulong, float, double, or decimal. However, there are no implicit conversions from other types to the `char` type.

The System.Char type provides several static methods for working with `char` values.

C# language specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- Char
- C# Reference
- C# Programming Guide
- C# Keywords
- Integral Types Table

- Built-In Types Table
- Implicit Numeric Conversions Table
- Explicit Numeric Conversions Table
- Nullable Types
- Strings

decimal (C# Reference)

8/25/2018 • 2 minutes to read • Edit Online

The `decimal` keyword indicates a 128-bit data type. Compared to other floating-point types, the `decimal` type has more precision and a smaller range, which makes it appropriate for financial and monetary calculations. The approximate range and precision for the `decimal` type are shown in the following table.

TYPE	APPROXIMATE RANGE	PRECISION	.NET TYPE
`decimal`	$\pm 1.0 \times 10^{-28}$ to $\pm 7.9228 \times 10^{28}$	28-29 significant digits	System.Decimal

The default value of a `decimal` is 0m.

Literals

If you want a numeric real literal to be treated as `decimal`, use the suffix m or M, for example:

```
decimal myMoney = 300.5m;
```

Without the suffix m, the number is treated as a double and generates a compiler error.

Conversions

The integral types are implicitly converted to `decimal` and the result evaluates to `decimal`. Therefore you can initialize a decimal variable using an integer literal, without the suffix, as follows:

```
decimal myMoney = 300;
```

There is no implicit conversion between other floating-point types and the `decimal` type; therefore, a cast must be used to convert between these two types. For example:

```
decimal myMoney = 99.9m;
double x = (double)myMoney;
myMoney = (decimal)x;
```

You can also mix `decimal` and numeric integral types in the same expression. However, mixing `decimal` and other floating-point types without a cast causes a compilation error.

For more information about implicit numeric conversions, see Implicit Numeric Conversions Table.

For more information about explicit numeric conversions, see Explicit Numeric Conversions Table.

Formatting decimal output

You can format the results by using the `String.Format` method, or through the Console.Write method, which calls `String.Format()`. The currency format is specified by using the standard currency format string "C" or "c," as shown in the second example later in this article. For more information about the `String.Format` method, see String.Format.

Example

The following example causes a compiler error by trying to add double and `decimal` variables.

```
decimal dec = 0m;
double dub = 9;
// The following line causes an error that reads "Operator '+' cannot be applied to
// operands of type 'double' and 'decimal'"
Console.WriteLine(dec + dub);

// You can fix the error by using explicit casting of either operand.
Console.WriteLine(dec + (decimal)dub);
Console.WriteLine((double)dec + dub);
```

The result is the following error:

`Operator '+' cannot be applied to operands of type 'double' and 'decimal'`

In this example, a `decimal` and an int are mixed in the same expression. The result evaluates to the `decimal` type.

```
public class TestDecimal
{
    static void Main()
    {
        decimal d = 9.1m;
        int y = 3;
        Console.WriteLine(d + y);   // Result converted to decimal
    }
}
// Output: 12.1
```

Example

In this example, the output is formatted by using the currency format string. Notice that `x` is rounded because the decimal places exceed $0.99. The variable `y`, which represents the maximum exact digits, is displayed exactly in the correct format.

```
public class TestDecimalFormat
{
    static void Main()
    {
        decimal x = 0.999m;
        decimal y = 9999999999999999999999999999m;
        Console.WriteLine("My amount = {0:C}", x);
        Console.WriteLine("Your amount = {0:C}", y);
    }
}
/* Output:
    My amount = $1.00
    Your amount = $9,999,999,999,999,999,999,999,999,999.00
*/
```

C# language specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- Decimal
- C# Reference
- C# Programming Guide
- C# Keywords
- Integral Types Table
- Built-In Types Table
- Implicit Numeric Conversions Table
- Explicit Numeric Conversions Table
- Standard Numeric Format Strings

double (C# Reference)

8/25/2018 • 2 minutes to read • Edit Online

The `double` keyword signifies a simple type that stores 64-bit floating-point values. The following table shows the precision and approximate range for the `double` type.

TYPE	APPROXIMATE RANGE	PRECISION	.NET TYPE
`double`	$\pm 5.0 \times 10^{-324}$ to $\pm 1.7 \times 10^{308}$	15-16 digits	System.Double

Literals

By default, a real numeric literal on the right side of the assignment operator is treated as `double`. However, if you want an integer number to be treated as `double`, use the suffix d or D, for example:

```
double x = 3D;
```

Conversions

You can mix numeric integral types and floating-point types in an expression. In this case, the integral types are converted to floating-point types. The evaluation of the expression is performed according to the following rules:

- If one of the floating-point types is `double`, the expression evaluates to `double`, or to bool in relational comparisons and comparisons for equality.
- If there is no `double` type in the expression, it evaluates to float, or to bool in relational comparisons and comparisons for equality.

A floating-point expression can contain the following sets of values:

- Positive and negative zero.
- Positive and negative infinity.
- Not-a-Number value (NaN).
- The finite set of nonzero values.

For more information about these values, see IEEE Standard for Binary Floating-Point Arithmetic, available on the IEEE Web site.

Example

In the following example, an int, a short, a float, and a `double` are added together giving a `double` result.

```
// Mixing types in expressions
class MixedTypes
{
    static void Main()
    {
        int x = 3;
        float y = 4.5f;
        short z = 5;
        double w = 1.7E+3;
        // Result of the 2nd argument is a double:
        Console.WriteLine("The sum is {0}", x + y + z + w);
    }
}
// Output: The sum is 1712.5
```

C# language specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See Also

- C# Reference
- C# Programming Guide
- C# Keywords
- Default Values Table
- Built-In Types Table
- Floating-Point Types Table
- Implicit Numeric Conversions Table
- Explicit Numeric Conversions Table

enum (C# Reference)

8/25/2018 • 4 minutes to read • Edit Online

The `enum` keyword is used to declare an enumeration, a distinct type that consists of a set of named constants called the enumerator list.

Usually it is best to define an enum directly within a namespace so that all classes in the namespace can access it with equal convenience. However, an enum can also be nested within a class or struct.

By default, the first enumerator has the value 0, and the value of each successive enumerator is increased by 1. For example, in the following enumeration, `Sat` is `0`, `Sun` is `1`, `Mon` is `2`, and so forth.

```
enum Day {Sat, Sun, Mon, Tue, Wed, Thu, Fri};
```

Enumerators can use initializers to override the default values, as shown in the following example.

```
enum Day {Sat=1, Sun, Mon, Tue, Wed, Thu, Fri};
```

In this enumeration, the sequence of elements is forced to start from `1` instead of `0`. However, including a constant that has the value of 0 is recommended. For more information, see Enumeration Types.

Every enumeration type has an underlying type, which can be any integral type except char. The default underlying type of enumeration elements is int. To declare an enum of another integral type, such as byte, use a colon after the identifier followed by the type, as shown in the following example.

```
enum Day : byte {Sat=1, Sun, Mon, Tue, Wed, Thu, Fri};
```

The approved types for an enum are byte, sbyte, short, ushort, int, uint, long, or ulong.

A variable of type `Day` can be assigned any value in the range of the underlying type; the values are not limited to the named constants.

The default value of an `enum E` is the value produced by the expression `(E)0`.

> **NOTE**
> An enumerator cannot contain white space in its name.

The underlying type specifies how much storage is allocated for each enumerator. However, an explicit cast is necessary to convert from `enum` type to an integral type. For example, the following statement assigns the enumerator `Sun` to a variable of the type int by using a cast to convert from `enum` to `int`.

```
int x = (int)Day.Sun;
```

When you apply System.FlagsAttribute to an enumeration that contains elements that can be combined with a bitwise `OR` operation, the attribute affects the behavior of the `enum` when it is used with some tools. You can notice these changes when you use tools such as the Console class methods and the Expression Evaluator. (See the third example.)

Robust programming

Just as with any constant, all references to the individual values of an enum are converted to numeric literals at compile time. This can create potential versioning issues as described in Constants.

Assigning additional values to new versions of enums, or changing the values of the enum members in a new version, can cause problems for dependent source code. Enum values often are used in switch statements. If additional elements have been added to the `enum` type, the default section of the switch statement can be selected unexpectedly.

If other developers use your code, you should provide guidelines about how their code should react if new elements are added to any `enum` types.

Example

In the following example, an enumeration, `Day`, is declared. Two enumerators are explicitly converted to integer and assigned to integer variables.

```
public class EnumTest
{
    enum Day { Sun, Mon, Tue, Wed, Thu, Fri, Sat };

    static void Main()
    {
        int x = (int)Day.Sun;
        int y = (int)Day.Fri;
        Console.WriteLine("Sun = {0}", x);
        Console.WriteLine("Fri = {0}", y);
    }
}
/* Output:
   Sun = 0
   Fri = 5
*/
```

Example

In the following example, the base-type option is used to declare an `enum` whose members are of type `long`. Notice that even though the underlying type of the enumeration is `long`, the enumeration members still must be explicitly converted to type `long` by using a cast.

```
public class EnumTest2
{
    enum Range : long { Max = 2147483648L, Min = 255L };
    static void Main()
    {
        long x = (long)Range.Max;
        long y = (long)Range.Min;
        Console.WriteLine("Max = {0}", x);
        Console.WriteLine("Min = {0}", y);
    }
}
/* Output:
   Max = 2147483648
   Min = 255
*/
```

Example

The following code example illustrates the use and effect of the System.FlagsAttribute attribute on an `enum` declaration.

```
// Add the attribute Flags or FlagsAttribute.
[Flags]
public enum CarOptions
{
    // The flag for SunRoof is 0001.
    SunRoof = 0x01,
    // The flag for Spoiler is 0010.
    Spoiler = 0x02,
    // The flag for FogLights is 0100.
    FogLights = 0x04,
    // The flag for TintedWindows is 1000.
    TintedWindows = 0x08,
}

class FlagTest
{
    static void Main()
    {
        // The bitwise OR of 0001 and 0100 is 0101.
        CarOptions options = CarOptions.SunRoof | CarOptions.FogLights;

        // Because the Flags attribute is specified, Console.WriteLine displays
        // the name of each enum element that corresponds to a flag that has
        // the value 1 in variable options.
        Console.WriteLine(options);
        // The integer value of 0101 is 5.
        Console.WriteLine((int)options);
    }
}
/* Output:
   SunRoof, FogLights
   5
*/
```

Comments

If you remove `Flags`, the example displays the following values:

`5`

`5`

C# language specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- C# Reference
- Enumeration Types
- C# Keywords
- Integral Types Table
- Built-In Types Table
- Implicit Numeric Conversions Table

- Explicit Numeric Conversions Table
- Enum Naming Conventions

float (C# Reference)

8/25/2018 • 2 minutes to read • Edit Online

The `float` keyword signifies a simple type that stores 32-bit floating-point values. The following table shows the precision and approximate range for the `float` type.

TYPE	APPROXIMATE RANGE	PRECISION	.NET TYPE
`float`	$\pm 1.5 \times 10^{-45}$ to $\pm 3.4 \times 10^{38}$	7 digits	System.Single

Literals

By default, a real numeric literal on the right side of the assignment operator is treated as double. Therefore, to initialize a float variable, use the suffix `f` or `F`, as in the following example:

```
float x = 3.5F;
```

If you do not use the suffix in the previous declaration, you will get a compilation error because you are trying to store a double value into a `float` variable.

Conversions

You can mix numeric integral types and floating-point types in an expression. In this case, the integral types are converted to floating-point types. The evaluation of the expression is performed according to the following rules:

- If one of the floating-point types is double, the expression evaluates to double, or to bool in relational comparisons or comparisons for equality.
- If there is no double type in the expression, the expression evaluates to `float`, or to bool in relational comparisons or comparisons for equality.

A floating-point expression can contain the following sets of values:

- Positive and negative zero
- Positive and negative infinity
- Not-a-Number value (NaN)
- The finite set of nonzero values

For more information about these values, see IEEE Standard for Binary Floating-Point Arithmetic, available on the IEEE website.

Example

In the following example, an int, a short, and a `float` are included in a mathematical expression giving a `float` result. (Remember that `float` is an alias for the System.Single type.) Notice that there is no double in the expression.

```
class FloatTest
{
    static void Main()
    {
        int x = 3;
        float y = 4.5f;
        short z = 5;
        var result = x * y / z;
        Console.WriteLine("The result is {0}", result);
        Type type = result.GetType();
        Console.WriteLine("result is of type {0}", type.ToString());
    }
}
/* Output:
  The result is 2.7
  result is of type System.Single //'float' is alias for 'Single'
 */
```

C# language specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- Single
- C# Reference
- C# Programming Guide
- Casting and Type Conversions
- C# Keywords
- Integral Types Table
- Built-In Types Table
- Implicit Numeric Conversions Table
- Explicit Numeric Conversions Table

int (C# Reference)

8/25/2018 • 2 minutes to read • Edit Online

`int` denotes an integral type that stores values according to the size and range shown in the following table.

TYPE	RANGE	SIZE	.NET TYPE
`int`	-2,147,483,648 to 2,147,483,647	Signed 32-bit integer	System.Int32

Literals

You can declare and initialize an `int` variable by assigning a decimal literal, a hexadecimal literal, or (starting with C# 7.0) a binary literal to it. If the integer literal is outside the range of `int` (that is, if it is less than Int32.MinValue or greater than Int32.MaxValue), a compilation error occurs.

In the following example, integers equal to 90,946 that are represented as decimal, hexadecimal, and binary literals are assigned to `int` values.

```
int intValue1 = 90946;
Console.WriteLine(intValue1);
int intValue2 = 0x16342;
Console.WriteLine(intValue2);

int intValue3 = 0b0001_0110_0011_0100_0010;
Console.WriteLine(intValue3);
// The example displays the following output:
//          90946
//          90946
//          90946
```

NOTE

You use the prefix `0x` or `0X` to denote a hexadecimal literal and the prefix `0b` or `0B` to denote a binary literal. Decimal literals have no prefix.

Starting with C# 7.0, a couple of features have been added to enhance readability.

- C# 7.0 allows the usage of the underscore character, `_`, as a digit separator.
- C# 7.2 allows `_` to be used as a digit separator for a binary or hexadecimal literal, after the prefix. A decimal literal isn't permitted to have a leading underscore.

Some examples are shown below.

```
int intValue1 = 90_946;
Console.WriteLine(intValue1);

int intValue2 = 0x0001_6342;
Console.WriteLine(intValue2);

int intValue3 = 0b0001_0110_0011_0100_0010;
Console.WriteLine(intValue3);

int intValue4 = 0x_0001_6342;       // C# 7.2 onwards
Console.WriteLine(intValue4);

int intValue5 = 0b_0001_0110_0011_0100_0010;       // C# 7.2 onwards
Console.WriteLine(intValue5);
// The example displays the following output:
//          90946
//          90946
//          90946
//          90946
//          90946
```

Integer literals can also include a suffix that denotes the type, although there is no suffix that denotes the `int` type. If an integer literal has no suffix, its type is the first of the following types in which its value can be represented:

1. `int`
2. uint
3. long
4. ulong

In these examples, the literal 90946 is of type `int`.

Conversions

There is a predefined implicit conversion from `int` to long, float, double, or decimal. For example:

```
// '123' is an int, so an implicit conversion takes place here:
float f = 123;
```

There is a predefined implicit conversion from sbyte, byte, short, ushort, or char to `int`. For example, the following assignment statement will produce a compilation error without a cast:

```
long aLong = 22;
int i1 = aLong;       // Error: no implicit conversion from long.
int i2 = (int)aLong;  // OK: explicit conversion.
```

Notice also that there is no implicit conversion from floating-point types to `int`. For example, the following statement generates a compiler error unless an explicit cast is used:

```
int x = 3.0;         // Error: no implicit conversion from double.
int y = (int)3.0;    // OK: explicit conversion.
```

For more information on arithmetic expressions with mixed floating-point types and integral types, see float and double.

C# Language Specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See Also

- Int32
- C# Reference
- C# Programming Guide
- C# Keywords
- Integral Types Table
- Built-In Types Table
- Implicit Numeric Conversions Table
- Explicit Numeric Conversions Table

long (C# Reference)

8/25/2018 • 3 minutes to read • Edit Online

`long` denotes an integral type that stores values according to the size and range shown in the following table.

TYPE	RANGE	SIZE	.NET TYPE
`long`	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	Signed 64-bit integer	System.Int64

Literals

You can declare and initialize a `long` variable by assigning a decimal literal, a hexadecimal literal, or (starting with C# 7.0) a binary literal to it.

In the following example, integers equal to 4,294,967,296 that are represented as decimal, hexadecimal, and binary literals are assigned to `long` values.

```
long longValue1 = 4294967296;
Console.WriteLine(longValue1);

long longValue2 = 0x100000000;
Console.WriteLine(longValue2);

long longValue3 = 0b1_0000_0000_0000_0000_0000_0000_0000_0000;
Console.WriteLine(longValue3);
// The example displays the following output:
//          4294967296
//          4294967296
//          4294967296
```

NOTE

You use the prefix `0x` or `0X` to denote a hexadecimal literal and the prefix `0b` or `0B` to denote a binary literal. Decimal literals have no prefix.

Starting with C# 7.0, a couple of features have been added to enhance readability.

- C# 7.0 allows the usage of the underscore character, `_`, as a digit separator.
- C# 7.2 allows `_` to be used as a digit separator for a binary or hexadecimal literal, after the prefix. A decimal literal isn't permitted to have a leading underscore.

Some examples are shown below.

```
long longValue1 = 4_294_967_296;
Console.WriteLine(longValue1);

long longValue2 = 0x1_0000_0000;
Console.WriteLine(longValue2);

long longValue3 = 0b1_0000_0000_0000_0000_0000_0000_0000_0000;
Console.WriteLine(longValue3);

long longValue4 = 0x_1_0000_0000;       // C# 7.2 onwards
Console.WriteLine(longValue4);

long longValue5 = 0b_1_0000_0000_0000_0000_0000_0000_0000_0000;       // C# 7.2 onwards
Console.WriteLine(longValue5);
// The example displays the following output:
//          4294967296
//          4294967296
//          4294967296
//          4294967296
//          4294967296
```

Integer literals can also include a suffix that denotes the type. The suffix `L` denotes a `long`. The following example uses the `L` suffix to denote a long integer:

```
long value = 4294967296L;
```

> **NOTE**
> You can also use the lowercase letter "l" as a suffix. However, this generates a compiler warning because the letter "l" is easily confused with the digit "1." Use "L" for clarity.

When you use the suffix `L`, the type of the literal integer is determined to be either `long` or ulong, depending on its size. In this case, it is `long` because it less than the range of ulong.

A common use of the suffix is to call overloaded methods. For example, the following overloaded methods have parameters of type `long` and int:

```
public static void SampleMethod(int i) {}
public static void SampleMethod(long l) {}
```

The `L` suffix guarantees that the correct overload is called:

```
SampleMethod(5);    // Calls the method with the int parameter
SampleMethod(5L);   // Calls the method with the long parameter
```

If an integer literal has no suffix, its type is the first of the following types in which its value can be represented:

1. int
2. uint
3. `long`
4. ulong

The literal 4294967296 in the previous examples is of type `long`, because it exceeds the range of uint (see Integral Types Table for the storage sizes of integral types).

If you use the `long` type with other integral types in the same expression, the expression is evaluated as `long` (or bool in the case of relational or Boolean expressions). For example, the following expression evaluates as `long`:

```
898L + 88
```

For information on arithmetic expressions with mixed floating-point types and integral types, see float and double.

Conversions

There is a predefined implicit conversion from `long` to float, double, or decimal. Otherwise a cast must be used. For example, the following statement will produce a compilation error without an explicit cast:

```
int x = 8L;        // Error: no implicit conversion from long to int
int y = (int)8L;   // OK: explicit conversion to int
```

There is a predefined implicit conversion from sbyte, byte, short, ushort, int, uint, or char to `long`.

Notice also that there is no implicit conversion from floating-point types to `long`. For example, the following statement generates a compiler error unless an explicit cast is used:

```
long x = 3.0;         // Error: no implicit conversion from double
long y = (long)3.0;   // OK: explicit conversion
```

C# Language Specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See Also

- Int64
- C# Reference
- C# Programming Guide
- C# Keywords
- Integral Types Table
- Built-In Types Table
- Implicit Numeric Conversions Table
- Explicit Numeric Conversions Table

sbyte (C# Reference)

8/25/2018 • 3 minutes to read • Edit Online

`sbyte` denotes an integral type that stores values according to the size and range shown in the following table.

TYPE	RANGE	SIZE	.NET TYPE
`sbyte`	-128 to 127	Signed 8-bit integer	System.SByte

Literals

You can declare and initialize an `sbyte` variable by assigning a decimal literal, a hexadecimal literal, or (starting with C# 7.0) a binary literal to it.

In the following example, integers equal to -102 that are represented as decimal, hexadecimal, and binary literals are converted from int to `sbyte` values.

```
sbyte sbyteValue1 = -102;
Console.WriteLine(sbyteValue1);

unchecked {
   sbyte sbyteValue2 = (sbyte)0x9A;
   Console.WriteLine(sbyteValue2);

   sbyte sbyteValue3 = (sbyte)0b1001_1010;
   Console.WriteLine(sbyteValue3);
}
// The example displays the following output:
//          -102
//          -102
//          -102
```

NOTE

You use the prefix `0x` or `0X` to denote a hexadecimal literal and the prefix `0b` or `0B` to denote a binary literal. Decimal literals have no prefix.

Starting with C# 7.0, a couple of features have been added to enhance readability.

- C# 7.0 allows the usage of the underscore character, `_`, as a digit separator.
- C# 7.2 allows `_` to be used as a digit separator for a binary or hexadecimal literal, after the prefix. A decimal literal isn't permitted to have a leading underscore.

Some examples are shown below.

```
unchecked {
   sbyte sbyteValue4 = (sbyte)0b1001_1010;
   Console.WriteLine(sbyteValue4);

   sbyte sbyteValue5 = (sbyte)0b_1001_1010;       // C# 7.2 onwards
   Console.WriteLine(sbyteValue5);
}
// The example displays the following output:
//          -102
//          -102
```

If the integer literal is outside the range of `sbyte` (that is, if it is less than SByte.MinValue or greater than SByte.MaxValue, a compilation error occurs. When an integer literal has no suffix, its type is the first of these types in which its value can be represented: int, uint, long, ulong. This means that, in this example, the numeric literals `0x9A` and `0b10011010` are interpreted as 32-bit signed integers with a value of 156, which exceeds SByte.MaxValue. Because of this, the casting operator is needed, and the assignment must occur in an unchecked context.

Compiler overload resolution

A cast must be used when calling overloaded methods. Consider, for example, the following overloaded methods that use `sbyte` and int parameters:

```
public static void SampleMethod(int i) {}
public static void SampleMethod(sbyte b) {}
```

Using the `sbyte` cast guarantees that the correct type is called, for example:

```
// Calling the method with the int parameter:
SampleMethod(5);
// Calling the method with the sbyte parameter:
SampleMethod((sbyte)5);
```

Conversions

There is a predefined implicit conversion from `sbyte` to short, int, long, float, double, or decimal.

You cannot implicitly convert nonliteral numeric types of larger storage size to `sbyte` (see Integral Types Table for the storage sizes of integral types). Consider, for example, the following two `sbyte` variables `x` and `y`:

```
sbyte x = 10, y = 20;
```

The following assignment statement will produce a compilation error, because the arithmetic expression on the right side of the assignment operator evaluates to int by default.

```
sbyte z = x + y;   // Error: conversion from int to sbyte
```

To fix this problem, cast the expression as in the following example:

```
sbyte z = (sbyte)(x + y);   // OK: explicit conversion
```

It is possible though to use the following statements, where the destination variable has the same storage size or a larger storage size:

```
sbyte x = 10, y = 20;
int m = x + y;
long n = x + y;
```

Notice also that there is no implicit conversion from floating-point types to `sbyte`. For example, the following statement generates a compiler error unless an explicit cast is used:

```
sbyte x = 3.0;         // Error: no implicit conversion from double
sbyte y = (sbyte)3.0;  // OK: explicit conversion
```

For information about arithmetic expressions with mixed floating-point types and integral types, see float and double.

For more information about implicit numeric conversion rules, see the Implicit Numeric Conversions Table.

C# Language Specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See Also

- SByte
- C# Reference
- C# Programming Guide
- C# Keywords
- Integral Types Table
- Built-In Types Table
- Implicit Numeric Conversions Table
- Explicit Numeric Conversions Table

short (C# Reference)

8/25/2018 • 2 minutes to read • Edit Online

`short` denotes an integral data type that stores values according to the size and range shown in the following table.

TYPE	RANGE	SIZE	.NET TYPE
`short`	-32,768 to 32,767	Signed 16-bit integer	System.Int16

Literals

You can declare and initialize a `short` variable by assigning a decimal literal, a hexadecimal literal, or (starting with C# 7.0) a binary literal to it. If the integer literal is outside the range of `short` (that is, if it is less than Int16.MinValue or greater than Int16.MaxValue), a compilation error occurs.

In the following example, integers equal to 1,034 that are represented as decimal, hexadecimal, and binary literals are implicitly converted from int to `short` values.

```
short shortValue1 = 1034;
Console.WriteLine(shortValue1);

short shortValue2 = 0x040A;
Console.WriteLine(shortValue2);

short shortValue3 = 0b0100_00001010;
Console.WriteLine(shortValue3);
// The example displays the following output:
//          1034
//          1034
//          1034
```

NOTE

You use the prefix `0x` or `0X` to denote a hexadecimal literal and the prefix `0b` or `0B` to denote a binary literal. Decimal literals have no prefix.

Starting with C# 7.0, a couple of features have been added to enhance readability.

- C# 7.0 allows the usage of the underscore character, `_`, as a digit separator.
- C# 7.2 allows `_` to be used as a digit separator for a binary or hexadecimal literal, after the prefix. A decimal literal isn't permitted to have a leading underscore.

Some examples are shown below.

```
short shortValue1 = 1_034;
Console.WriteLine(shortValue1);

short shortValue2 = 0b00000100_00001010;
Console.WriteLine(shortValue2);

short shortValue3 = 0b_00000100_00001010;       // C# 7.2 onwards
Console.WriteLine(shortValue3);
// The example displays the following output:
//          1034
//          1034
//          1034
```

Compiler overload resolution

A cast must be used when calling overloaded methods. Consider, for example, the following overloaded methods that use `short` and int parameters:

```
public static void SampleMethod(int i) {}
public static void SampleMethod(short s) {}
```

Using the `short` cast guarantees that the correct type is called, for example:

```
SampleMethod(5);         // Calling the method with the int parameter
SampleMethod((short)5);  // Calling the method with the short parameter
```

Conversions

There is a predefined implicit conversion from `short` to int, long, float, double, or decimal.

You cannot implicitly convert nonliteral numeric types of larger storage size to `short` (see Integral Types Table for the storage sizes of integral types). Consider, for example, the following two `short` variables `x` and `y`:

```
short x = 5, y = 12;
```

The following assignment statement produces a compilation error because the arithmetic expression on the right-hand side of the assignment operator evaluates to int by default.

```
short z  = x + y;        // Compiler error CS0266: no conversion from int to short
```

To fix this problem, use a cast:

```
short z  = (short)(x + y);   // Explicit conversion
```

It is also possible to use the following statements, where the destination variable has the same storage size or a larger storage size:

```
int m = x + y;
long n = x + y;
```

There is no implicit conversion from floating-point types to `short`. For example, the following statement generates a compiler error unless an explicit cast is used:

```
short x = 3.0;          // Error: no implicit conversion from double
short y = (short)3.0;   // OK: explicit conversion
```

For information on arithmetic expressions with mixed floating-point types and integral types, see float and double.

For more information on implicit numeric conversion rules, see the Implicit Numeric Conversions Table.

C# Language Specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See Also

- Int16
- C# Reference
- C# Programming Guide
- C# Keywords
- Integral Types Table
- Built-In Types Table
- Implicit Numeric Conversions Table
- Explicit Numeric Conversions Table

struct (C# Reference)

8/25/2018 • 2 minutes to read • Edit Online

A `struct` type is a value type that is typically used to encapsulate small groups of related variables, such as the coordinates of a rectangle or the characteristics of an item in an inventory. The following example shows a simple struct declaration:

```
public struct Book
{
    public decimal price;
    public string title;
    public string author;
}
```

Remarks

Structs can also contain constructors, constants, fields, methods, properties, indexers, operators, events, and nested types, although if several such members are required, you should consider making your type a class instead.

For examples, see Using Structs.

Structs can implement an interface but they cannot inherit from another struct. For that reason, struct members cannot be declared as `protected`.

For more information, see Structs.

Examples

For examples and more information, see Using Structs.

C# Language Specification

For examples, see Using Structs.

See Also

- C# Reference
- C# Programming Guide
- C# Keywords
- Default Values Table
- Built-In Types Table
- Types
- Value Types
- class
- interface
- Classes and Structs

uint (C# Reference)

8/25/2018 • 3 minutes to read • Edit Online

The `uint` keyword signifies an integral type that stores values according to the size and range shown in the following table.

TYPE	RANGE	SIZE	.NET TYPE
`uint`	0 to 4,294,967,295	Unsigned 32-bit integer	System.UInt32

Note The `uint` type is not CLS-compliant. Use `int` whenever possible.

Literals

You can declare and initialize a `uint` variable by assigning a decimal literal, a hexadecimal literal, or (starting with C# 7.0) a binary literal to it. If the integer literal is outside the range of `uint` (that is, if it is less than UInt32.MinValue or greater than UInt32.MaxValue), a compilation error occurs.

In the following example, integers equal to 3,000,000,000 that are represented as decimal, hexadecimal, and binary literals are assigned to `uint` values.

```
uint uintValue1 = 3000000000;
Console.WriteLine(uintValue1);

uint uintValue2 = 0xB2D05E00;
Console.WriteLine(uintValue2);

uint uintValue3 = 0b1011_0010_1101_0000_0101_1110_0000_0000;
Console.WriteLine(uintValue3);
// The example displays the following output:
//          3000000000
//          3000000000
//          3000000000
```

> **NOTE**
>
> You use the prefix `0x` or `0X` to denote a hexadecimal literal and the prefix `0b` or `0B` to denote a binary literal. Decimal literals have no prefix.

Starting with C# 7.0, a couple of features have been added to enhance readability.

- C# 7.0 allows the usage of the underscore character, `_`, as a digit separator.
- C# 7.2 allows `_` to be used as a digit separator for a binary or hexadecimal literal, after the prefix. A decimal literal isn't permitted to have a leading underscore.

Some examples are shown below.

```
uint uintValue1 = 3_000_000_000;
Console.WriteLine(uintValue1);

uint uintValue2 = 0xB2D0_5E00;
Console.WriteLine(uintValue2);

uint uintValue3 = 0b1011_0010_1101_0000_0101_1110_0000_0000;
Console.WriteLine(uintValue3);

uint uintValue4 = 0x_B2D0_5E00;       // C# 7.2 onwards
Console.WriteLine(uintValue4);

uint uintValue5 = 0b_1011_0010_1101_0000_0101_1110_0000_0000;       // C# 7.2 onwards
Console.WriteLine(uintValue5);
// The example displays the following output:
//          3000000000
//          3000000000
//          3000000000
//          3000000000
//          3000000000
```

Integer literals can also include a suffix that denotes the type. The suffix `U` or 'u' denotes either a `uint` or a `ulong`, depending on the numeric value of the literal. The following example uses the `u` suffix to denote an unsigned integer of both types. Note that the first literal is a `uint` because its value is less than UInt32.MaxValue, while the second is a `ulong` because its value is greater than UInt32.MaxValue.

```
object value1 = 4000000000u;
Console.WriteLine($"{value1} ({4000000000u.GetType().Name})");
object value2 = 6000000000u;
Console.WriteLine($"{value2} ({6000000000u.GetType().Name})");
```

If an integer literal has no suffix, its type is the first of the following types in which its value can be represented:

1. int
2. `uint`
3. long
4. ulong

Conversions

There is a predefined implicit conversion from `uint` to long, ulong, float, double, or decimal. For example:

```
float myFloat = 4294967290;   // OK: implicit conversion to float
```

There is a predefined implicit conversion from byte, ushort, or char to `uint`. Otherwise you must use a cast. For example, the following assignment statement will produce a compilation error without a cast:

```
long aLong = 22;
// Error -- no implicit conversion from long:
uint uInt1 = aLong;
// OK -- explicit conversion:
uint uInt2 = (uint)aLong;
```

Notice also that there is no implicit conversion from floating-point types to `uint`. For example, the following statement generates a compiler error unless an explicit cast is used:

```
// Error -- no implicit conversion from double:
uint x = 3.0;
// OK -- explicit conversion:
uint y = (uint)3.0;
```

For information about arithmetic expressions with mixed floating-point types and integral types, see float and double.

For more information about implicit numeric conversion rules, see the Implicit Numeric Conversions Table.

C# Language Specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See Also

- UInt32
- C# Reference
- C# Programming Guide
- C# Keywords
- Integral Types Table
- Built-In Types Table
- Implicit Numeric Conversions Table
- Explicit Numeric Conversions Table

ulong (C# Reference)

8/25/2018 • 3 minutes to read • Edit Online

The `ulong` keyword denotes an integral type that stores values according to the size and range shown in the following table.

TYPE	RANGE	SIZE	.NET TYPE
`ulong`	0 to 18,446,744,073,709,551,615	Unsigned 64-bit integer	System.UInt64

Literals

You can declare and initialize a `ulong` variable by assigning a decimal literal, a hexadecimal literal, or (starting with C# 7.0) a binary literal to it. If the integer literal is outside the range of `ulong` (that is, if it is less than UInt64.MinValue or greater than UInt64.MaxValue), a compilation error occurs.

In the following example, integers equal to 7,934,076,125 that are represented as decimal, hexadecimal, and binary literals are assigned to `ulong` values.

```
ulong ulongValue1 = 7934076125;
Console.WriteLine(ulongValue1);

ulong ulongValue2 = 0x0001D8e864DD;
Console.WriteLine(ulongValue2);

ulong ulongValue3 = 0b0001_1101_1000_1110_1000_0110_0100_1101_1101;
Console.WriteLine(ulongValue3);
// The example displays the following output:
//          7934076125
//          7934076125
//          7934076125
```

NOTE

You use the prefix `0x` or `0X` to denote a hexadecimal literal and the prefix `0b` or `0B` to denote a binary literal. Decimal literals have no prefix.

Starting with C# 7.0, a couple of features have been added to enhance readability.

- C# 7.0 allows the usage of the underscore character, `_`, as a digit separator.
- C# 7.2 allows `_` to be used as a digit separator for a binary or hexadecimal literal, after the prefix. A decimal literal isn't permitted to have a leading underscore.

Some examples are shown below.

```
long longValue1 = 4_294_967_296;
Console.WriteLine(longValue1);

long longValue2 = 0x1_0000_0000;
Console.WriteLine(longValue2);

long longValue3 = 0b1_0000_0000_0000_0000_0000_0000_0000_0000;
Console.WriteLine(longValue3);

long longValue4 = 0x_1_0000_0000;       // C# 7.2 onwards
Console.WriteLine(longValue4);

long longValue5 = 0b_1_0000_0000_0000_0000_0000_0000_0000_0000;       // C# 7.2 onwards
Console.WriteLine(longValue5);
// The example displays the following output:
//          4294967296
//          4294967296
//          4294967296
//          4294967296
//          4294967296
```

Integer literals can also include a suffix that denotes the type. The suffix `UL` or `ul` unambiguously identifies a numeric literal as a `ulong` value. The `L` suffix denotes a `ulong` if the literal value exceeds Int64.MaxValue. And the `U` or `u` suffix denotes a `ulong` if the literal value exceeds UInt32.MaxValue. The following example uses the `ul` suffix to denote a long integer:

```
object value1 = 700000000000ul;
Console.WriteLine($"{value1} ({700000000000ul.GetType().Name})");
```

If an integer literal has no suffix, its type is the first of the following types in which its value can be represented:

1. int
2. uint
3. long
4. `ulong`

Compiler overload resolution

A common use of the suffix is with calling overloaded methods. Consider, for example, the following overloaded methods that use `ulong` and int parameters:

```
public static void SampleMethod(int i) {}
public static void SampleMethod(ulong l) {}
```

Using a suffix with the `ulong` parameter guarantees that the correct type is called, for example:

```
SampleMethod(5);    // Calling the method with the int parameter
SampleMethod(5UL);  // Calling the method with the ulong parameter
```

Conversions

There is a predefined implicit conversion from `ulong` to float, double, or decimal.

There is no implicit conversion from `ulong` to any integral type. For example, the following statement will produce a compilation error without an explicit cast:

```
long long1 = 8UL;   // Error: no implicit conversion from ulong
```

There is a predefined implicit conversion from byte, ushort, uint, or char to `ulong`.

Also, there is no implicit conversion from floating-point types to `ulong`. For example, the following statement generates a compiler error unless an explicit cast is used:

```
// Error -- no implicit conversion from double:
ulong x = 3.0;
// OK -- explicit conversion:
ulong y = (ulong)3.0;
```

For information on arithmetic expressions with mixed floating-point types and integral types, see float and double.

For more information on implicit numeric conversion rules, see the Implicit Numeric Conversions Table.

C# Language Specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See Also

- UInt64
- C# Reference
- C# Programming Guide
- C# Keywords
- Integral Types Table
- Built-In Types Table
- Implicit Numeric Conversions Table
- Explicit Numeric Conversions Table

ushort (C# Reference)

8/25/2018 • 3 minutes to read • Edit Online

The `ushort` keyword indicates an integral data type that stores values according to the size and range shown in the following table.

TYPE	RANGE	SIZE	.NET TYPE
`ushort`	0 to 65,535	Unsigned 16-bit integer	System.UInt16

Literals

You can declare and initialize a `ushort` variable by assigning a decimal literal, a hexadecimal literal, or (starting with C# 7.0) a binary literal to it. If the integer literal is outside the range of `ushort` (that is, if it is less than UInt16.MinValue or greater than UInt16.MaxValue), a compilation error occurs.

In the following example, integers equal to 65,034 that are represented as decimal, hexadecimal, and binary literals are implicitly converted from int to `ushort` values.

```
ushort ushortValue1 = 65034;
Console.WriteLine(ushortValue1);

ushort ushortValue2 = 0xFE0A;
Console.WriteLine(ushortValue2);

ushort ushortValue3 = 0b1111_1110_0000_1010;
Console.WriteLine(ushortValue3);
// The example displays the following output:
//          65034
//          65034
//          65034
```

NOTE

You use the prefix `0x` or `0X` to denote a hexadecimal literal and the prefix `0b` or `0B` to denote a binary literal. Decimal literals have no prefix.

Starting with C# 7.0, a couple of features have been added to enhance readability.

- C# 7.0 allows the usage of the underscore character, `_`, as a digit separator.
- C# 7.2 allows `_` to be used as a digit separator for a binary or hexadecimal literal, after the prefix. A decimal literal isn't permitted to have a leading underscore.

Some examples are shown below.

```
ushort ushortValue1 = 65_034;
Console.WriteLine(ushortValue1);

ushort ushortValue2 = 0b11111110_00001010;
Console.WriteLine(ushortValue2);

ushort ushortValue3 = 0b_11111110_00001010;     // C# 7.2 onwards
Console.WriteLine(ushortValue3);
// The example displays the following output:
//          65034
//          65034
//          65034
```

Compiler overload resolution

A cast must be used when you call overloaded methods. Consider, for example, the following overloaded methods that use `ushort` and int parameters:

```
public static void SampleMethod(int i) {}
public static void SampleMethod(ushort s) {}
```

Using the `ushort` cast guarantees that the correct type is called, for example:

```
// Calls the method with the int parameter:
SampleMethod(5);
// Calls the method with the ushort parameter:
SampleMethod((ushort)5);
```

Conversions

There is a predefined implicit conversion from `ushort` to int, uint, long, ulong, float, double, or decimal.

There is a predefined implicit conversion from byte or char to `ushort`. Otherwise a cast must be used to perform an explicit conversion. Consider, for example, the following two `ushort` variables `x` and `y`:

```
ushort x = 5, y = 12;
```

The following assignment statement will produce a compilation error, because the arithmetic expression on the right side of the assignment operator evaluates to `int` by default.

```
ushort z = x + y;   // Error: conversion from int to ushort
```

To fix this problem, use a cast:

```
ushort z = (ushort)(x + y);   // OK: explicit conversion
```

It is possible though to use the following statements, where the destination variable has the same storage size or a larger storage size:

```
int m = x + y;
long n = x + y;
```

Notice also that there is no implicit conversion from floating-point types to `ushort`. For example, the following statement generates a compiler error unless an explicit cast is used:

```
// Error -- no implicit conversion from double:
ushort x = 3.0;
// OK -- explicit conversion:
ushort y = (ushort)3.0;
```

For information about arithmetic expressions with mixed floating-point types and integral types, see float and double.

For more information about implicit numeric conversion rules, see the Implicit Numeric Conversions Table.

C# Language Specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See Also

- UInt16
- C# Reference
- C# Programming Guide
- C# Keywords
- Integral Types Table
- Built-In Types Table
- Implicit Numeric Conversions Table
- Explicit Numeric Conversions Table

Reference types (C# Reference)

8/29/2018 • 2 minutes to read • Edit Online

There are two kinds of types in C#: reference types and value types. Variables of reference types store references to their data (objects), while variables of value types directly contain their data. With reference types, two variables can reference the same object; therefore, operations on one variable can affect the object referenced by the other variable. With value types, each variable has its own copy of the data, and it is not possible for operations on one variable to affect the other (except in the case of in, ref and out parameter variables; see in, ref and out parameter modifier).

The following keywords are used to declare reference types:

- class
- interface
- delegate

C# also provides the following built-in reference types:

- dynamic
- object
- string

See also

- C# Reference
- C# Programming Guide
- C# Keywords
- Types
- Value Types

class (C# Reference)

8/25/2018 • 2 minutes to read • Edit Online

Classes are declared using the keyword `class`, as shown in the following example:

```
class TestClass
{
    // Methods, properties, fields, events, delegates
    // and nested classes go here.
}
```

Remarks

Only single inheritance is allowed in C#. In other words, a class can inherit implementation from one base class only. However, a class can implement more than one interface. The following table shows examples of class inheritance and interface implementation:

INHERITANCE	EXAMPLE
None	`class ClassA { }`
Single	`class DerivedClass: BaseClass { }`
None, implements two interfaces	`class ImplClass: IFace1, IFace2 { }`
Single, implements one interface	`class ImplDerivedClass: BaseClass, IFace1 { }`

Classes that you declare directly within a namespace, not nested within other classes, can be either public or internal. Classes are `internal` by default.

Class members, including nested classes, can be public, `protected internal`, protected, internal, private, or `private protected`. Members are private by default.

For more information, see Access Modifiers.

You can declare generic classes that have type parameters. For more information, see Generic Classes.

A class can contain declarations of the following members:

- Constructors
- Constants
- Fields
- Finalizers
- Methods
- Properties
- Indexers
- Operators

- Events
- Delegates
- Classes
- Interfaces
- Structs
- Enumerations

Example

The following example demonstrates declaring class fields, constructors, and methods. It also demonstrates object instantiation and printing instance data. In this example, two classes are declared. The first class, `Child`, contains two private fields (`name` and `age`), two public constructors and one public method. The second class, `StringTest`, is used to contain `Main`.

```
class Child
{
    private int age;
    private string name;

    // Default constructor:
    public Child()
    {
        name = "N/A";
    }

    // Constructor:
    public Child(string name, int age)
    {
        this.name = name;
        this.age = age;
    }

    // Printing method:
    public void PrintChild()
    {
        Console.WriteLine("{0}, {1} years old.", name, age);
    }
}

class StringTest
{
    static void Main()
    {
        // Create objects by using the new operator:
        Child child1 = new Child("Craig", 11);
        Child child2 = new Child("Sally", 10);

        // Create an object using the default constructor:
        Child child3 = new Child();

        // Display results:
        Console.Write("Child #1: ");
        child1.PrintChild();
        Console.Write("Child #2: ");
        child2.PrintChild();
        Console.Write("Child #3: ");
        child3.PrintChild();
    }
}
/* Output:
    Child #1: Craig, 11 years old.
    Child #2: Sally, 10 years old.
    Child #3: N/A, 0 years old.
*/
```

Comments

Notice that in the previous example the private fields (`name` and `age`) can only be accessed through the public method of the `Child` class. For example, you cannot print the child's name, from the `Main` method, using a statement like this:

```
Console.Write(child1.name);   // Error
```

Accessing private members of `Child` from `Main` would only be possible if `Main` were a member of the class.

Types declared inside a class without an access modifier default to `private`, so the data members in this example would still be `private` if the keyword were removed.

Finally, notice that for the object created using the default constructor (`child3`), the age field was initialized to zero by default.

C# language specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- C# Reference
- C# Programming Guide
- C# Keywords
- Reference Types

delegate (C# Reference)

8/25/2018 • 2 minutes to read • Edit Online

The declaration of a delegate type is similar to a method signature. It has a return value and any number of parameters of any type:

```
public delegate void TestDelegate(string message);
public delegate int TestDelegate(MyType m, long num);
```

A `delegate` is a reference type that can be used to encapsulate a named or an anonymous method. Delegates are similar to function pointers in C++; however, delegates are type-safe and secure. For applications of delegates, see Delegates and Generic Delegates.

Remarks

Delegates are the basis for Events.

A delegate can be instantiated by associating it either with a named or anonymous method. For more information, see Named Methods and Anonymous Methods.

The delegate must be instantiated with a method or lambda expression that has a compatible return type and input parameters. For more information on the degree of variance that is allowed in the method signature, see Variance in Delegates. For use with anonymous methods, the delegate and the code to be associated with it are declared together. Both ways of instantiating delegates are discussed in this section.

Example

```
// Declare delegate -- defines required signature:
delegate double MathAction(double num);

class DelegateTest
{
    // Regular method that matches signature:
    static double Double(double input)
    {
        return input * 2;
    }

    static void Main()
    {
        // Instantiate delegate with named method:
        MathAction ma = Double;

        // Invoke delegate ma:
        double multByTwo = ma(4.5);
        Console.WriteLine("multByTwo: {0}", multByTwo);

        // Instantiate delegate with anonymous method:
        MathAction ma2 = delegate(double input)
        {
            return input * input;
        };

        double square = ma2(5);
        Console.WriteLine("square: {0}", square);

        // Instantiate delegate with lambda expression
        MathAction ma3 = s => s * s * s;
        double cube = ma3(4.375);

        Console.WriteLine("cube: {0}", cube);
    }
    // Output:
    // multByTwo: 9
    // square: 25
    // cube: 83.740234375
}
```

C# language specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- C# Reference
- C# Programming Guide
- C# Keywords
- Reference Types
- Delegates
- Events
- Delegates with Named vs. Anonymous Methods
- Anonymous Methods

dynamic (C# Reference)

9/7/2018 • 3 minutes to read • Edit Online

The `dynamic` type enables the operations in which it occurs to bypass compile-time type checking. Instead, these operations are resolved at run time. The `dynamic` type simplifies access to COM APIs such as the Office Automation APIs, and also to dynamic APIs such as IronPython libraries, and to the HTML Document Object Model (DOM).

Type `dynamic` behaves like type `object` in most circumstances. However, operations that contain expressions of type `dynamic` are not resolved or type checked by the compiler. The compiler packages together information about the operation, and that information is later used to evaluate the operation at run time. As part of the process, variables of type `dynamic` are compiled into variables of type `object`. Therefore, type `dynamic` exists only at compile time, not at run time.

The following example contrasts a variable of type `dynamic` to a variable of type `object`. To verify the type of each variable at compile time, place the mouse pointer over `dyn` or `obj` in the `WriteLine` statements. IntelliSense shows **dynamic** for `dyn` and **object** for `obj`.

```
class Program
{
    static void Main(string[] args)
    {
        dynamic dyn = 1;
        object obj = 1;

        // Rest the mouse pointer over dyn and obj to see their
        // types at compile time.
        System.Console.WriteLine(dyn.GetType());
        System.Console.WriteLine(obj.GetType());
    }
}
```

The `WriteLine` statements display the run-time types of `dyn` and `obj`. At that point, both have the same type, integer. The following output is produced:

`System.Int32`

`System.Int32`

To see the difference between `dyn` and `obj` at compile time, add the following two lines between the declarations and the `WriteLine` statements in the previous example.

```
dyn = dyn + 3;
obj = obj + 3;
```

A compiler error is reported for the attempted addition of an integer and an object in expression `obj + 3`. However, no error is reported for `dyn + 3`. The expression that contains `dyn` is not checked at compile time because the type of `dyn` is `dynamic`.

Context

The `dynamic` keyword can appear directly or as a component of a constructed type in the following situations:

- In declarations, as the type of a property, field, indexer, parameter, return value, local variable, or type constraint. The following class definition uses `dynamic` in several different declarations.

```
class ExampleClass
{
    // A dynamic field.
    static dynamic field;

    // A dynamic property.
    dynamic prop { get; set; }

    // A dynamic return type and a dynamic parameter type.
    public dynamic exampleMethod(dynamic d)
    {
        // A dynamic local variable.
        dynamic local = "Local variable";
        int two = 2;

        if (d is int)
        {
            return local;
        }
        else
        {
            return two;
        }
    }
}
```

- In explicit type conversions, as the target type of a conversion.

```
static void convertToDynamic()
{
    dynamic d;
    int i = 20;
    d = (dynamic)i;
    Console.WriteLine(d);

    string s = "Example string.";
    d = (dynamic)s;
    Console.WriteLine(d);

    DateTime dt = DateTime.Today;
    d = (dynamic)dt;
    Console.WriteLine(d);

}
// Results:
// 20
// Example string.
// 7/25/2018 12:00:00 AM
```

- In any context where types serve as values, such as on the right side of an `is` operator or an `as` operator, or as the argument to `typeof` as part of a constructed type. For example, `dynamic` can be used in the following expressions.

```csharp
int i = 8;
dynamic d;
// With the is operator.
// The dynamic type behaves like object. The following
// expression returns true unless someVar has the value null.
if (someVar is dynamic) { }

// With the as operator.
d = i as dynamic;

// With typeof, as part of a constructed type.
Console.WriteLine(typeof(List<dynamic>));

// The following statement causes a compiler error.
//Console.WriteLine(typeof(dynamic));
```

Example

The following example uses `dynamic` in several declarations. The `Main` method also contrasts compile-time type checking with run-time type checking.

```
using System;

namespace DynamicExamples
{
    class Program
    {
        static void Main(string[] args)
        {
            ExampleClass ec = new ExampleClass();
            Console.WriteLine(ec.exampleMethod(10));
            Console.WriteLine(ec.exampleMethod("value"));

            // The following line causes a compiler error because exampleMethod
            // takes only one argument.
            //Console.WriteLine(ec.exampleMethod(10, 4));

            dynamic dynamic_ec = new ExampleClass();
            Console.WriteLine(dynamic_ec.exampleMethod(10));

            // Because dynamic_ec is dynamic, the following call to exampleMethod
            // with two arguments does not produce an error at compile time.
            // However, itdoes cause a run-time error.
            //Console.WriteLine(dynamic_ec.exampleMethod(10, 4));
        }
    }

    class ExampleClass
    {
        static dynamic field;
        dynamic prop { get; set; }

        public dynamic exampleMethod(dynamic d)
        {
            dynamic local = "Local variable";
            int two = 2;

            if (d is int)
            {
                return local;
            }
            else
            {
                return two;
            }
        }
    }
}
// Results:
// Local variable
// 2
// Local variable
```

For more information and examples, see Using Type dynamic.

See also

- System.Dynamic.ExpandoObject
- System.Dynamic.DynamicObject
- Using Type dynamic
- object
- is
- as
- typeof

- How to: Safely cast Using pattern matching, as, and is Operators
- Walkthrough: Creating and Using Dynamic Objects

interface (C# Reference)

8/25/2018 • 2 minutes to read • Edit Online

An interface contains only the signatures of methods, properties, events or indexers. A class or struct that implements the interface must implement the members of the interface that are specified in the interface definition. In the following example, class `ImplementationClass` must implement a method named `SampleMethod` that has no parameters and returns `void`.

For more information and examples, see Interfaces.

Example

```
interface ISampleInterface
{
    void SampleMethod();
}

class ImplementationClass : ISampleInterface
{
    // Explicit interface member implementation:
    void ISampleInterface.SampleMethod()
    {
        // Method implementation.
    }

    static void Main()
    {
        // Declare an interface instance.
        ISampleInterface obj = new ImplementationClass();

        // Call the member.
        obj.SampleMethod();
    }
}
```

An interface can be a member of a namespace or a class and can contain signatures of the following members:

- Methods
- Properties
- Indexers
- Events

An interface can inherit from one or more base interfaces.

When a base type list contains a base class and interfaces, the base class must come first in the list.

A class that implements an interface can explicitly implement members of that interface. An explicitly implemented member cannot be accessed through a class instance, but only through an instance of the interface.

For more details and code examples on explicit interface implementation, see Explicit Interface Implementation.

Example

The following example demonstrates interface implementation. In this example, the interface contains the property declaration and the class contains the implementation. Any instance of a class that implements `IPoint` has integer properties `x` and `y`.

```
interface IPoint
{
   // Property signatures:
   int x
   {
      get;
      set;
   }

   int y
   {
      get;
      set;
   }
}

class Point : IPoint
{
   // Fields:
   private int _x;
   private int _y;

   // Constructor:
   public Point(int x, int y)
   {
      _x = x;
      _y = y;
   }

   // Property implementation:
   public int x
   {
      get
      {
         return _x;
      }

      set
      {
         _x = value;
      }
   }

   public int y
   {
      get
      {
         return _y;
      }
      set
      {
         _y = value;
      }
   }
}

class MainClass
{
   static void PrintPoint(IPoint p)
   {
      Console.WriteLine("x={0}, y={1}", p.x, p.y);
   }

   static void Main()
```

```
    {
        IPoint p = new Point(2, 3);
        Console.Write("My Point: ");
        PrintPoint(p);
    }
}
// Output: My Point: x=2, y=3
```

C# language specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- C# Reference
- C# Programming Guide
- C# Keywords
- Reference Types
- Interfaces
- Using Properties
- Using Indexers
- class
- struct
- Interfaces

object (C# Reference)

8/24/2018 • 2 minutes to read • Edit Online

The `object` type is an alias for Object in .NET. In the unified type system of C#, all types, predefined and user-defined, reference types and value types, inherit directly or indirectly from Object. You can assign values of any type to variables of type `object`. When a variable of a value type is converted to object, it is said to be *boxed*. When a variable of type object is converted to a value type, it is said to be *unboxed*. For more information, see Boxing and Unboxing.

Example

The following sample shows how variables of type `object` can accept values of any data type and how variables of type `object` can use methods on Object from the .NET Framework.

```
class ObjectTest
{
   public int i = 10;
}

class MainClass2
{
   static void Main()
   {
      object a;
      a = 1;   // an example of boxing
      Console.WriteLine(a);
      Console.WriteLine(a.GetType());
      Console.WriteLine(a.ToString());

      a = new ObjectTest();
      ObjectTest classRef;
      classRef = (ObjectTest)a;
      Console.WriteLine(classRef.i);
   }
}
/* Output
    1
    System.Int32
    1
 * 10
*/
```

C# language specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- C# Reference
- C# Programming Guide
- C# Keywords
- Reference Types
- Value Types

string (C# Reference)

8/25/2018 • 2 minutes to read • Edit Online

The `string` type represents a sequence of zero or more Unicode characters. `string` is an alias for String in .NET.

Although `string` is a reference type, the equality operators (`==` and `!=`) are defined to compare the values of `string` objects, not references. This makes testing for string equality more intuitive. For example:

```
string a = "hello";
string b = "h";
// Append to contents of 'b'
b += "ello";
Console.WriteLine(a == b);
Console.WriteLine((object)a == (object)b);
```

This displays "True" and then "False" because the content of the strings are equivalent, but `a` and `b` do not refer to the same string instance.

The + operator concatenates strings:

```
string a = "good " + "morning";
```

This creates a string object that contains "good morning".

Strings are *immutable*--the contents of a string object cannot be changed after the object is created, although the syntax makes it appear as if you can do this. For example, when you write this code, the compiler actually creates a new string object to hold the new sequence of characters, and that new object is assigned to b. The string "h" is then eligible for garbage collection.

```
string b = "h";
b += "ello";
```

The [] operator can be used for readonly access to individual characters of a `string`:

```
string str = "test";
char x = str[2];  // x = 's';
```

String literals are of type `string` and can be written in two forms, quoted and @-quoted. Quoted string literals are enclosed in double quotation marks ("):

```
"good morning"  // a string literal
```

String literals can contain any character literal. Escape sequences are included. The following example uses escape sequence `\\` for backslash, `\u0066` for the letter f, and `\n` for newline.

```
string a = "\\\u0066\n";
Console.WriteLine(a);
```

NOTE

The escape code `\udddd` (where `dddd` is a four-digit number) represents the Unicode character U+`dddd`. Eight-digit Unicode escape codes are also recognized: `\Udddddddd`.

Verbatim string literals start with `@` and are also enclosed in double quotation marks. For example:

```
@"good morning"  // a string literal
```

The advantage of verbatim strings is that escape sequences are *not* processed, which makes it easy to write, for example, a fully qualified file name:

```
@"c:\Docs\Source\a.txt"  // rather than "c:\\Docs\\Source\\a.txt"
```

To include a double quotation mark in an @-quoted string, double it:

```
@"""Ahoy!"" cried the captain." // "Ahoy!" cried the captain.
```

For other uses of the `@` special character, see @ -- verbatim identifier.

For more information about strings in C#, see Strings.

Example

```
class SimpleStringTest
{
   static void Main()
   {
      string a = "\u0068ello ";
      string b = "world";
      Console.WriteLine( a + b );
      Console.WriteLine( a + b == "Hello World" ); // == performs a case-sensitive comparison
   }
}
/* Output:
    hello world
    False
 */
```

C# Language Specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See Also

- C# Reference
- C# Programming Guide
- Best Practices for Using Strings
- C# Keywords
- C# Programming Guide
- Reference Types

- Value Types
- Basic String Operations
- Creating New Strings
- Formatting Numeric Results Table

void (C# Reference)

8/25/2018 • 2 minutes to read • Edit Online

When used as the return type for a method, `void` specifies that the method doesn't return a value.

`void` isn't allowed in the parameter list of a method. A method that takes no parameters and returns no value is declared as follows:

```
public void SampleMethod()
{
    // Body of the method.
}
```

`void` is also used in an unsafe context to declare a pointer to an unknown type. For more information, see Pointer types.

`void` is an alias for the .NET Framework System.Void type.

C# Language Specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- C# Reference
- C# Programming Guide
- C# Keywords
- Reference Types
- Value Types
- Methods
- Unsafe Code and Pointers

var (C# Reference)

8/25/2018 • 2 minutes to read • Edit Online

Beginning in Visual C# 3.0, variables that are declared at method scope can have an implicit "type" `var`. An implicitly typed local variable is strongly typed just as if you had declared the type yourself, but the compiler determines the type. The following two declarations of `i` are functionally equivalent:

```
var i = 10; // Implicitly typed.
int i = 10; // Explicitly typed.
```

For more information, see Implicitly Typed Local Variables and Type Relationships in LINQ Query Operations.

Example

The following example shows two query expressions. In the first expression, the use of `var` is permitted but is not required, because the type of the query result can be stated explicitly as an `IEnumerable<string>`. However, in the second expression, `var` allows the result to be a collection of anonymous types, and the name of that type is not accessible except to the compiler itself. Use of `var` eliminates the requirement to create a new class for the result. Note that in Example #2, the `foreach` iteration variable `item` must also be implicitly typed.

```
// Example #1: var is optional because
// the select clause specifies a string
string[] words = { "apple", "strawberry", "grape", "peach", "banana" };
var wordQuery = from word in words
                where word[0] == 'g'
                select word;

// Because each element in the sequence is a string,
// not an anonymous type, var is optional here also.
foreach (string s in wordQuery)
{
    Console.WriteLine(s);
}

// Example #2: var is required when
// the select clause specifies an anonymous type
var custQuery = from cust in customers
                where cust.City == "Phoenix"
                select new { cust.Name, cust.Phone };

// var must be used because each item
// in the sequence is an anonymous type
foreach (var item in custQuery)
{
    Console.WriteLine("Name={0}, Phone={1}", item.Name, item.Phone);
}
```

See Also

- C# Reference
- C# Programming Guide
- Implicitly Typed Local Variables

Reference tables for types (C# Reference)

8/29/2018 • 2 minutes to read • Edit Online

The following reference tables summarize the C# types:

- Built-in Types Table
- Integral types
- Floating-point types
- Default values
- Value types
- Implicit numeric conversions
- Explicit Numeric Conversions Table

For information on formatting the output of numeric types, see Formatting Numeric Results Table.

See also

- C# Reference
- C# Programming Guide
- Reference Types
- Value Types

Built-in types table (C# Reference)

8/19/2018 • 2 minutes to read • Edit Online

The following table shows the keywords for built-in C# types, which are aliases of predefined types in the System namespace.

C# TYPE	.NET TYPE
bool	System.Boolean
byte	System.Byte
sbyte	System.SByte
char	System.Char
decimal	System.Decimal
double	System.Double
float	System.Single
int	System.Int32
uint	System.UInt32
long	System.Int64
ulong	System.UInt64
object	System.Object
short	System.Int16
ushort	System.UInt16
string	System.String

Remarks

All of the types in the table, except `object` and `string`, are referred to as simple types.

The C# type keywords and their aliases are interchangeable. For example, you can declare an integer variable by using either of the following declarations:

```
int x = 123;
System.Int32 y = 123;
```

Use the typeof operator to get the System.Type instance that represents the specified type:

```
Type stringType = typeof(string);
Console.WriteLine(stringType.FullName);

Type doubleType = typeof(System.Double);
Console.WriteLine(doubleType.FullName);

// Output:
// System.String
// System.Double
```

See also

- C# Reference
- C# Programming Guide
- C# Keywords
- Reference tables for types
- Value types
- Reference types
- Default values table
- dynamic

Integral types table (C# Reference)

8/22/2018 • 2 minutes to read • Edit Online

The following table shows the sizes and ranges of the integral types, which constitute a subset of simple types.

TYPE	RANGE	SIZE
sbyte	-128 to 127	Signed 8-bit integer
byte	0 to 255	Unsigned 8-bit integer
char	U+0000 to U+ffff	Unicode 16-bit character
short	-32,768 to 32,767	Signed 16-bit integer
ushort	0 to 65,535	Unsigned 16-bit integer
int	-2,147,483,648 to 2,147,483,647	Signed 32-bit integer
uint	0 to 4,294,967,295	Unsigned 32-bit integer
long	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	Signed 64-bit integer
ulong	0 to 18,446,744,073,709,551,615	Unsigned 64-bit integer

Remarks

If the value represented by an integer literal exceeds UInt64.MaxValue, a compiler error CS1021 occurs.

Use the System.Numerics.BigInteger class to represent an arbitrarily large signed integer.

See also

- C# Reference
- C# Programming Guide
- C# Keywords
- Reference tables for types
- Floating-point types table
- Default values table
- Formatting numeric results table
- Built-in types table

Floating-point types table (C# Reference)

8/22/2018 • 2 minutes to read • Edit Online

The following table shows the precision and approximate ranges for the floating-point types.

TYPE	APPROXIMATE RANGE	PRECISION
float	$\pm 1.5 \times 10^{-45}$ to $\pm 3.4 \times 10^{38}$	7 digits
double	$\pm 5.0 \times 10^{-324}$ to $\pm 1.7 \times 10^{308}$	15-16 digits
decimal	$\pm 1.0 \times 10^{-28}$ to $\pm 7.9228 \times 10^{28}$	28-29 digits

See also

- C# Reference
- C# Programming Guide
- C# Keywords
- Reference tables for types
- Integral types table
- Default values table
- Formatting numeric results table
- Built-in types table

Default values table (C# Reference)

8/24/2018 • 2 minutes to read • Edit Online

The following table shows the default values of value types.

VALUE TYPE	DEFAULT VALUE
bool	`false`
byte	0
char	'\0'
decimal	0M
double	0.0D
enum	The value produced by the expression `(E)0`, where `E` is the enum identifier.
float	0.0F
int	0
long	0L
sbyte	0
short	0
struct	The value produced by setting all value-type fields to their default values and all reference-type fields to `null`.
uint	0
ulong	0
ushort	0

Remarks

You cannot use uninitialized variables in C#. You can initialize a variable with the default value of its type. You also can use the default value of a type to specify the default value of a method's optional argument.

Use the default value expression to produce the default value of a type, as the following example shows:

```
int a = default(int);
```

Beginning with C# 7.1, you can use the `default` literal to initialize a variable with the default value of its type:

```
int a = default;
```

You also can use the default constructor or the implicit default constructor to produce the default value of a value type, as the following example shows. For more information about constructors, see the Constructors article.

```
int a = new int();
```

The default value of any reference type is `null`. The default value of a nullable type is an instance for which the HasValue property is `false` and the Value property is undefined.

See also

- C# Reference
- C# Programming Guide
- C# Keywords
- Reference tables for types
- Value types
- Value types table
- Built-in types table

Value types table (C# Reference)

8/25/2018 • 2 minutes to read • Edit Online

The following table shows the C# value types.

VALUE TYPE	CATEGORY	TYPE SUFFIX
bool	Boolean	
byte	Unsigned, numeric, integral	
char	Unsigned, numeric, integral	
decimal	Numeric, floating-point	M or m
double	Numeric, floating-point	D or d
enum	Enumeration	
float	Numeric, floating-point	F or f
int	Signed, numeric, integral	
long	Signed, numeric, integral	L or l
sbyte	Signed, numeric, integral	
short	Signed, numeric, integral	
struct	User-defined structure	
uint	Unsigned, numeric, integral	U or u
ulong	Unsigned, numeric, integral	UL, Ul, uL, ul, LU, Lu, lU, or lu
ushort	Unsigned, numeric, integral	

Remarks

You use a type suffix to specify a type of a numerical literal. For example:

```
decimal a = 0.1M;
```

If an integer numerical literal has no suffix, it has the first of the following types in which its value can be represented: `int`, `uint`, `long`, `ulong`.

If a real numerical literal has no suffix, it's of type `double`.

See also

- C# Reference
- C# Programming Guide
- Reference tables for types
- Default values table
- Value types
- Formatting numeric results table

Implicit numeric conversions table (C# Reference)

9/7/2018 • 2 minutes to read • Edit Online

The following table shows the predefined implicit conversions between .NET numeric types.

FROM	TO
sbyte	`short`, `int`, `long`, `float`, `double`, or `decimal`
byte	`short`, `ushort`, `int`, `uint`, `long`, `ulong`, `float`, `double`, or `decimal`
short	`int`, `long`, `float`, `double`, or `decimal`
ushort	`int`, `uint`, `long`, `ulong`, `float`, `double`, or `decimal`
int	`long`, `float`, `double`, or `decimal`
uint	`long`, `ulong`, `float`, `double`, or `decimal`
long	`float`, `double`, or `decimal`
char	`ushort`, `int`, `uint`, `long`, `ulong`, `float`, `double`, or `decimal`
float	`double`
ulong	`float`, `double`, or `decimal`

Remarks

- Any integral type is implicitly convertible to any floating-point type.
- Precision but not magnitude might be lost in the conversions from `int`, `uint`, `long`, or `ulong` to `float` and from `long` or `ulong` to `double`.
- There are no implicit conversions to the `char` type.
- There are no implicit conversions between the `float` and `double` types and the `decimal` type.
- A value of a constant expression of type `int` (for example, a value represented by an integral literal) can be converted to `sbyte`, `byte`, `short`, `ushort`, `uint`, or `ulong`, provided it's within the range of the destination type:

```
byte a = 13;    // Compiles
byte b = 300;   // CS0031: Constant value '300' cannot be converted to a 'byte'
```

For more information about implicit conversions, see the Implicit conversions section of the C# language specification.

See also

- C# Reference
- C# Programming Guide
- Integral types table
- Floating-point types table
- Built-in types table
- Explicit numeric conversions table
- Casting and type conversions

Explicit numeric conversions table (C# Reference)

9/7/2018 • 3 minutes to read • Edit Online

The following table shows the predefined explicit conversions between .NET numeric types for which there is no implicit conversion.

FROM	TO
sbyte	`byte`, `ushort`, `uint`, `ulong`, or `char`
byte	`sbyte` or `char`
short	`sbyte`, `byte`, `ushort`, `uint`, `ulong`, or `char`
ushort	`sbyte`, `byte`, `short`, or `char`
int	`sbyte`, `byte`, `short`, `ushort`, `uint`, `ulong`,or `char`
uint	`sbyte`, `byte`, `short`, `ushort`, `int`, or `char`
long	`sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `ulong`, or `char`
ulong	`sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, or `char`
char	`sbyte`, `byte`, or `short`
float	`sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`,or `decimal`
double	`sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`,or `decimal`
decimal	`sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, or `double`

Remarks

- The explicit numeric conversion may cause loss of precision or result in throwing an exception, typically an OverflowException.
- When you convert a value of an integral type to another integral type, the result depends on the overflow checking context. In a checked context, the conversion succeeds if the source value is within the range of the destination type. Otherwise, an OverflowException is thrown. In an unchecked context, the conversion always succeeds, and proceeds as follows:
  - If the source type is larger than the destination type, then the source value is truncated by

discarding its "extra" most significant bits. The result is then treated as a value of the destination type.

  - If the source type is smaller than the destination type, then the source value is either sign-extended or zero-extended so that it is the same size as the destination type. Sign-extension is used if the source type is signed; zero-extension is used if the source type is unsigned. The result is then treated as a value of the destination type.

  - If the source type is the same size as the destination type, then the source value is treated as a value of the destination type.

- When you convert a `decimal` value to an integral type, this value is rounded towards zero to the nearest integral value. If the resulting integral value is outside the range of the destination type, an OverflowException is thrown.

- When you convert a `double` or `float` value to an integral type, this value is rounded towards zero to the nearest integral value. If the resulting integral value is outside the range of the destination type, the result depends on the overflow checking context. In a checked context, an OverflowException is thrown, while in an unchecked context, the result is an unspecified value of the destination type.

- When you convert `double` to `float`, the `double` value is rounded to the nearest `float` value. If the `double` value is too small or too large to fit into the destination type, the result will be zero or infinity.

- When you convert `float` or `double` to `decimal`, the source value is converted to `decimal` representation and rounded to the nearest number after the 28th decimal place if required. Depending on the value of the source value, one of the following results may occur:

  - If the source value is too small to be represented as a `decimal`, the result becomes zero.

  - If the source value is NaN (not a number), infinity, or too large to be represented as a `decimal`, an OverflowException is thrown.

- When you convert `decimal` to `float` or `double`, the `decimal` value is rounded to the nearest `double` or `float` value.

For more information about explicit conversions, see the Explicit conversions section of the C# language specification.

See also

- C# Reference
- C# Programming Guide
- Casting and type conversions
- () Operator
- Integral types table
- Floating-point types table
- Built-in types table
- Implicit numeric conversions table

Formatting Numeric Results Table (C# Reference)

8/25/2018 • 2 minutes to read • Edit Online

You can format numeric results by using the String.Format method, through the Console.Write or Console.WriteLine methods, which call `String.Format`, or by using string interpolation. The format is specified by using format strings. The following table contains the supported standard format strings. The format string takes the following form: `Axx`, where `A` is the format specifier and `xx` is the precision specifier. The format specifier controls the type of formatting applied to the numeric value, and the precision specifier controls the number of significant digits or decimal places of the formatted output. The value of the precision specifier ranges from 0 to 99.

For more information about standard and custom formatting strings, see Formatting Types.

FORMAT SPECIFIER	DESCRIPTION	EXAMPLES	OUTPUT
C or c	Currency	Console.Write("{0:C}", 2.5); Console.Write("{0:C}", -2.5);	$2.50 ($2.50)
D or d	Decimal	Console.Write("{0:D5}", 25);	00025
E or e	Scientific	Console.Write("{0:E}", 250000);	2.500000E+005
F or f	Fixed-point	Console.Write("{0:F2}", 25); Console.Write("{0:F0}", 25);	25.00 25
G or g	General	Console.Write("{0:G}", 2.5);	2.5
N or n	Number	Console.Write("{0:N}", 2500000);	2,500,000.00
X or x	Hexadecimal	Console.Write("{0:X}", 250); Console.Write("{0:X}", 0xffff);	FA FFFF

See Also

- C# Reference
- C# Programming Guide
- Standard Numeric Format Strings
- Reference Tables for Types
- string

Modifiers (C# Reference)

8/25/2018 • 2 minutes to read • Edit Online

Modifiers are used to modify declarations of types and type members. This section introduces the C# modifiers.

MODIFIER	PURPOSE
Access Modifiers - public - private - internal - protected	Specifies the declared accessibility of types and type members.
abstract	Indicates that a class is intended only to be a base class of other classes.
async	Indicates that the modified method, lambda expression, or anonymous method is asynchronous.
const	Specifies that the value of the field or the local variable cannot be modified.
event	Declares an event.
extern	Indicates that the method is implemented externally.
new	Explicitly hides a member inherited from a base class.
override	Provides a new implementation of a virtual member inherited from a base class.
partial	Defines partial classes, structs and methods throughout the same assembly.
readonly	Declares a field that can only be assigned values as part of the declaration or in a constructor in the same class.
sealed	Specifies that a class cannot be inherited.
static	Declares a member that belongs to the type itself instead of to a specific object.
unsafe	Declares an unsafe context.
virtual	Declares a method or an accessor whose implementation can be changed by an overriding member in a derived class.
volatile	Indicates that a field can be modified in the program by something such as the operating system, the hardware, or a concurrently executing thread.

See Also

- C# Reference
- C# Programming Guide
- C# Keywords

Access Modifiers (C# Reference)

8/25/2018 • 2 minutes to read • Edit Online

Access modifiers are keywords used to specify the declared accessibility of a member or a type. This section introduces the four access modifiers:

- `public`
- `protected`
- `internal`
- `private`

The following six accessibility levels can be specified using the access modifiers:

- `public`: Access is not restricted.
- `protected`: Access is limited to the containing class or types derived from the containing class.
- `internal`: Access is limited to the current assembly.
- `protected internal`: Access is limited to the current assembly or types derived from the containing class.
- `private`: Access is limited to the containing type.
- `private protected`: Access is limited to the containing class or types derived from the containing class within the current assembly.

This section also introduces the following:

- Accessibility Levels: Using the four access modifiers to declare six levels of accessibility.
- Accessibility Domain: Specifies where, in the program sections, a member can be referenced.
- Restrictions on Using Accessibility Levels: A summary of the restrictions on using declared accessibility levels.

See Also

- C# Reference
- C# Programming Guide
- C# Keywords
- Access Modifiers
- Access Keywords
- Modifiers

Accessibility Levels (C# Reference)

8/25/2018 • 2 minutes to read • Edit Online

Use the access modifiers, `public`, `protected`, `internal`, or `private`, to specify one of the following declared accessibility levels for members.

DECLARED ACCESSIBILITY	MEANING
`public`	Access is not restricted.
`protected`	Access is limited to the containing class or types derived from the containing class.
`internal`	Access is limited to the current assembly.
`protected internal`	Access is limited to the current assembly or types derived from the containing class.
`private`	Access is limited to the containing type.
`private protected`	Access is limited to the containing class or types derived from the containing class within the current assembly. Available since C# 7.2.

Only one access modifier is allowed for a member or type, except when you use the `protected internal` or `private protected` combinations.

Access modifiers are not allowed on namespaces. Namespaces have no access restrictions.

Depending on the context in which a member declaration occurs, only certain declared accessibilities are permitted. If no access modifier is specified in a member declaration, a default accessibility is used.

Top-level types, which are not nested in other types, can only have `internal` or `public` accessibility. The default accessibility for these types is `internal`.

Nested types, which are members of other types, can have declared accessibilities as indicated in the following table.

MEMBERS OF	DEFAULT MEMBER ACCESSIBILITY	ALLOWED DECLARED ACCESSIBILITY OF THE MEMBER
`enum`	`public`	None

MEMBERS OF	DEFAULT MEMBER ACCESSIBILITY	ALLOWED DECLARED ACCESSIBILITY OF THE MEMBER

`class`	`private`	`public`
		`protected`
		`internal`
		`private`
		`protected internal`
		`private protected`

`interface`	`public`	None

`struct`	`private`	`public`
		`internal`
		`private`

The accessibility of a nested type depends on its accessibility domain, which is determined by both the declared accessibility of the member and the accessibility domain of the immediately containing type. However, the accessibility domain of a nested type cannot exceed that of the containing type.

C# Language Specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See Also

- C# Reference
- C# Programming Guide
- C# Keywords
- Access Modifiers
- Accessibility Domain
- Restrictions on Using Accessibility Levels
- Access Modifiers
- public
- private
- protected
- internal

Accessibility Domain (C# Reference)

8/25/2018 • 2 minutes to read • Edit Online

The accessibility domain of a member specifies in which program sections a member can be referenced. If the member is nested within another type, its accessibility domain is determined by both the accessibility level of the member and the accessibility domain of the immediately containing type.

The accessibility domain of a top-level type is at least the program text of the project that it is declared in. That is, the domain includes all of the source files of this project. The accessibility domain of a nested type is at least the program text of the type in which it is declared. That is, the domain is the type body, which includes all nested types. The accessibility domain of a nested type never exceeds that of the containing type. These concepts are demonstrated in the following example.

Example

This example contains a top-level type, `T1`, and two nested classes, `M1` and `M2`. The classes contain fields that have different declared accessibilities. In the `Main` method, a comment follows each statement to indicate the accessibility domain of each member. Notice that the statements that try to reference the inaccessible members are commented out. If you want to see the compiler errors caused by referencing an inaccessible member, remove the comments one at a time.

```
namespace AccessibilityDomainNamespace
{
    public class T1
    {
        public static int publicInt;
        internal static int internalInt;
        private static int privateInt = 0;
        static T1()
        {
            // T1 can access public or internal members
            // in a public or private (or internal) nested class
            M1.publicInt = 1;
            M1.internalInt = 2;
            M2.publicInt = 3;
            M2.internalInt = 4;

            // Cannot access the private member privateInt
            // in either class:
            // M1.privateInt = 2; //CS0122
        }

        public class M1
        {
            public static int publicInt;
            internal static int internalInt;
            private static int privateInt = 0;
        }

        private class M2
        {
            public static int publicInt = 0;
            internal static int internalInt = 0;
            private static int privateInt = 0;
        }
    }
```

```
    class MainClass
    {
        static void Main()
        {
            // Access is unlimited:
            T1.publicInt = 1;

            // Accessible only in current assembly:
            T1.internalInt = 2;

            // Error CS0122: inaccessible outside T1:
            // T1.privateInt = 3;

            // Access is unlimited:
            T1.M1.publicInt = 1;

            // Accessible only in current assembly:
            T1.M1.internalInt = 2;

            // Error CS0122: inaccessible outside M1:
            //    T1.M1.privateInt = 3;

            // Error CS0122: inaccessible outside T1:
            //    T1.M2.publicInt = 1;

            // Error CS0122: inaccessible outside T1:
            //    T1.M2.internalInt = 2;

            // Error CS0122: inaccessible outside M2:
            //    T1.M2.privateInt = 3;



            // Keep the console open in debug mode.
            System.Console.WriteLine("Press any key to exit.");
            System.Console.ReadKey();

        }
    }
}
```

C# Language Specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See Also

- C# Reference
- C# Programming Guide
- C# Keywords
- Access Modifiers
- Accessibility Levels
- Restrictions on Using Accessibility Levels
- Access Modifiers
- public
- private
- protected
- internal

Restrictions on using accessibility levels (C# Reference)

8/29/2018 • 2 minutes to read • Edit Online

When you specify a type in a declaration, check whether the accessibility level of the type is dependent on the accessibility level of a member or of another type. For example, the direct base class must be at least as accessible as the derived class. The following declarations cause a compiler error because the base class `BaseClass` is less accessible than `MyClass`:

```
class BaseClass {...}
public class MyClass: BaseClass {...} // Error
```

The following table summarizes the restrictions on declared accessibility levels.

CONTEXT	REMARKS
Classes	The direct base class of a class type must be at least as accessible as the class type itself.
Interfaces	The explicit base interfaces of an interface type must be at least as accessible as the interface type itself.
Delegates	The return type and parameter types of a delegate type must be at least as accessible as the delegate type itself.
Constants	The type of a constant must be at least as accessible as the constant itself.
Fields	The type of a field must be at least as accessible as the field itself.
Methods	The return type and parameter types of a method must be at least as accessible as the method itself.
Properties	The type of a property must be at least as accessible as the property itself.
Events	The type of an event must be at least as accessible as the event itself.
Indexers	The type and parameter types of an indexer must be at least as accessible as the indexer itself.
Operators	The return type and parameter types of an operator must be at least as accessible as the operator itself.
Constructors	The parameter types of a constructor must be at least as accessible as the constructor itself.

Example

The following example contains erroneous declarations of different types. The comment following each declaration indicates the expected compiler error.

```
// Restrictions on Using Accessibility Levels
// CS0052 expected as well as CS0053, CS0056, and CS0057
// To make the program work, change access level of both class B
// and MyPrivateMethod() to public.

using System;

// A delegate:
delegate int MyDelegate();

class B
{
    // A private method:
    static int MyPrivateMethod()
    {
        return 0;
    }
}

public class A
{
    // Error: The type B is less accessible than the field A.myField.
    public B myField = new B();

    // Error: The type B is less accessible
    // than the constant A.myConst.
    public readonly B myConst = new B();

    public B MyMethod()
    {
        // Error: The type B is less accessible
        // than the method A.MyMethod.
        return new B();
    }

    // Error: The type B is less accessible than the property A.MyProp
    public B MyProp
    {
        set
        {
        }
    }

    MyDelegate d = new MyDelegate(B.MyPrivateMethod);
    // Even when B is declared public, you still get the error:
    // "The parameter B.MyPrivateMethod is not accessible due to
    // protection level."

    public static B operator +(A m1, B m2)
    {
        // Error: The type B is less accessible
        // than the operator A.operator +(A,B)
        return new B();
    }

    static void Main()
    {
        Console.Write("Compiled successfully");
    }
}
```

C# language specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- C# Reference
- C# Programming Guide
- C# Keywords
- Access Modifiers
- Accessibility Domain
- Accessibility Levels
- Access Modifiers
- public
- private
- protected
- internal

internal (C# Reference)

8/25/2018 • 2 minutes to read • Edit Online

The `internal` keyword is an access modifier for types and type members.

> This page covers `internal` access. The `internal` keyword is also part of the `protected internal` access modifier.

Internal types or members are accessible only within files in the same assembly, as in this example:

```
public class BaseClass
{
    // Only accessible within the same assembly
    internal static int x = 0;
}
```

For a comparison of `internal` with the other access modifiers, see Accessibility Levels and Access Modifiers.

For more information about assemblies, see Assemblies and the Global Assembly Cache.

A common use of internal access is in component-based development because it enables a group of components to cooperate in a private manner without being exposed to the rest of the application code. For example, a framework for building graphical user interfaces could provide `Control` and `Form` classes that cooperate by using members with internal access. Since these members are internal, they are not exposed to code that is using the framework.

It is an error to reference a type or a member with internal access outside the assembly within which it was defined.

Example

This example contains two files, `Assembly1.cs` and `Assembly1_a.cs`. The first file contains an internal base class, `BaseClass`. In the second file, an attempt to instantiate `BaseClass` will produce an error.

```
// Assembly1.cs
// Compile with: /target:library
internal class BaseClass
{
   public static int intM = 0;
}
```

```
// Assembly1_a.cs
// Compile with: /reference:Assembly1.dll
class TestAccess
{
   static void Main()
   {
      BaseClass myBase = new BaseClass();   // CS0122
   }
}
```

Example

In this example, use the same files you used in example 1, and change the accessibility level of `BaseClass` to `public`. Also change the accessibility level of the member `IntM` to `internal`. In this case, you can instantiate the class, but you cannot access the internal member.

```
// Assembly2.cs
// Compile with: /target:library
public class BaseClass
{
   internal static int intM = 0;
}
```

```
// Assembly2_a.cs
// Compile with: /reference:Assembly2.dll
public class TestAccess
{
   static void Main()
   {
      BaseClass myBase = new BaseClass();   // Ok.
      BaseClass.intM = 444;    // CS0117
   }
}
```

C# Language Specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See Also

- C# Reference
- C# Programming Guide
- C# Keywords
- Access Modifiers
- Accessibility Levels
- Modifiers
- public
- private
- protected

private (C# Reference)

8/29/2018 • 2 minutes to read • Edit Online

The `private` keyword is a member access modifier.

> This page covers `private` access. The `private` keyword is also part of the `private protected` access modifier.

Private access is the least permissive access level. Private members are accessible only within the body of the class or the struct in which they are declared, as in this example:

```
class Employee
{
    private int i;
    double d;   // private access by default
}
```

Nested types in the same body can also access those private members.

It is a compile-time error to reference a private member outside the class or the struct in which it is declared.

For a comparison of `private` with the other access modifiers, see Accessibility Levels and Access Modifiers.

Example

In this example, the `Employee` class contains two private data members, `name` and `salary`. As private members, they cannot be accessed except by member methods. Public methods named `GetName` and `Salary` are added to allow controlled access to the private members. The `name` member is accessed by way of a public method, and the `salary` member is accessed by way of a public read-only property. (See Properties for more information.)

```
class Employee2
{
    private string name = "FirstName, LastName";
    private double salary = 100.0;

    public string GetName()
    {
        return name;
    }

    public double Salary
    {
        get { return salary; }
    }
}

class PrivateTest
{
    static void Main()
    {
        Employee2 e = new Employee2();

        // The data members are inaccessible (private), so
        // they can't be accessed like this:
        //    string n = e.name;
        //    double s = e.salary;

        // 'name' is indirectly accessed via method:
        string n = e.GetName();

        // 'salary' is indirectly accessed via property
        double s = e.Salary;
    }
}
```

C# language specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- C# Reference
- C# Programming Guide
- C# Keywords
- Access Modifiers
- Accessibility Levels
- Modifiers
- public
- protected
- internal

protected (C# Reference)

8/29/2018 • 2 minutes to read • Edit Online

The `protected` keyword is a member access modifier.

> This page covers `protected` access. The `protected` keyword is also part of the `protected internal` and `private protected` access modifiers.

A protected member is accessible within its class and by derived class instances.

For a comparison of `protected` with the other access modifiers, see Accessibility Levels.

Example

A protected member of a base class is accessible in a derived class only if the access occurs through the derived class type. For example, consider the following code segment:

```
class A
{
    protected int x = 123;
}

class B : A
{
    static void Main()
    {
        A a = new A();
        B b = new B();

        // Error CS1540, because x can only be accessed by
        // classes derived from A.
        // a.x = 10;

        // OK, because this class derives from A.
        b.x = 10;
    }
}
```

The statement `a.x = 10` generates an error because it is made within the static method Main, and not an instance of class B.

Struct members cannot be protected because the struct cannot be inherited.

Example

In this example, the class `DerivedPoint` is derived from `Point`. Therefore, you can access the protected members of the base class directly from the derived class.

```
class Point
{
    protected int x;
    protected int y;
}

class DerivedPoint: Point
{
    static void Main()
    {
        DerivedPoint dpoint = new DerivedPoint();

        // Direct access to protected members:
        dpoint.x = 10;
        dpoint.y = 15;
        Console.WriteLine("x = {0}, y = {1}", dpoint.x, dpoint.y);
    }
}
// Output: x = 10, y = 15
```

If you change the access levels of `x` and `y` to private, the compiler will issue the error messages:

`'Point.y' is inaccessible due to its protection level.`

`'Point.x' is inaccessible due to its protection level.`

C# language specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- C# Reference
- C# Programming Guide
- C# Keywords
- Access Modifiers
- Accessibility Levels
- Modifiers
- public
- private
- internal
- Security concerns for internal virtual keywords

public (C# Reference)

8/29/2018 • 2 minutes to read • Edit Online

The `public` keyword is an access modifier for types and type members. Public access is the most permissive access level. There are no restrictions on accessing public members, as in this example:

```
class SampleClass
{
    public int x; // No access restrictions.
}
```

See Access Modifiers and Accessibility Levels for more information.

Example

In the following example, two classes are declared, `PointTest` and `MainClass`. The public members `x` and `y` of `PointTest` are accessed directly from `MainClass`.

```
class PointTest
{
    public int x;
    public int y;
}

class MainClass4
{
    static void Main()
    {
        PointTest p = new PointTest();
        // Direct access to public members:
        p.x = 10;
        p.y = 15;
        Console.WriteLine("x = {0}, y = {1}", p.x, p.y);
    }
}
// Output: x = 10, y = 15
```

If you change the `public` access level to private or protected, you will get the error message:

'PointTest.y' is inaccessible due to its protection level.

C# language specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- C# Reference
- C# Programming Guide
- Access Modifiers
- C# Keywords
- Access Modifiers

- Accessibility Levels
- Modifiers
- private
- protected
- internal

protected internal (C# Reference)

8/29/2018 • 2 minutes to read • Edit Online

The `protected internal` keyword combination is a member access modifier. A protected internal member is accessible from the current assembly or from types that are derived from the containing class. For a comparison of `protected internal` with the other access modifiers, see Accessibility Levels.

Example

A protected internal member of a base class is accessible from any type within its containing assembly. It is also accessible in a derived class located in another assembly only if the access occurs through a variable of the derived class type. For example, consider the following code segment:

```
// Assembly1.cs
// Compile with: /target:library
public class BaseClass
{
   protected internal int myValue = 0;
}

class TestAccess
{
    void Access()
    {
        BaseClass baseObject = new BaseClass();
        baseObject.myValue = 5;
    }
}
```

```
// Assembly2.cs
// Compile with: /reference:Assembly1.dll
class DerivedClass : BaseClass
{
    static void Main()
    {
        BaseClass baseObject = new BaseClass();
        DerivedClass derivedObject = new DerivedClass();

        // Error CS1540, because myValue can only be accessed by
        // classes derived from BaseClass.
        // baseObject.myValue = 10;

        // OK, because this class derives from BaseClass.
        derivedObject.myValue = 10;
    }
}
```

This example contains two files, `Assembly1.cs` and `Assembly2.cs`. The first file contains a public base class, `BaseClass`, and another class, `TestAccess`. `BaseClass` owns a protected internal member, `myValue`, which is accessed by the `TestAccess` type. In the second file, an attempt to access `myValue` through an instance of `BaseClass` will produce an error, while an access to this member through an instance of a derived class, `DerivedClass` will succeed.

Struct members cannot be `protected internal` because the struct cannot be inherited.

C# language specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- C# Reference
- C# Programming Guide
- C# Keywords
- Access Modifiers
- Accessibility Levels
- Modifiers
- public
- private
- internal
- Security concerns for internal virtual keywords

private protected (C# Reference)

8/29/2018 • 2 minutes to read • Edit Online

The `private protected` keyword combination is a member access modifier. A private protected member is accessible by types derived from the containing class, but only within its containing assembly. For a comparison of `private protected` with the other access modifiers, see Accessibility Levels.

> **NOTE**
>
> The `private protected` access modifier is valid in C# version 7.2 and later.

Example

A private protected member of a base class is accessible from derived types in its containing assembly only if the static type of the variable is the derived class type. For example, consider the following code segment:

```
// Assembly1.cs
// Compile with: /target:library
public class BaseClass
{
    private protected int myValue = 0;
}

public class DerivedClass1 : BaseClass
{
    void Access()
    {
        BaseClass baseObject = new BaseClass();

        // Error CS1540, because myValue can only be accessed by
        // classes derived from BaseClass.
        // baseObject.myValue = 5;

        // OK, accessed through the current derived class instance
        myValue = 5;
    }
}
```

```
// Assembly2.cs
// Compile with: /reference:Assembly1.dll
class DerivedClass2 : BaseClass
{
    void Access()
    {
        // Error CS0122, because myValue can only be
        // accessed by types in Assembly1
        // myValue = 10;
    }
}
```

This example contains two files, `Assembly1.cs` and `Assembly2.cs`. The first file contains a public base class, `BaseClass`, and a type derived from it, `DerivedClass1`. `BaseClass` owns a private protected member, `myValue`, which `DerivedClass1` tries to access in two ways. The first attempt to access `myValue` through an instance of `BaseClass` will produce an error. However, the attempt to use it as an inherited member in `DerivedClass1` will succeed. In the second file, an attempt to access `myValue` as an inherited member of `DerivedClass2` will produce an error, as it is only accessible by derived types in Assembly1.

Struct members cannot be `private protected` because the struct cannot be inherited.

C# language specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- C# Reference
- C# Programming Guide
- C# Keywords
- Access Modifiers
- Accessibility Levels
- Modifiers
- public
- private
- internal
- Security concerns for internal virtual keywords

abstract (C# Reference)

8/25/2018 • 3 minutes to read • Edit Online

The `abstract` modifier indicates that the thing being modified has a missing or incomplete implementation. The abstract modifier can be used with classes, methods, properties, indexers, and events. Use the `abstract` modifier in a class declaration to indicate that a class is intended only to be a base class of other classes. Members marked as abstract, or included in an abstract class, must be implemented by classes that derive from the abstract class.

Example

In this example, the class `Square` must provide an implementation of `Area` because it derives from `ShapesClass`:

```
abstract class ShapesClass
{
    abstract public int Area();
}
class Square : ShapesClass
{
    int side = 0;

    public Square(int n)
    {
        side = n;
    }
    // Area method is required to avoid
    // a compile-time error.
    public override int Area()
    {
        return side * side;
    }

    static void Main()
    {
        Square sq = new Square(12);
        Console.WriteLine("Area of the square = {0}", sq.Area());
    }

    interface I
    {
        void M();
    }
    abstract class C : I
    {
        public abstract void M();
    }

}
// Output: Area of the square = 144
```

Abstract classes have the following features:

- An abstract class cannot be instantiated.
- An abstract class may contain abstract methods and accessors.
- It is not possible to modify an abstract class with the sealed modifier because the two modifers have opposite meanings. The `sealed` modifier prevents a class from being inherited and the `abstract` modifier

requires a class to be inherited.

- A non-abstract class derived from an abstract class must include actual implementations of all inherited abstract methods and accessors.

Use the `abstract` modifier in a method or property declaration to indicate that the method or property does not contain implementation.

Abstract methods have the following features:

- An abstract method is implicitly a virtual method.
- Abstract method declarations are only permitted in abstract classes.
- Because an abstract method declaration provides no actual implementation, there is no method body; the method declaration simply ends with a semicolon and there are no curly braces ({ }) following the signature. For example:

  ```
  public abstract void MyMethod();
  ```

  The implementation is provided by a method override, which is a member of a non-abstract class.

- It is an error to use the static or virtual modifiers in an abstract method declaration.

Abstract properties behave like abstract methods, except for the differences in declaration and invocation syntax.

- It is an error to use the `abstract` modifier on a static property.
- An abstract inherited property can be overridden in a derived class by including a property declaration that uses the override modifier.

For more information about abstract classes, see Abstract and Sealed Classes and Class Members.

An abstract class must provide implementation for all interface members.

An abstract class that implements an interface might map the interface methods onto abstract methods. For example:

```
interface I
{
    void M();
}
abstract class C : I
{
    public abstract void M();
}
```

Example

In this example, the class `DerivedClass` is derived from an abstract class `BaseClass`. The abstract class contains an abstract method, `AbstractMethod`, and two abstract properties, `X` and `Y`.

```
abstract class BaseClass   // Abstract class
{
    protected int _x = 100;
    protected int _y = 150;
    public abstract void AbstractMethod();   // Abstract method
    public abstract int X    { get; }
    public abstract int Y    { get; }
}

class DerivedClass : BaseClass
{
    public override void AbstractMethod()
    {
        _x++;
        _y++;
    }

    public override int X   // overriding property
    {
        get
        {
            return _x + 10;
        }
    }

    public override int Y   // overriding property
    {
        get
        {
            return _y + 10;
        }
    }

    static void Main()
    {
        DerivedClass o = new DerivedClass();
        o.AbstractMethod();
        Console.WriteLine("x = {0}, y = {1}", o.X, o.Y);
    }
}
// Output: x = 111, y = 161
```

In the preceding example, if you attempt to instantiate the abstract class by using a statement like this:

```
BaseClass bc = new BaseClass();   // Error
```

You will get an error saying that the compiler cannot create an instance of the abstract class 'BaseClass'.

C# Language Specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See Also

- C# Reference
- C# Programming Guide
- Modifiers
- virtual
- override

- C# Keywords

async (C# Reference)

8/25/2018 • 4 minutes to read • Edit Online

Use the `async` modifier to specify that a method, lambda expression, or anonymous method is asynchronous. If you use this modifier on a method or expression, it's referred to as an *async method*. The following example defines an async method named `ExampleMethodAsync`:

```
public async Task<int> ExampleMethodAsync()
{
    // . . . .
}
```

If you're new to asynchronous programming or do not understand how an async method uses the `await` keyword to do potentially long-running work without blocking the caller's thread, read the introduction in Asynchronous Programming with async and await. The following code is found inside an async method and calls the HttpClient.GetStringAsync method:

```
string contents = await httpClient.GetStringAsync(requestUrl);
```

An async method runs synchronously until it reaches its first `await` expression, at which point the method is suspended until the awaited task is complete. In the meantime, control returns to the caller of the method, as the example in the next section shows.

If the method that the `async` keyword modifies doesn't contain an `await` expression or statement, the method executes synchronously. A compiler warning alerts you to any async methods that don't contain `await` statements, because that situation might indicate an error. See Compiler Warning (level 1) CS4014.

The `async` keyword is contextual in that it's a keyword only when it modifies a method, a lambda expression, or an anonymous method. In all other contexts, it's interpreted as an identifier.

Example

The following example shows the structure and flow of control between an async event handler, `StartButton_Click`, and an async method, `ExampleMethodAsync`. The result from the async method is the number of characters of a web page. The code is suitable for a Windows Presentation Foundation (WPF) app or Windows Store app that you create in Visual Studio; see the code comments for setting up the app.

You can run this code in Visual Studio as a Windows Presentation Foundation (WPF) app or a Windows Store app. You need a Button control named `StartButton` and a Textbox control named `ResultsTextBox`. Remember to set the names and handler so that you have something like this:

```
<Button Content="Button" HorizontalAlignment="Left" Margin="88,77,0,0" VerticalAlignment="Top" Width="75"
        Click="StartButton_Click" Name="StartButton"/>
<TextBox HorizontalAlignment="Left" Height="137" Margin="88,140,0,0" TextWrapping="Wrap"
         Text="&lt;Enter a URL&gt;" VerticalAlignment="Top" Width="310" Name="ResultsTextBox"/>
```

To run the code as a WPF app:

- Paste this code into the `MainWindow` class in MainWindow.xaml.cs.
- Add a reference to System.Net.Http.

- Add a `using` directive for System.Net.Http.

To run the code as a Windows Store app:

- Paste this code into the `MainPage` class in MainPage.xaml.cs.
- Add using directives for System.Net.Http and System.Threading.Tasks.

```
private async void StartButton_Click(object sender, RoutedEventArgs e)
{
    // ExampleMethodAsync returns a Task<int>, which means that the method
    // eventually produces an int result. However, ExampleMethodAsync returns
    // the Task<int> value as soon as it reaches an await.
    ResultsTextBox.Text += "\n";

    try
    {
        int length = await ExampleMethodAsync();
        // Note that you could put "await ExampleMethodAsync()" in the next line where
        // "length" is, but due to when '+=' fetches the value of ResultsTextBox, you
        // would not see the global side effect of ExampleMethodAsync setting the text.
        ResultsTextBox.Text += String.Format("Length: {0:N0}\n", length);
    }
    catch (Exception)
    {
        // Process the exception if one occurs.
    }
}

public async Task<int> ExampleMethodAsync()
{
    var httpClient = new HttpClient();
    int exampleInt = (await httpClient.GetStringAsync("http://msdn.microsoft.com")).Length;
    ResultsTextBox.Text += "Preparing to finish ExampleMethodAsync.\n";
    // After the following return statement, any method that's awaiting
    // ExampleMethodAsync (in this case, StartButton_Click) can get the
    // integer result.
    return exampleInt;
}
// The example displays the following output:
// Preparing to finish ExampleMethodAsync.
// Length: 53292
```

IMPORTANT

For more information about tasks and the code that executes while waiting for a task, see Asynchronous Programming with async and await. For a full WPF example that uses similar elements, see Walkthrough: Accessing the Web by Using Async and Await.

Return Types

An async method can have the following return types:

- Task
- Task<TResult>
- void, which should only be used for event handlers.
- Starting with C# 7.0, any type that has an accessible `GetAwaiter` method. The `System.Threading.Tasks.ValueTask<TResult>` type is one such implementation. It is available by adding the NuGet package `System.Threading.Tasks.Extensions`.

The async method can't declare any in, ref or out parameters, nor can it have a reference return value, but it can call methods that have such parameters.

You specify `Task<TResult>` as the return type of an async method if the return statement of the method specifies an operand of type `TResult`. You use `Task` if no meaningful value is returned when the method is completed. That is, a call to the method returns a `Task`, but when the `Task` is completed, any `await` expression that's awaiting the `Task` evaluates to `void`.

You use the `void` return type primarily to define event handlers, which require that return type. The caller of a `void`-returning async method can't await it and can't catch exceptions that the method throws.

Starting with C# 7.0, you return another type, typically a value type, that has a `GetAwaiter` method to miminize memory allocations in performance-critical sections of code.

For more information and examples, see Async Return Types.

See Also

- AsyncStateMachineAttribute
- await
- Walkthrough: Accessing the Web by Using Async and Await
- Asynchronous Programming with async and await

const (C# Reference)

8/25/2018 • 2 minutes to read • Edit Online

You use the `const` keyword to declare a constant field or a constant local. Constant fields and locals aren't variables and may not be modified. Constants can be numbers, Boolean values, strings, or a null reference. Don't create a constant to represent information that you expect to change at any time. For example, don't use a constant field to store the price of a service, a product version number, or the brand name of a company. These values can change over time, and because compilers propagate constants, other code compiled with your libraries will have to be recompiled to see the changes. See also the readonly keyword. For example:

```
const int x = 0;
public const double gravitationalConstant = 6.673e-11;
private const string productName = "Visual C#";
```

Remarks

The type of a constant declaration specifies the type of the members that the declaration introduces. The initializer of a constant local or a constant field must be a constant expression that can be implicitly converted to the target type.

A constant expression is an expression that can be fully evaluated at compile time. Therefore, the only possible values for constants of reference types are `string` and a null reference.

The constant declaration can declare multiple constants, such as:

```
public const double x = 1.0, y = 2.0, z = 3.0;
```

The `static` modifier is not allowed in a constant declaration.

A constant can participate in a constant expression, as follows:

```
public const int c1 = 5;
public const int c2 = c1 + 100;
```

NOTE

The readonly keyword differs from the `const` keyword. A `const` field can only be initialized at the declaration of the field. A `readonly` field can be initialized either at the declaration or in a constructor. Therefore, `readonly` fields can have different values depending on the constructor used. Also, although a `const` field is a compile-time constant, the `readonly` field can be used for run-time constants, as in this line: `public static readonly uint l1 = (uint)DateTime.Now.Ticks;`

Example

```
public class ConstTest
{
    class SampleClass
    {
        public int x;
        public int y;
        public const int c1 = 5;
        public const int c2 = c1 + 5;

        public SampleClass(int p1, int p2)
        {
            x = p1;
            y = p2;
        }
    }

    static void Main()
    {
        SampleClass mC = new SampleClass(11, 22);
        Console.WriteLine("x = {0}, y = {1}", mC.x, mC.y);
        Console.WriteLine("c1 = {0}, c2 = {1}",
                          SampleClass.c1, SampleClass.c2);
    }
}
/* Output
    x = 11, y = 22
    c1 = 5, c2 = 10
*/
```

Example

This example demonstrates how to use constants as local variables.

```
public class SealedTest
{
    static void Main()
    {
        const int c = 707;
        Console.WriteLine("My local constant = {0}", c);
    }
}
// Output: My local constant = 707
```

C# language specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- C# Reference
- C# Programming Guide
- C# Keywords
- Modifiers
- readonly

event (C# Reference)

8/25/2018 • 2 minutes to read • Edit Online

The `event` keyword is used to declare an event in a publisher class.

Example

The following example shows how to declare and raise an event that uses EventHandler as the underlying delegate type. For the complete code example that also shows how to use the generic EventHandler<TEventArgs> delegate type and how to subscribe to an event and create an event handler method, see How to: Publish Events that Conform to .NET Framework Guidelines.

```
public class SampleEventArgs
{
    public SampleEventArgs(string s) { Text = s; }
    public String Text { get; } // readonly
}
public class Publisher
{
    // Declare the delegate (if using non-generic pattern).
    public delegate void SampleEventHandler(object sender, SampleEventArgs e);

    // Declare the event.
    public event SampleEventHandler SampleEvent;

    // Wrap the event in a protected virtual method
    // to enable derived classes to raise the event.
    protected virtual void RaiseSampleEvent()
    {
        // Raise the event by using the () operator.
        if (SampleEvent != null)
            SampleEvent(this, new SampleEventArgs("Hello"));
    }
}
```

Events are a special kind of multicast delegate that can only be invoked from within the class or struct where they are declared (the publisher class). If other classes or structs subscribe to the event, their event handler methods will be called when the publisher class raises the event. For more information and code examples, see Events and Delegates.

Events can be marked as public, private, protected, internal, protected internal or private protected. These access modifiers define how users of the class can access the event. For more information, see Access Modifiers.

Keywords and Events

The following keywords apply to events.

KEYWORD	DESCRIPTION	FOR MORE INFORMATION
static	Makes the event available to callers at any time, even if no instance of the class exists.	Static Classes and Static Class Members

KEYWORD	DESCRIPTION	FOR MORE INFORMATION
virtual	Allows derived classes to override the event behavior by using the override keyword.	Inheritance
sealed	Specifies that for derived classes it is no longer virtual.	
abstract	The compiler will not generate the `add` and `remove` event accessor blocks and therefore derived classes must provide their own implementation.	

An event may be declared as a static event by using the static keyword. This makes the event available to callers at any time, even if no instance of the class exists. For more information, see Static Classes and Static Class Members.

An event can be marked as a virtual event by using the virtual keyword. This enables derived classes to override the event behavior by using the override keyword. For more information, see Inheritance. An event overriding a virtual event can also be sealed, which specifies that for derived classes it is no longer virtual. Lastly, an event can be declared abstract, which means that the compiler will not generate the `add` and `remove` event accessor blocks. Therefore derived classes must provide their own implementation.

C# Language Specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See Also

- C# Reference
- C# Programming Guide
- C# Keywords
- add
- remove
- Modifiers
- How to: Combine Delegates (Multicast Delegates)

extern (C# Reference)

8/25/2018 • 2 minutes to read • Edit Online

The `extern` modifier is used to declare a method that is implemented externally. A common use of the `extern` modifier is with the `DllImport` attribute when you are using Interop services to call into unmanaged code. In this case, the method must also be declared as `static`, as shown in the following example:

```
[DllImport("avifil32.dll")]
private static extern void AVIFileInit();
```

The `extern` keyword can also define an external assembly alias, which makes it possible to reference different versions of the same component from within a single assembly. For more information, see extern alias.

It is an error to use the abstract and `extern` modifiers together to modify the same member. Using the `extern` modifier means that the method is implemented outside the C# code, whereas using the `abstract` modifier means that the method implementation is not provided in the class.

The extern keyword has more limited uses in C# than in C++. To compare the C# keyword with the C++ keyword, see Using extern to Specify Linkage in the C++ Language Reference.

Example 1

In this example, the program receives a string from the user and displays it inside a message box. The program uses the `MessageBox` method imported from the User32.dll library.

```
//using System.Runtime.InteropServices;
class ExternTest
{
    [DllImport("User32.dll", CharSet=CharSet.Unicode)]
    public static extern int MessageBox(IntPtr h, string m, string c, int type);

    static int Main()
    {
        string myString;
        Console.Write("Enter your message: ");
        myString = Console.ReadLine();
        return MessageBox((IntPtr)0, myString, "My Message Box", 0);
    }

}
```

Example 2

This example illustrates a C# program that calls into a C library (a native DLL).

1. Create the following C file and name it `cmdll.c`:

```
// cmdll.c
// Compile with: -LD
int __declspec(dllexport) SampleMethod(int i)
{
  return i*10;
}
```

2. Open a Visual Studio x64 (or x32) Native Tools Command Prompt window from the Visual Studio installation directory and compile the `cmdll.c` file by typing **cl -LD cmdll.c** at the command prompt.

3. In the same directory, create the following C# file and name it `cm.cs`:

```
// cm.cs
using System;
using System.Runtime.InteropServices;
public class MainClass
{
    [DllImport("Cmdll.dll")]
      public static extern int SampleMethod(int x);

    static void Main()
    {
        Console.WriteLine("SampleMethod() returns {0}.", SampleMethod(5));
    }
}
```

4. Open a Visual Studio x64 (or x32) Native Tools Command Prompt window from the Visual Studio installation directory and compile the `cm.cs` file by typing:

> **csc cm.cs** (for the x64 command prompt) —or— **csc -platform:x86 cm.cs** (for the x32 command prompt)

This will create the executable file `cm.exe`.

5. Run `cm.exe`. The `SampleMethod` method passes the value 5 to the DLL file, which returns the value multiplied by 10. The program produces the following output:

```
SampleMethod() returns 50.
```

C# language specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- DllImportAttribute
- C# Reference
- C# Programming Guide
- C# Keywords
- Modifiers

in (Generic Modifier) (C# Reference)

8/25/2018 • 2 minutes to read • Edit Online

For generic type parameters, the `in` keyword specifies that the type parameter is contravariant. You can use the `in` keyword in generic interfaces and delegates.

Contravariance enables you to use a less derived type than that specified by the generic parameter. This allows for implicit conversion of classes that implement variant interfaces and implicit conversion of delegate types. Covariance and contravariance in generic type parameters are supported for reference types, but they are not supported for value types.

A type can be declared contravariant in a generic interface or delegate only if it defines the type of a method's parameters and not of a method's return type. `In`, `ref`, and `out` parameters must be invariant, meaning they are neither covariant or contravariant.

An interface that has a contravariant type parameter allows its methods to accept arguments of less derived types than those specified by the interface type parameter. For example, in the IComparer<T> interface, type T is contravariant, you can assign an object of the `IComparer<Person>` type to an object of the `IComparer<Employee>` type without using any special conversion methods if `Employee` inherits `Person`.

A contravariant delegate can be assigned another delegate of the same type, but with a less derived generic type parameter.

For more information, see Covariance and Contravariance.

Contravariant generic interface

The following example shows how to declare, extend, and implement a contravariant generic interface. It also shows how you can use implicit conversion for classes that implement this interface.

```
// Contravariant interface.
interface IContravariant<in A> { }

// Extending contravariant interface.
interface IExtContravariant<in A> : IContravariant<A> { }

// Implementing contravariant interface.
class Sample<A> : IContravariant<A> { }

class Program
{
    static void Test()
    {
        IContravariant<Object> iobj = new Sample<Object>();
        IContravariant<String> istr = new Sample<String>();

        // You can assign iobj to istr because
        // the IContravariant interface is contravariant.
        istr = iobj;
    }
}
```

Contravariant generic delegate

The following example shows how to declare, instantiate, and invoke a contravariant generic delegate. It also shows how you can implicitly convert a delegate type.

```
// Contravariant delegate.
public delegate void DContravariant<in A>(A argument);

// Methods that match the delegate signature.
public static void SampleControl(Control control)
{ }
public static void SampleButton(Button button)
{ }

public void Test()
{

    // Instantiating the delegates with the methods.
    DContravariant<Control> dControl = SampleControl;
    DContravariant<Button> dButton = SampleButton;

    // You can assign dControl to dButton
    // because the DContravariant delegate is contravariant.
    dButton = dControl;

    // Invoke the delegate.
    dButton(new Button());
}
```

C# language specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- out
- Covariance and Contravariance
- Modifiers

out (generic modifier) (C# Reference)

8/24/2018 • 2 minutes to read • Edit Online

For generic type parameters, the `out` keyword specifies that the type parameter is covariant. You can use the `out` keyword in generic interfaces and delegates.

Covariance enables you to use a more derived type than that specified by the generic parameter. This allows for implicit conversion of classes that implement variant interfaces and implicit conversion of delegate types. Covariance and contravariance are supported for reference types, but they are not supported for value types.

An interface that has a covariant type parameter enables its methods to return more derived types than those specified by the type parameter. For example, because in .NET Framework 4, in IEnumerable<T>, type T is covariant, you can assign an object of the `IEnumerable(Of String)` type to an object of the `IEnumerable(Of Object)` type without using any special conversion methods.

A covariant delegate can be assigned another delegate of the same type, but with a more derived generic type parameter.

For more information, see Covariance and Contravariance.

Example - covariant generic interface

The following example shows how to declare, extend, and implement a covariant generic interface. It also shows how to use implicit conversion for classes that implement a covariant interface.

```
// Covariant interface.
interface ICovariant<out R> { }

// Extending covariant interface.
interface IExtCovariant<out R> : ICovariant<R> { }

// Implementing covariant interface.
class Sample<R> : ICovariant<R> { }

class Program
{
    static void Test()
    {
        ICovariant<Object> iobj = new Sample<Object>();
        ICovariant<String> istr = new Sample<String>();

        // You can assign istr to iobj because
        // the ICovariant interface is covariant.
        iobj = istr;
    }
}
```

In a generic interface, a type parameter can be declared covariant if it satisfies the following conditions:

- The type parameter is used only as a return type of interface methods and not used as a type of method arguments.

> **NOTE**
>
> There is one exception to this rule. If in a covariant interface you have a contravariant generic delegate as a method parameter, you can use the covariant type as a generic type parameter for this delegate. For more information about covariant and contravariant generic delegates, see Variance in Delegates and Using Variance for Func and Action Generic Delegates.

- The type parameter is not used as a generic constraint for the interface methods.

Example - covariant generic delegate

The following example shows how to declare, instantiate, and invoke a covariant generic delegate. It also shows how to implicitly convert delegate types.

```
// Covariant delegate.
public delegate R DCovariant<out R>();

// Methods that match the delegate signature.
public static Control SampleControl()
{ return new Control(); }

public static Button SampleButton()
{ return new Button(); }

public void Test()
{
    // Instantiate the delegates with the methods.
    DCovariant<Control> dControl = SampleControl;
    DCovariant<Button> dButton = SampleButton;

    // You can assign dButton to dControl
    // because the DCovariant delegate is covariant.
    dControl = dButton;

    // Invoke the delegate.
    dControl();
}
```

In a generic delegate, a type can be declared covariant if it is used only as a method return type and not used for method arguments.

C# language specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- Variance in Generic Interfaces
- in
- Modifiers

override (C# Reference)

8/24/2018 • 2 minutes to read • Edit Online

The `override` modifier is required to extend or modify the abstract or virtual implementation of an inherited method, property, indexer, or event.

Example

In this example, the `Square` class must provide an overridden implementation of `Area` because `Area` is inherited from the abstract `ShapesClass`:

```
abstract class ShapesClass
{
    abstract public int Area();
}
class Square : ShapesClass
{
    int side = 0;

    public Square(int n)
    {
        side = n;
    }
    // Area method is required to avoid
    // a compile-time error.
    public override int Area()
    {
        return side * side;
    }

    static void Main()
    {
        Square sq = new Square(12);
        Console.WriteLine("Area of the square = {0}", sq.Area());
    }

    interface I
    {
        void M();
    }
    abstract class C : I
    {
        public abstract void M();
    }

}
// Output: Area of the square = 144
```

An `override` method provides a new implementation of a member that is inherited from a base class. The method that is overridden by an `override` declaration is known as the overridden base method. The overridden base method must have the same signature as the `override` method. For information about inheritance, see Inheritance.

You cannot override a non-virtual or static method. The overridden base method must be `virtual`, `abstract`, or `override`.

An `override` declaration cannot change the accessibility of the `virtual` method. Both the `override` method and the `virtual` method must have the same access level modifier.

You cannot use the `new`, `static`, or `virtual` modifiers to modify an `override` method.

An overriding property declaration must specify exactly the same access modifier, type, and name as the inherited property, and the overridden property must be `virtual`, `abstract`, or `override`.

For more information about how to use the `override` keyword, see Versioning with the Override and New Keywords and Knowing when to use Override and New Keywords.

Example

This example defines a base class named `Employee`, and a derived class named `SalesEmployee`. The `SalesEmployee` class includes an extra field, `salesbonus`, and overrides the method `CalculatePay` in order to take it into account.

```
class TestOverride
{
    public class Employee
    {
        public string name;

        // Basepay is defined as protected, so that it may be
        // accessed only by this class and derived classes.
        protected decimal basepay;

        // Constructor to set the name and basepay values.
        public Employee(string name, decimal basepay)
        {
            this.name = name;
            this.basepay = basepay;
        }

        // Declared virtual so it can be overridden.
        public virtual decimal CalculatePay()
        {
            return basepay;
        }
    }

    // Derive a new class from Employee.
    public class SalesEmployee : Employee
    {
        // New field that will affect the base pay.
        private decimal salesbonus;

        // The constructor calls the base-class version, and
        // initializes the salesbonus field.
        public SalesEmployee(string name, decimal basepay,
                  decimal salesbonus) : base(name, basepay)
        {
            this.salesbonus = salesbonus;
        }

        // Override the CalculatePay method
        // to take bonus into account.
        public override decimal CalculatePay()
        {
            return basepay + salesbonus;
        }
    }

    static void Main()
    {
        // Create some new employees.
        SalesEmployee employee1 = new SalesEmployee("Alice",
                      1000, 500);
        Employee employee2 = new Employee("Bob", 1200);

        Console.WriteLine("Employee4 " + employee1.name +
                  " earned: " + employee1.CalculatePay());
        Console.WriteLine("Employee4 " + employee2.name +
                  " earned: " + employee2.CalculatePay());
    }
}
/*
    Output:
    Employee4 Alice earned: 1500
    Employee4 Bob earned: 1200
*/
```

C# language specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- C# Reference
- C# Programming Guide
- Inheritance
- C# Keywords
- Modifiers
- abstract
- virtual
- new
- Polymorphism

readonly (C# Reference)

8/29/2018 • 3 minutes to read • Edit Online

The `readonly` keyword is a modifier that can be used in three contexts:

- In a field declaration, `readonly` indicates that assignment to the field can only occur as part of the declaration or in a constructor in the same class.
- In a `readonly struct` definition, `readonly` indicates that the `struct` is immutable.
- In a `ref readonly` method return, the `readonly` modifier indicates that method returns a reference and writes are not allowed to that reference.

The final two contexts were added in C# 7.2.

Readonly field example

In this example, the value of the field `year` cannot be changed in the method `ChangeYear`, even though it is assigned a value in the class constructor:

```
class Age
{
    readonly int year;
    Age(int year)
    {
        this.year = year;
    }
    void ChangeYear()
    {
        //year = 1967; // Compile error if uncommented.
    }
}
```

You can assign a value to a `readonly` field only in the following contexts:

- When the variable is initialized in the declaration, for example:

```
public readonly int y = 5;
```

- In an instance constructor of the class that contains the instance field declaration.
- In the static constructor of the class that contains the static field declaration.

These constructor contexts are also the only contexts in which it is valid to pass a `readonly` field as an out or ref parameter.

> **NOTE**
>
> The `readonly` keyword is different from the const keyword. A `const` field can only be initialized at the declaration of the field. A `readonly` field can be initialized either at the declaration or in a constructor. Therefore, `readonly` fields can have different values depending on the constructor used. Also, while a `const` field is a compile-time constant, the `readonly` field can be used for runtime constants as in the following example:

```
public static readonly uint timeStamp = (uint)DateTime.Now.Ticks;
```

```
class SampleClass
{
    public int x;
    // Initialize a readonly field
    public readonly int y = 25;
    public readonly int z;

    public SampleClass()
    {
        // Initialize a readonly instance field
        z = 24;
    }

    public SampleClass(int p1, int p2, int p3)
    {
        x = p1;
        y = p2;
        z = p3;
    }

    public static void Main()
    {
        SampleClass p1 = new SampleClass(11, 21, 32);   // OK
        Console.WriteLine($"p1: x={p1.x}, y={p1.y}, z={p1.z}");
        SampleClass p2 = new SampleClass();
        p2.x = 55;   // OK
        Console.WriteLine($"p2: x={p2.x}, y={p2.y}, z={p2.z}");
    }
    /*
     Output:
        p1: x=11, y=21, z=32
        p2: x=55, y=25, z=24
    */
}
```

In the preceding example, if you use a statement like the following example:

`p2.y = 66; // Error`

you will get the compiler error message:

`A readonly field cannot be assigned to (except in a constructor or a variable initializer)`

Readonly struct example

The `readonly` modifier on a `struct` definition declares that the struct is **immutable**. Every instance field of the `struct` must be marked `readonly`, as shown in the following example:

```
public readonly struct Point
{
    public double X { get; }
    public double Y { get; }

    public Point(double x, double y) => (X, Y) = (x, y);

    public override string ToString() => $"({X}, {Y})";
}
```

The preceding example uses readonly auto properties to declare its storage. That instructs the compiler to create `readonly` backing fields for those properties. You could also declare `readonly` fields directly:

```
public readonly struct Point
{
    public readonly double X;
    public readonly double Y;

    public Point(double x, double y) => (X, Y) = (x, y);

    public override string ToString() => $"({X}, {Y})";
}
```

Adding a field not marked `readonly` generates compiler error `CS8340`: "Instance fields of readonly structs must be readonly."

Ref readonly return example

The `readonly` modifier on a `ref return` indicates that the returned reference cannot be modified. The following example returns a reference to the origin. It uses the `readonly` modifier to indicate that callers cannot modify the origin:

```
private static readonly Point origin = new Point(0, 0);
public static ref readonly Point Origin => ref origin;
```

The type returned doesn't need to be a `readonly struct`. Any type that can be returned by `ref` can be returned by `ref readonly`.

C# language specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- C# Reference
- C# Programming Guide
- C# Keywords
- Modifiers
- const
- Fields

sealed (C# Reference)

8/25/2018 • 2 minutes to read • Edit Online

When applied to a class, the `sealed` modifier prevents other classes from inheriting from it. In the following example, class `B` inherits from class `A`, but no class can inherit from class `B`.

```
class A {}
sealed class B : A {}
```

You can also use the `sealed` modifier on a method or property that overrides a virtual method or property in a base class. This enables you to allow classes to derive from your class and prevent them from overriding specific virtual methods or properties.

Example

In the following example, `Z` inherits from `Y` but `Z` cannot override the virtual function `F` that is declared in `X` and sealed in `Y`.

```
class X
{
    protected virtual void F() { Console.WriteLine("X.F"); }
    protected virtual void F2() { Console.WriteLine("X.F2"); }
}
class Y : X
{
    sealed protected override void F() { Console.WriteLine("Y.F"); }
    protected override void F2() { Console.WriteLine("Y.F2"); }
}
class Z : Y
{
    // Attempting to override F causes compiler error CS0239.
    // protected override void F() { Console.WriteLine("Z.F"); }

    // Overriding F2 is allowed.
    protected override void F2() { Console.WriteLine("Z.F2"); }
}
```

When you define new methods or properties in a class, you can prevent deriving classes from overriding them by not declaring them as virtual.

It is an error to use the abstract modifier with a sealed class, because an abstract class must be inherited by a class that provides an implementation of the abstract methods or properties.

When applied to a method or property, the `sealed` modifier must always be used with override.

Because structs are implicitly sealed, they cannot be inherited.

For more information, see Inheritance.

For more examples, see Abstract and Sealed Classes and Class Members.

Example

```
sealed class SealedClass
{
    public int x;
    public int y;
}

class SealedTest2
{
    static void Main()
    {
        SealedClass sc = new SealedClass();
        sc.x = 110;
        sc.y = 150;
        Console.WriteLine("x = {0}, y = {1}", sc.x, sc.y);
    }
}
// Output: x = 110, y = 150
```

In the previous example, you might try to inherit from the sealed class by using the following statement:

`class MyDerivedC: SealedClass {} // Error`

The result is an error message:

`'MyDerivedC': cannot derive from sealed type 'SealedClass'`

C# Language Specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

Remarks

To determine whether to seal a class, method, or property, you should generally consider the following two points:

- The potential benefits that deriving classes might gain through the ability to customize your class.
- The potential that deriving classes could modify your classes in such a way that they would no longer work correctly or as expected.

See Also

- C# Reference
- C# Programming Guide
- C# Keywords
- Static Classes and Static Class Members
- Abstract and Sealed Classes and Class Members
- Access Modifiers
- Modifiers
- override
- virtual

static (C# Reference)

8/25/2018 • 3 minutes to read • Edit Online

Use the `static` modifier to declare a static member, which belongs to the type itself rather than to a specific object. The `static` modifier can be used with classes, fields, methods, properties, operators, events, and constructors, but it cannot be used with indexers, finalizers, or types other than classes. For more information, see Static Classes and Static Class Members.

Example

The following class is declared as `static` and contains only `static` methods:

```
static class CompanyEmployee
{
    public static void DoSomething() { /*...*/ }
    public static void DoSomethingElse() { /*...*/  }
}
```

A constant or type declaration is implicitly a static member.

A static member cannot be referenced through an instance. Instead, it is referenced through the type name. For example, consider the following class:

```
public class MyBaseC
{
    public struct MyStruct
    {
        public static int x = 100;
    }
}
```

To refer to the static member `x`, use the fully qualified name, `MyBaseC.MyStruct.x`, unless the member is accessible from the same scope:

```
Console.WriteLine(MyBaseC.MyStruct.x);
```

While an instance of a class contains a separate copy of all instance fields of the class, there is only one copy of each static field.

It is not possible to use this to reference static methods or property accessors.

If the `static` keyword is applied to a class, all the members of the class must be static.

Classes and static classes may have static constructors. Static constructors are called at some point between when the program starts and the class is instantiated.

> **NOTE**
>
> The `static` keyword has more limited uses than in C++. To compare with the C++ keyword, see Storage classes (C++).

To demonstrate static members, consider a class that represents a company employee. Assume that the class contains a method to count employees and a field to store the number of employees. Both the method and the field do not belong to any instance employee. Instead they belong to the company class. Therefore, they should be declared as static members of the class.

Example

This example reads the name and ID of a new employee, increments the employee counter by one, and displays the information for the new employee and the new number of employees. For simplicity, this program reads the current number of employees from the keyboard. In a real application, this information should be read from a file.

```
public class Employee4
{
    public string id;
    public string name;

    public Employee4()
    {
    }

    public Employee4(string name, string id)
    {
        this.name = name;
        this.id = id;
    }

    public static int employeeCounter;

    public static int AddEmployee()
    {
        return ++employeeCounter;
    }
}

class MainClass : Employee4
{
    static void Main()
    {
        Console.Write("Enter the employee's name: ");
        string name = Console.ReadLine();
        Console.Write("Enter the employee's ID: ");
        string id = Console.ReadLine();

        // Create and configure the employee object:
        Employee4 e = new Employee4(name, id);
        Console.Write("Enter the current number of employees: ");
        string n = Console.ReadLine();
        Employee4.employeeCounter = Int32.Parse(n);
        Employee4.AddEmployee();

        // Display the new information:
        Console.WriteLine("Name: {0}", e.name);
        Console.WriteLine("ID:   {0}", e.id);
        Console.WriteLine("New Number of Employees: {0}",
                      Employee4.employeeCounter);
    }
}
    /*
    Input:
    Matthias Berndt
    AF643G
    15
     *
    Sample Output:
    Enter the employee's name: Matthias Berndt
    Enter the employee's ID: AF643G
    Enter the current number of employees: 15
    Name: Matthias Berndt
    ID:   AF643G
    New Number of Employees: 16
    */
```

Example

This example shows that although you can initialize a static field by using another static field not yet declared, the results will be undefined until you explicitly assign a value to the static field.

```
class Test
{
   static int x = y;
   static int y = 5;

   static void Main()
   {
      Console.WriteLine(Test.x);
      Console.WriteLine(Test.y);

      Test.x = 99;
      Console.WriteLine(Test.x);
   }
}
/*
Output:
    0
    5
    99
*/
```

C# Language Specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See Also

- C# Reference
- C# Programming Guide
- C# Keywords
- Modifiers
- Static Classes and Static Class Members

unsafe (C# Reference)

8/25/2018 • 2 minutes to read • Edit Online

The `unsafe` keyword denotes an unsafe context, which is required for any operation involving pointers. For more information, see Unsafe Code and Pointers.

You can use the `unsafe` modifier in the declaration of a type or a member. The entire textual extent of the type or member is therefore considered an unsafe context. For example, the following is a method declared with the `unsafe` modifier:

```
      unsafe static void FastCopy(byte[] src, byte[] dst, int count)
{
    // Unsafe context: can use pointers here.
}
```

The scope of the unsafe context extends from the parameter list to the end of the method, so pointers can also be used in the parameter list:

```
unsafe static void FastCopy ( byte* ps, byte* pd, int count ) {...}
```

You can also use an unsafe block to enable the use of an unsafe code inside this block. For example:

```
      unsafe
{
    // Unsafe context: can use pointers here.
}
```

To compile unsafe code, you must specify the /unsafe compiler option. Unsafe code is not verifiable by the common language runtime.

Example

```
// compile with: -unsafe

class UnsafeTest
{
   // Unsafe method: takes pointer to int:
   unsafe static void SquarePtrParam(int* p)
   {
      *p *= *p;
   }

   unsafe static void Main()
   {
      int i = 5;
      // Unsafe method: uses address-of operator (&):
      SquarePtrParam(&i);
      Console.WriteLine(i);
   }
}
// Output: 25
```

C# Language Specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See Also

- C# Reference
- C# Programming Guide
- C# Keywords
- fixed Statement
- Unsafe Code and Pointers
- Fixed Size Buffers

virtual (C# Reference)

8/25/2018 • 3 minutes to read • Edit Online

The `virtual` keyword is used to modify a method, property, indexer, or event declaration and allow for it to be overridden in a derived class. For example, this method can be overridden by any class that inherits it:

```
public virtual double Area()
{
    return x * y;
}
```

The implementation of a virtual member can be changed by an overriding member in a derived class. For more information about how to use the `virtual` keyword, see Versioning with the Override and New Keywords and Knowing When to Use Override and New Keywords.

Remarks

When a virtual method is invoked, the run-time type of the object is checked for an overriding member. The overriding member in the most derived class is called, which might be the original member, if no derived class has overridden the member.

By default, methods are non-virtual. You cannot override a non-virtual method.

You cannot use the `virtual` modifier with the `static`, `abstract`, `private`, or `override` modifiers. The following example shows a virtual property:

```
class MyBaseClass
{
    // virtual auto-implemented property. Overrides can only
    // provide specialized behavior if they implement get and set accessors.
    public virtual string Name { get; set; }

    // ordinary virtual property with backing field
    private int num;
    public virtual int Number
    {
        get { return num; }
        set { num = value; }
    }
}


class MyDerivedClass : MyBaseClass
{
    private string name;

   // Override auto-implemented property with ordinary property
   // to provide specialized accessor behavior.
    public override string Name
    {
        get
        {
            return name;
        }
        set
        {
            if (value != String.Empty)
            {
                name = value;
            }
            else
            {
                name = "Unknown";
            }
        }
    }

}
```

Virtual properties behave like abstract methods, except for the differences in declaration and invocation syntax.

- It is an error to use the `virtual` modifier on a static property.
- A virtual inherited property can be overridden in a derived class by including a property declaration that uses the `override` modifier.

Example

In this example, the `Shape` class contains the two coordinates `x`, `y`, and the `Area()` virtual method. Different shape classes such as `Circle`, `Cylinder`, and `Sphere` inherit the `Shape` class, and the surface area is calculated for each figure. Each derived class has its own override implementation of `Area()`.

Notice that the inherited classes `Circle`, `Sphere`, and `Cylinder` all use constructors that initialize the base class, as shown in the following declaration.

```
public Cylinder(double r, double h): base(r, h) {}
```

The following program calculates and displays the appropriate area for each figure by invoking the appropriate implementation of the `Area()` method, according to the object that is associated with the method.

```
class TestClass
{
    public class Shape
    {
        public const double PI = Math.PI;
        protected double x, y;
        public Shape()
        {
        }
        public Shape(double x, double y)
        {
            this.x = x;
            this.y = y;
        }

        public virtual double Area()
        {
            return x * y;
        }
    }

    public class Circle : Shape
    {
        public Circle(double r) : base(r, 0)
        {
        }

        public override double Area()
        {
            return PI * x * x;
        }
    }

    class Sphere : Shape
    {
        public Sphere(double r) : base(r, 0)
        {
        }

        public override double Area()
        {
            return 4 * PI * x * x;
        }
    }

    class Cylinder : Shape
    {
        public Cylinder(double r, double h) : base(r, h)
        {
        }

        public override double Area()
        {
            return 2 * PI * x * x + 2 * PI * x * y;
        }
    }

    static void Main()
    {
        double r = 3.0, h = 5.0;
        Shape c = new Circle(r);
        Shape s = new Sphere(r);
        Shape l = new Cylinder(r, h);
        // Display results:
        Console.WriteLine("Area of Circle   = {0:F2}", c.Area());
        Console.WriteLine("Area of Sphere   = {0:F2}", s.Area());
```

```
        Console.WriteLine("Area of Cylinder = {0:F2}", l.Area());
        }
    }
    /*
        Output:
        Area of Circle   = 28.27
        Area of Sphere   = 113.10
        Area of Cylinder = 150.80
    */
```

C# Language Specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See Also

- C# Reference
- C# Programming Guide
- Modifiers
- C# Keywords
- Polymorphism
- abstract
- override
- new

volatile (C# Reference)

8/25/2018 • 2 minutes to read • Edit Online

The `volatile` keyword indicates that a field might be modified by multiple threads that are executing at the same time. Fields that are declared `volatile` are not subject to compiler optimizations that assume access by a single thread. These restrictions ensure that all threads will observe volatile writes performed by any other thread in the order in which they were performed. There is no guarantee of a single total ordering of volatile writes as seen from all threads of execution.

The `volatile` modifier is usually used for a field that is accessed by multiple threads without using the lock statement to serialize access.

The `volatile` keyword can be applied to fields of these types:

- Reference types.
- Pointer types (in an unsafe context). Note that although the pointer itself can be volatile, the object that it points to cannot. In other words, you cannot declare a "pointer to volatile."
- Types such as sbyte, byte, short, ushort, int, uint, char, float, and bool.
- An enum type with one of the following base types: byte, sbyte, short, ushort, int, or uint.
- Generic type parameters known to be reference types.
- IntPtr and UIntPtr.

The volatile keyword can only be applied to fields of a class or struct. Local variables cannot be declared `volatile`.

Example

The following example shows how to declare a public field variable as `volatile`.

```
class VolatileTest
{
    public volatile int i;

    public void Test(int _i)
    {
        i = _i;
    }
}
```

Example

The following example demonstrates how an auxiliary or worker thread can be created and used to perform processing in parallel with that of the primary thread. For background information about multithreading, see Managed Threading and Threading (C#).

```
using System;
using System.Threading;

public class Worker
{
    // This method is called when the thread is started.
    public void DoWork()
    {
        while (!_shouldStop)
        {
            Console.WriteLine("Worker thread: working...");
        }
        Console.WriteLine("Worker thread: terminating gracefully.");
    }
    public void RequestStop()
    {
        _shouldStop = true;
    }
    // Keyword volatile is used as a hint to the compiler that this data
    // member is accessed by multiple threads.
    private volatile bool _shouldStop;
}

public class WorkerThreadExample
{
    static void Main()
    {
        // Create the worker thread object. This does not start the thread.
        Worker workerObject = new Worker();
        Thread workerThread = new Thread(workerObject.DoWork);

        // Start the worker thread.
        workerThread.Start();
        Console.WriteLine("Main thread: starting worker thread...");

        // Loop until the worker thread activates.
        while (!workerThread.IsAlive) ;

        // Put the main thread to sleep for 1 millisecond to
        // allow the worker thread to do some work.
        Thread.Sleep(1);

        // Request that the worker thread stop itself.
        workerObject.RequestStop();

        // Use the Thread.Join method to block the current thread
        // until the object's thread terminates.
        workerThread.Join();
        Console.WriteLine("Main thread: worker thread has terminated.");
    }
    // Sample output:
    // Main thread: starting worker thread...
    // Worker thread: working...
    // Worker thread: working...
    // Worker thread: working...
    // Worker thread: working...
    // Worker thread: working...
    // Worker thread: working...
    // Worker thread: terminating gracefully.
    // Main thread: worker thread has terminated.
}
```

C# Language Specification

For more information, see the C# Language Specification. The language specification is the definitive source for

C# syntax and usage.

See Also

- C# Reference
- C# Programming Guide
- C# Keywords
- Modifiers

Statement Keywords (C# Reference)

8/25/2018 • 2 minutes to read • Edit Online

Statements are program instructions. Except as described in the topics referenced in the following table, statements are executed in sequence. The following table lists the C# statement keywords. For more information about statements that are not expressed with any keyword, see Statements.

CATEGORY	C# KEYWORDS
Selection statements	if, else, switch, case
Iteration statements	do, for, foreach, in, while
Jump statements	break, continue, default, goto, return, yield
Exception handling statements	throw, try-catch, try-finally, try-catch-finally
Checked and unchecked	checked, unchecked
fixed Statement	fixed
lock Statement	lock

See Also

- C# Reference
- Statements
- C# Keywords

Selection Statements (C# Reference)

8/25/2018 • 2 minutes to read • Edit Online

A selection statement causes the program control to be transferred to a specific flow based upon whether a certain condition is `true` or not.

The following keywords are used in selection statements:

- if
- else
- switch
- case
- default

See Also

- C# Reference
- C# Programming Guide
- C# Keywords
- Statement Keywords

if-else (C# Reference)

8/25/2018 • 5 minutes to read • Edit Online

An `if` statement identifies which statement to run based on the value of a `Boolean` expression. In the following example, the `Boolean` variable `result` is set to `true` and then checked in the `if` statement. The output is `The variable is set to true.`.

```
bool condition = true;

if (condition)
{
    Console.WriteLine("The variable is set to true.");
}
else
{
    Console.WriteLine("The variable is set to false.");
}
```

You can run the examples in this topic by placing them in the `Main` method of a console app.

An `if` statement in C# can take two forms, as the following example shows.

```
// if-else statement
if (condition)
{
    then-statement;
}
else
{
    else-statement;
}
// Next statement in the program.

// if statement without an else
if (condition)
{
    then-statement;
}
// Next statement in the program.
```

In an `if-else` statement, if `condition` evaluates to true, the `then-statement` runs. If `condition` is false, the `else-statement` runs. Because `condition` can't be simultaneously true and false, the `then-statement` and the `else-statement` of an `if-else` statement can never both run. After the `then-statement` or the `else-statement` runs, control is transferred to the next statement after the `if` statement.

In an `if` statement that doesn't include an `else` statement, if `condition` is true, the `then-statement` runs. If `condition` is false, control is transferred to the next statement after the `if` statement.

Both the `then-statement` and the `else-statement` can consist of a single statement or multiple statements that are enclosed in braces (`{}`). For a single statement, the braces are optional but recommended.

The statement or statements in the `then-statement` and the `else-statement` can be of any kind, including another `if` statement nested inside the original `if` statement. In nested `if` statements, each `else` clause belongs to the last `if` that doesn't have a corresponding `else`. In the following example, `Result1` appears if both `m > 10` and `n > 20` evaluate to true. If `m > 10` is true but `n > 20` is false, `Result2` appears.

```
// Try with m = 12 and then with m = 8.
int m = 12;
int n = 18;

if (m > 10)
    if (n > 20)
    {
        Console.WriteLine("Result1");
    }
    else
    {
        Console.WriteLine("Result2");
    }
```

If, instead, you want `Result2` to appear when `(m > 10)` is false, you can specify that association by using braces to establish the start and end of the nested `if` statement, as the following example shows.

```
// Try with m = 12 and then with m = 8.
if (m > 10)
{
    if (n > 20)
        Console.WriteLine("Result1");
}
else
{
    Console.WriteLine("Result2");
}
```

`Result2` appears if the condition `(m > 10)` evaluates to false.

Example

In the following example, you enter a character from the keyboard, and the program uses a nested `if` statement to determine whether the input character is an alphabetic character. If the input character is an alphabetic character, the program checks whether the input character is lowercase or uppercase. A message appears for each case.

```
Console.Write("Enter a character: ");
char c = (char)Console.Read();
if (Char.IsLetter(c))
{
    if (Char.IsLower(c))
    {
        Console.WriteLine("The character is lowercase.");
    }
    else
    {
        Console.WriteLine("The character is uppercase.");
    }
}
else
{
    Console.WriteLine("The character isn't an alphabetic character.");
}

//Sample Output:

//Enter a character: 2
//The character isn't an alphabetic character.

//Enter a character: A
//The character is uppercase.

//Enter a character: h
//The character is lowercase.
```

Example

You also can nest an `if` statement inside an else block, as the following partial code shows. The example nests `if` statements inside two else blocks and one then block. The comments specify which conditions are true or false in each block.

```
// Change the values of these variables to test the results.
bool Condition1 = true;
bool Condition2 = true;
bool Condition3 = true;
bool Condition4 = true;

if (Condition1)
{
    // Condition1 is true.
}
else if (Condition2)
{
    // Condition1 is false and Condition2 is true.
}
else if (Condition3)
{
    if (Condition4)
    {
        // Condition1 and Condition2 are false. Condition3 and Condition4 are true.
    }
    else
    {
        // Condition1, Condition2, and Condition4 are false. Condition3 is true.
    }
}
else
{
    // Condition1, Condition2, and Condition3 are false.
}
```

Example

The following example determines whether an input character is a lowercase letter, an uppercase letter, or a number. If all three conditions are false, the character isn't an alphanumeric character. The example displays a message for each case.

```
Console.Write("Enter a character: ");
char ch = (char)Console.Read();

if (Char.IsUpper(ch))
{
    Console.WriteLine("The character is an uppercase letter.");
}
else if (Char.IsLower(ch))
{
    Console.WriteLine("The character is a lowercase letter.");
}
else if (Char.IsDigit(ch))
{
    Console.WriteLine("The character is a number.");
}
else
{
    Console.WriteLine("The character is not alphanumeric.");
}

//Sample Input and Output:
//Enter a character: E
//The character is an uppercase letter.

//Enter a character: e
//The character is a lowercase letter.

//Enter a character: 4
//The character is a number.

//Enter a character: =
//The character is not alphanumeric.
```

Just as a statement in the else block or the then block can be any valid statement, you can use any valid Boolean expression for the condition. You can use logical operators such as &&, &, ||, | and ! to make compound conditions. The following code shows examples.

```
// NOT
bool result = true;
if (!result)
{
    Console.WriteLine("The condition is true (result is false).");
}
else
{
    Console.WriteLine("The condition is false (result is true).");
}

// Short-circuit AND
int m = 9;
int n = 7;
int p = 5;
if (m >= n && m >= p)
{
    Console.WriteLine("Nothing is larger than m.");
}

// AND and NOT
if (m >= n && !(p > m))
{
    Console.WriteLine("Nothing is larger than m.");
}

// Short-circuit OR
if (m > n || m > p)
{
    Console.WriteLine("m isn't the smallest.");
}

// NOT and OR
m = 4;
if (!(m >= n || m >= p))
{
    Console.WriteLine("Now m is the smallest.");
}
// Output:
// The condition is false (result is true).
// Nothing is larger than m.
// Nothing is larger than m.
// m isn't the smallest.
// Now m is the smallest.
```

C# language specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See Also

- C# Reference
- C# Programming Guide
- C# Keywords
- ?: Operator
- if-else Statement (C++)
- switch

switch (C# reference)

8/15/2018 • 15 minutes to read • Edit Online

`switch` is a selection statement that chooses a single *switch section* to execute from a list of candidates based on a pattern match with the *match expression*.

```
using System;

public class Example
{
   public static void Main()
   {
      int caseSwitch = 1;

      switch (caseSwitch)
      {
          case 1:
              Console.WriteLine("Case 1");
              break;
          case 2:
              Console.WriteLine("Case 2");
              break;
          default:
              Console.WriteLine("Default case");
              break;
      }
   }
}
// The example displays the following output:
//       Case 1
```

The `switch` statement is often used as an alternative to an if-else construct if a single expression is tested against three or more conditions. For example, the following `switch` statement determines whether a variable of type `Color` has one of three values:

```
using System;

public enum Color { Red, Green, Blue }

public class Example
{
   public static void Main()
   {
      Color c = (Color) (new Random()).Next(0, 3);
      switch (c)
      {
         case Color.Red:
            Console.WriteLine("The color is red");
            break;
         case Color.Green:
            Console.WriteLine("The color is green");
            break;
         case Color.Blue:
            Console.WriteLine("The color is blue");
            break;
         default:
            Console.WriteLine("The color is unknown.");
            break;
      }
   }
}
```

It is equivalent to the following example that uses an `if` - `else` construct.

```
using System;

public enum Color { Red, Green, Blue }

public class Example
{
   public static void Main()
   {
      Color c = (Color) (new Random()).Next(0, 3);
      if (c == Color.Red)
         Console.WriteLine("The color is red");
      else if (c == Color.Green)
         Console.WriteLine("The color is green");
      else if (c == Color.Blue)
         Console.WriteLine("The color is blue");
      else
         Console.WriteLine("The color is unknown.");
   }
}
// The example displays the following output:
//       The color is red
```

The match expression

The match expression provides the value to match against the patterns in `case` labels. Its syntax is:

```
   switch (expr)
```

In C# 6, the match expression must be an expression that returns a value of the following types:

- a char.
- a string.

- a bool.
- an integral value, such as an int or a long.
- an enum value.

Starting with C# 7.0, the match expression can be any non-null expression.

The switch section

A `switch` statement includes one or more switch sections. Each switch section contains one or more *case labels* (either a case or default label) followed by one or more statements. The `switch` statement may include at most one default label placed in any switch section. The following example shows a simple `switch` statement that has three switch sections, each containing two statements. The second switch section contains the `case 2:` and `case 3:` labels.

A `switch` statement can include any number of switch sections, and each section can have one or more case labels, as shown in the following example. However, no two case labels may contain the same expression.

```
using System;

public class Example
{
   public static void Main()
   {
      Random rnd = new Random();
      int caseSwitch = rnd.Next(1,4);

      switch (caseSwitch)
      {
          case 1:
              Console.WriteLine("Case 1");
              break;
          case 2:
          case 3:
              Console.WriteLine($"Case {caseSwitch}");
              break;
          default:
              Console.WriteLine($"An unexpected value ({caseSwitch})");
              break;
      }
   }
}
// The example displays output like the following:
//       Case 1
```

Only one switch section in a switch statement executes. C# does not allow execution to continue from one switch section to the next. Because of this, the following code generates a compiler error, CS0163: "Control cannot fall through from one case label () to another."

```
switch (caseSwitch)
{
    // The following switch section causes an error.
    case 1:
        Console.WriteLine("Case 1...");
        // Add a break or other jump statement here.
    case 2:
        Console.WriteLine("... and/or Case 2");
        break;
}
```

This requirement is usually met by explicitly exiting the switch section by using a break, goto, or return statement. However, the following code is also valid, because it ensures that program control cannot fall through to the `default` switch section.

```
switch (caseSwitch)
{
    // The following switch section causes an error.
    case 1:
        Console.WriteLine("Case 1...");
        break;
    case 2:
    case 3:
        Console.WriteLine("... and/or Case 2");
        break;
    case 4:
        while (true)
           Console.WriteLine("Endless looping. . . .");
    default:
        Console.WriteLine("Default value...");
        break;
}
```

Execution of the statement list in the switch section with a case label that matches the match expression begins with the first statement and proceeds through the statement list, typically until a jump statement, such as a `break`, `goto case`, `goto label`, `return`, or `throw`, is reached. At that point, control is transferred outside the `switch` statement or to another case label. A `goto` statement, if it is used, must transfer control to a constant label. This restriction is necessary, since attempting to transfer control to a non-constant label can have undesirable side-effects, such transferring control to an unintended location in code or creating an endless loop.

Case labels

Each case label specifies a pattern to compare to the match expression (the `caseSwitch` variable in the previous examples). If they match, control is transferred to the switch section that contains the **first** matching case label. If no case label pattern matches the match expression, control is transferred to the section with the `default` case label, if there is one. If there is no `default` case, no statements in any switch section are executed, and control is transferred outside the `switch` statement.

For information on the `switch` statement and pattern matching, see the Pattern matching with the `switch` statement section.

Because C# 6 supports only the constant pattern and does not allow the repetition of constant values, case labels define mutually exclusive values, and only one pattern can match the match expression. As a result, the order in which `case` statements appear is unimportant.

In C# 7.0, however, because other patterns are supported, case labels need not define mutually exclusive values, and multiple patterns can match the match expression. Because only the statements in the switch section that contains the first matching pattern are executed, the order in which `case` statements appear is now important. If C# detects a switch section whose case statement or statements are equivalent to or are subsets of previous statements, it generates a compiler error, CS8120, "The switch case has already been handled by a previous case."

The following example illustrates a `switch` statement that uses a variety of non-mutually exclusive patterns. If you move the `case 0:` switch section so that it is no longer the first section in the `switch` statement, C# generates a compiler error because an integer whose value is zero is a subset of all integers, which is the pattern defined by the `case int val` statement.

```
using System;
```

```
using System.Collections.Generic;
using System.Linq;

public class Example
{
   public static void Main()
   {
      var values = new List<object>();
      for (int ctr = 0; ctr <= 7; ctr++) {
         if (ctr == 2)
            values.Add(DiceLibrary.Roll2());
         else if (ctr == 4)
            values.Add(DiceLibrary.Pass());
         else
            values.Add(DiceLibrary.Roll());
      }

      Console.WriteLine($"The sum of { values.Count } die is { DiceLibrary.DiceSum(values) }");
   }
}

public static class DiceLibrary
{
   // Random number generator to simulate dice rolls.
   static Random rnd = new Random();

   // Roll a single die.
   public static int Roll()
   {
      return rnd.Next(1, 7);
   }

   // Roll two dice.
   public static List<object> Roll2()
   {
      var rolls = new List<object>();
      rolls.Add(Roll());
      rolls.Add(Roll());
      return rolls;
   }

   // Calculate the sum of n dice rolls.
   public static int DiceSum(IEnumerable<object> values)
   {
      var sum = 0;
      foreach (var item in values)
      {
            switch (item)
            {
               // A single zero value.
               case 0:
                  break;
               // A single value.
               case int val:
                  sum += val;
                  break;
               // A non-empty collection.
               case IEnumerable<object> subList when subList.Any():
                  sum += DiceSum(subList);
                  break;
               // An empty collection.
               case IEnumerable<object> subList:
                  break;
               //  A null reference.
               case null:
                  break;
               // A value that is neither an integer nor a collection.
               default:
                  throw new InvalidOperationException("unknown item type");
```

```
            }
        }
        return sum;
    }

    public static object Pass()
    {
        if (rnd.Next(0, 2) == 0)
            return null;
        else
            return new List<object>();
    }
}
```

You can correct this issue and eliminate the compiler warning in one of two ways:

- By changing the order of the switch sections.
- By using a when clause in the `case` label.

The `default` case

The `default` case specifies the switch section to execute if the match expression does not match any other `case` label. If a `default` case is not present and the match expression does not match any other `case` label, program flow falls through the `switch` statement.

The `default` case can appear in any order in the `switch` statement. Regardless of its order in the source code, it is always evaluated last, after all `case` labels have been evaluated.

Pattern matching with the `switch` statement

Each `case` statement defines a pattern that, if it matches the match expression, causes its containing switch section to be executed. All versions of C# support the constant pattern. The remaining patterns are supported beginning with C# 7.0.

Constant pattern

The constant pattern tests whether the match expression equals a specified constant. Its syntax is:

```
case constant:
```

where *constant* is the value to test for. *constant* can be any of the following constant expressions:

- A bool literal, either `true` or `false`.
- Any integral constant, such as an int, a long, or a byte.
- The name of a declared `const` variable.
- An enumeration constant.
- A char literal.
- A string literal.

The constant expression is evaluated as follows:

- If *expr* and *constant* are integral types, the C# equality operator determines whether the expression returns `true` (that is, whether `expr == constant`).
- Otherwise, the value of the expression is determined by a call to the static Object.Equals(expr, constant) method.

The following example uses the constant pattern to determine whether a particular date is a weekend, the first day of the work week, the last day of the work week, or the middle of the work week. It evaluates the DateTime.DayOfWeek property of the current day against the members of the DayOfWeek enumeration.

```
using System;

class Program
{
    static void Main()
    {
        switch (DateTime.Now.DayOfWeek)
        {
           case DayOfWeek.Sunday:
           case DayOfWeek.Saturday:
              Console.WriteLine("The weekend");
              break;
           case DayOfWeek.Monday:
              Console.WriteLine("The first day of the work week.");
              break;
           case DayOfWeek.Friday:
              Console.WriteLine("The last day of the work week.");
              break;
           default:
              Console.WriteLine("The middle of the work week.");
              break;
        }
    }
}
// The example displays output like the following:
//       The middle of the work week.
```

The following example uses the constant pattern to handle user input in a console application that simulates an automatic coffee machine.

```
using System;

class Example
{
   static void Main()
   {
       Console.WriteLine("Coffee sizes: 1=small 2=medium 3=large");
       Console.Write("Please enter your selection: ");
       string str = Console.ReadLine();
       int cost = 0;

       // Because of the goto statements in cases 2 and 3, the base cost of 25
       // cents is added to the additional cost for the medium and large sizes.
       switch (str)
       {
          case "1":
          case "small":
              cost += 25;
              break;
          case "2":
          case "medium":
              cost += 25;
              goto case "1";
          case "3":
          case "large":
              cost += 50;
              goto case "1";
          default:
              Console.WriteLine("Invalid selection. Please select 1, 2, or 3.");
              break;
      }
      if (cost != 0)
      {
          Console.WriteLine("Please insert {0} cents.", cost);
      }
      Console.WriteLine("Thank you for your business.");
   }
}
// The example displays output like the following:
//         Coffee sizes: 1=small 2=medium 3=large
//         Please enter your selection: 2
//         Please insert 50 cents.
//         Thank you for your business.
```

Type pattern

The type pattern enables concise type evaluation and conversion. When used with the `switch` statement to perform pattern matching, it tests whether an expression can be converted to a specified type and, if it can be, casts it to a variable of that type. Its syntax is:

```
case type varname
```

where *type* is the name of the type to which the result of *expr* is to be converted, and *varname* is the object to which the result of *expr* is converted if the match succeeds.

The `case` expression is `true` if any of the following is true:

- *expr* is an instance of the same type as *type*.
- *expr* is an instance of a type that derives from *type*. In other words, the result of *expr* can be upcast to an instance of *type*.

- *expr* has a compile-time type that is a base class of *type*, and *expr* has a runtime type that is *type* or is derived from *type*. The *compile-time type* of a variable is the variable's type as defined in its type declaration. The *runtime type* of a variable is the type of the instance that is assigned to that variable.
- *expr* is an instance of a type that implements the *type* interface.

If the case expression is true, *varname* is definitely assigned and has local scope within the switch section only.

Note that `null` does not match a type. To match a `null`, you use the following `case` label:

```
case null:
```

The following example uses the type pattern to provide information about various kinds of collection types.

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;

class Example
{
    static void Main(string[] args)
    {
        int[] values = { 2, 4, 6, 8, 10 };
        ShowCollectionInformation(values);

        var names = new List<string>();
        names.AddRange( new string[] { "Adam", "Abigail", "Bertrand", "Bridgette" } );
        ShowCollectionInformation(names);

        List<int> numbers = null;
        ShowCollectionInformation(numbers);
    }

    private static void ShowCollectionInformation(object coll)
    {
        switch (coll)
        {
            case Array arr:
               Console.WriteLine($"An array with {arr.Length} elements.");
               break;
            case IEnumerable<int> ieInt:
               Console.WriteLine($"Average: {ieInt.Average(s => s)}");
               break;
            case IList list:
               Console.WriteLine($"{list.Count} items");
               break;
            case IEnumerable ie:
               string result = "";
               foreach (var item in ie)
                  result += "${e} ";
               Console.WriteLine(result);
               break;
            case null:
               // Do nothing for a null.
               break;
            default:
               Console.WriteLine($"A instance of type {coll.GetType().Name}");
               break;
        }
    }
}
// The example displays the following output:
//     An array with 5 elements.
//     4 items
```

Without pattern matching, this code might be written as follows. The use of type pattern matching produces more compact, readable code by eliminating the need to test whether the result of a conversion is a `null` or to perform repeated casts.

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;

class Example
{
    static void Main(string[] args)
    {
        int[] values = { 2, 4, 6, 8, 10 };
        ShowCollectionInformation(values);

        var names = new List<string>();
        names.AddRange( new string[] { "Adam", "Abigail", "Bertrand", "Bridgette" } );
        ShowCollectionInformation(names);

        List<int> numbers = null;
        ShowCollectionInformation(numbers);
    }

    private static void ShowCollectionInformation(object coll)
    {
        if (coll is Array) {
           Array arr = (Array) coll;
           Console.WriteLine($"An array with {arr.Length} elements.");
        }
        else if (coll is IEnumerable<int>) {
            IEnumerable<int> ieInt = (IEnumerable<int>) coll;
            Console.WriteLine($"Average: {ieInt.Average(s => s)}");
        }
        else if (coll is IList) {
            IList list = (IList) coll;
            Console.WriteLine($"{list.Count} items");
        }
        else if (coll is IEnumerable) {
            IEnumerable ie = (IEnumerable) coll;
            string result = "";
            foreach (var item in ie)
               result += "${e} ";
            Console.WriteLine(result);
        }
        else if (coll == null) {
            // Do nothing.
        }
        else {
            Console.WriteLine($"An instance of type {coll.GetType().Name}");
        }
    }
}
// The example displays the following output:
//     An array with 5 elements.
//     4 items
```

The `case` statement and the `when` clause

Starting with C# 7.0, because case statements need not be mutually exclusive, you can add a `when` clause to specify an additional condition that must be satisfied for the case statement to evaluate to true. The `when` clause can be any expression that returns a Boolean value.

The following example defines a base `Shape` class, a `Rectangle` class that derives from `Shape`, and a `Square` class that derives from `Rectangle`. It uses the `when` clause to ensure that the `ShowShapeInfo` treats a `Rectangle` object that has been assigned equal lengths and widths as a `Square` even if it has not been instantiated as a `Square` object. The method does not attempt to display information either about an object that is `null` or a shape whose area is zero.

```
using System;

public abstract class Shape
{
   public abstract double Area { get; }
   public abstract double Circumference { get; }
}

public class Rectangle : Shape
{
   public Rectangle(double length, double width)
   {
      Length = length;
      Width = width;
   }

   public double Length { get; set; }
   public double Width { get; set; }

   public override double Area
   {
      get { return Math.Round(Length * Width,2); }
   }

   public override double Circumference
   {
      get { return (Length + Width) * 2; }
   }
}

public class Square : Rectangle
{
   public Square(double side) : base(side, side)
   {
      Side = side;
   }

   public double Side { get; set; }
}

public class Circle : Shape
{
   public Circle(double radius)
   {
      Radius = radius;
   }

   public double Radius { get; set; }

   public override double Circumference
   {
      get { return 2 * Math.PI * Radius; }
   }

   public override double Area
   {
      get { return Math.PI * Math.Pow(Radius, 2); }
   }
}

public class Example
{
   public static void Main()
   {
      Shape sh = null;
      Shape[] shapes = { new Square(10), new Rectangle(5, 7),
```

```
                         sh, new Square(0), new Rectangle(8, 8),
                         new Circle(3) };
      foreach (var shape in shapes)
         ShowShapeInfo(shape);
   }

   private static void ShowShapeInfo(Shape sh)
   {
      switch (sh)
      {
         // Note that this code never evaluates to true.
         case Shape shape when shape == null:
            Console.WriteLine($"An uninitialized shape (shape == null)");
            break;
         case null:
            Console.WriteLine($"An uninitialized shape");
            break;
         case Shape shape when sh.Area == 0:
            Console.WriteLine($"The shape: {sh.GetType().Name} with no dimensions");
            break;
         case Square sq when sh.Area > 0:
            Console.WriteLine("Information about square:");
            Console.WriteLine($"   Length of a side: {sq.Side}");
            Console.WriteLine($"   Area: {sq.Area}");
            break;
         case Rectangle r when r.Length == r.Width && r.Area > 0:
            Console.WriteLine("Information about square rectangle:");
            Console.WriteLine($"   Length of a side: {r.Length}");
            Console.WriteLine($"   Area: {r.Area}");
            break;
         case Rectangle r when sh.Area > 0:
            Console.WriteLine("Information about rectangle:");
            Console.WriteLine($"   Dimensions: {r.Length} x {r.Width}");
            Console.WriteLine($"   Area: {r.Area}");
            break;
         case Shape shape when sh != null:
            Console.WriteLine($"A {sh.GetType().Name} shape");
            break;
         default:
            Console.WriteLine($"The {nameof(sh)} variable does not represent a Shape.");
            break;
      }
   }
}
// The example displays the following output:
//       Information about square:
//          Length of a side: 10
//          Area: 100
//       Information about rectangle:
//          Dimensions: 5 x 7
//          Area: 35
//       An uninitialized shape
//       The shape: Square with no dimensions
//       Information about square rectangle:
//          Length of a side: 8
//          Area: 64
//       A Circle shape
```

Note that the `when` clause in the example that attempts to test whether a `Shape` object is `null` does not execute. The correct type pattern to test for a `null` is `case null:`.

C# language specification

For more information, see The switch statement in the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- C# Reference
- C# Programming Guide
- C# Keywords
- if-else
- Pattern Matching

Iteration Statements (C# Reference)

8/25/2018 • 2 minutes to read • Edit Online

You can create loops by using the iteration statements. Iteration statements cause embedded statements to be executed a number of times, subject to the loop-termination criteria. These statements are executed in order, except when a jump statement is encountered.

The following keywords are used in iteration statements:

- do
- for
- foreach, in
- while

See also

- C# Reference
- C# Programming Guide
- C# Keywords
- Statement Keywords

do (C# Reference)

8/25/2018 • 2 minutes to read • Edit Online

The `do` statement executes a statement or a block of statements while a specified boolean expression evaluates to `true`. Because that expression is evaluated after each execution of the loop, a `do-while` loop executes one or more times. This differs from the while loop, which executes zero or more times.

At any point within the `do` statement block, you can break out of the loop by using the break statement.

You can step directly to the evaluation of the `while` expression by using the continue statement. If the expression evaluates to `true`, execution continues at the first statement in the loop. Otherwise, execution continues at the first statement after the loop.

You also can exit a `do-while` loop by the goto, return, or throw statements.

Example

The following example shows the usage of the `do` statement. Select **Run** to run the example code. After that you can modify the code and run it again.

```
int n = 0;
do
{
    Console.WriteLine(n);
    n++;
} while (n < 5);
```

C# language specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- C# Reference
- C# Programming Guide
- C# Keywords
- do-while Statement (C++)
- Iteration Statements
- while statement

for (C# reference)

8/15/2018 • 2 minutes to read • Edit Online

The `for` statement executes a statement or a block of statements while a specified boolean expression evaluates to `true`.

At any point within the `for` statement block, you can break out of the loop by using the break statement, or step to the next iteration in the loop by using the continue statement. You also can exit a `for` loop by the goto, return, or throw statements.

Structure of the `for` statement

The `for` statement defines *initializer*, *condition*, and *iterator* sections:

```
for (initializer; condition; iterator)
    body
```

All three sections are optional. The body of the loop is either a statement or a block of statements.

The following example shows the `for` statement with all of the sections defined:

```
for (int i = 0; i < 5; i++)
{
    Console.WriteLine(i);
}
```

The *initializer* section

The statements in the *initializer* section are executed only once, before entering the loop. The *initializer* section is either of the following:

- The declaration and initialization of a local loop variable, which can't be accessed from outside the loop.
- Zero or more statement expressions from the following list, separated by commas:
    - assignment statement
    - invocation of a method
    - prefix or postfix increment expression, such as `++i` or `i++`
    - prefix or postfix decrement expression, such as `--i` or `i--`
    - creation of an object by using new keyword
    - await expression

The *initializer* section in the example above declares and initializes the local loop variable `i`:

```
int i = 0
```

The *condition* section

The *condition* section, if present, must be a boolean expression. That expression is evaluated before every loop iteration. If the *condition* section is not present or the boolean expression evaluates to `true`, the next loop iteration is executed; otherwise, the loop is exited.

The *condition* section in the example above determines if the loop terminates based on the value of the local loop variable:

```
i < 5
```

The *iterator* section

The *iterator* section defines what happens after each iteration of the body of the loop. The *iterator* section contains zero or more of the following statement expressions, separated by commas:

- assignment statement
- invocation of a method
- prefix or postfix increment expression, such as `++i` or `i++`
- prefix or postfix decrement expression, such as `--i` or `i--`
- creation of an object by using new keyword
- await expression

The *iterator* section in the example above increments the local loop variable:

```
i++
```

Examples

The following example illustrates several less common usages of the `for` statement sections: assigning a value to an external loop variable in the *initializer* section, invoking a method in both the *initializer* and the *iterator* sections, and changing the values of two variables in the *iterator* section. Select **Run** to run the example code. After that you can modify the code and run it again.

```
int i;
int j = 10;
for (i = 0, Console.WriteLine($"Start: i={i}, j={j}"); i < j; i++, j--, Console.WriteLine($"Step: i={i}, j=
{j}"))
{
    // Body of the loop.
}
```

The following example defines the infinite `for` loop:

```
for ( ; ; )
{
    // Body of the loop.
}
```

C# language specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- The for statement (C# language specification)
- C# Reference
- C# Programming Guide
- C# Keywords
- foreach, in
- for Statement (C++)
- Iteration Statements

foreach, in (C# reference)

8/15/2018 • 2 minutes to read • Edit Online

The `foreach` statement executes a statement or a block of statements for each element in an instance of the type that implements the System.Collections.IEnumerable or System.Collections.Generic.IEnumerable<T> interface. The `foreach` statement is not limited to those types and can be applied to an instance of any type that satisfies the following conditions:

- has the public parameterless `GetEnumerator` method whose return type is either class, struct, or interface type,
- the return type of the `GetEnumerator` method has the public `Current` property and the public parameterless `MoveNext` method whose return type is Boolean.

At any point within the `foreach` statement block, you can break out of the loop by using the break statement, or step to the next iteration in the loop by using the continue statement. You also can exit a `foreach` loop by the goto, return, or throw statements.

Examples

> **NOTE**
>
> The C# examples in this article run in the Try.NET inline code runner and playground. Select the **Run** button to run an example in an interactive window. Once you execute the code, you can modify it and run the modified code by selecting **Run** again. The modified code either runs in the interactive window or, if compilation fails, the interactive window displays all C# compiler error messages.

The following example shows usage of the `foreach` statement with an instance of the List<T> type that implements the IEnumerable<T> interface:

```
var fibNumbers = new List<int> { 0, 1, 1, 2, 3, 5, 8, 13 };
int count = 0;
foreach (int element in fibNumbers)
{
    count++;
    Console.WriteLine($"Element #{count}: {element}");
}
Console.WriteLine($"Number of elements: {count}");
```

The next example uses the `foreach` statement with an instance of the System.Span<T> type, which doesn't implement any interfaces:

```
public class IterateSpanExample
{
    public static void Main()
    {
        Span<int> numbers = new int[] { 3, 14, 15, 92, 6 };
        foreach (int number in numbers)
        {
            Console.Write($"{number} ");
        }
        Console.WriteLine();
    }
}
```

Beginning with C# 7.3, if the enumerator's `Current` property returns a reference return value (`ref T` where `T` is the type of the collection element), you can declare the iteration variable with the `ref` or `ref readonly` modifier. The following example uses a `ref` iteration variable to set the value of each item in a stackalloc array. The `ref readonly` version iterates the collection to print all the values. The `readonly` declaration uses an implicit local variable declaration. Implicit variable declarations can be used with either `ref` or `ref readonly` declarations, as can explicitly typed variable declarations.

```
public class ForeachRefExample
{
    public static void Main()
    {
        Span<int> storage = stackalloc int[10];
        int num = 0;
        foreach (ref int item in storage)
        {
            item = num++;
        }

        foreach (ref readonly var item in storage)
        {
            Console.Write($"{item} ");
        }
        // Output:
        // 0 1 2 3 4 5 6 7 8 9
    }
}
```

C# language specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- The foreach statement (C# language specification)
- Using foreach with Arrays
- for
- Iteration Statements
- C# Keywords
- C# Reference
- C# Programming Guide

while (C# Reference)

8/25/2018 • 2 minutes to read • Edit Online

The `while` statement executes a statement or a block of statements while a specified boolean expression evaluates to `true`. Because that expression is evaluated before each execution of the loop, a `while` loop executes zero or more times. This differs from the do loop, which executes one or more times.

At any point within the `while` statement block, you can break out of the loop by using the break statement.

You can step directly to the evaluation of the `while` expression by using the continue statement. If the expression evaluates to `true`, execution continues at the first statement in the loop. Otherwise, execution continues at the first statement after the loop.

You also can exit a `while` loop by the goto, return, or throw statements.

Example

The following example shows the usage of the `while` statement. Select **Run** to run the example code. After that you can modify the code and run it again.

```
int n = 0;
while (n < 5)
{
    Console.WriteLine(n);
    n++;
}
```

C# language specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- C# Reference
- C# Programming Guide
- C# Keywords
- while Statement (C++)
- Iteration Statements
- do statement

Jump Statements (C# Reference)

8/25/2018 • 2 minutes to read • Edit Online

Branching is performed using jump statements, which cause an immediate transfer of the program control. The following keywords are used in jump statements:

- break
- continue
- goto
- return
- throw

See Also

- C# Reference
- C# Programming Guide
- C# Keywords
- Statement Keywords

break (C# Reference)

8/25/2018 • 2 minutes to read • Edit Online

The `break` statement terminates the closest enclosing loop or switch statement in which it appears. Control is passed to the statement that follows the terminated statement, if any.

Example

In this example, the conditional statement contains a counter that is supposed to count from 1 to 100; however, the `break` statement terminates the loop after 4 counts.

```
class BreakTest
{
    static void Main()
    {
        for (int i = 1; i <= 100; i++)
        {
            if (i == 5)
            {
                break;
            }
            Console.WriteLine(i);
        }

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/*
 Output:
    1
    2
    3
    4
*/
```

Example

In this example, the `break` statement is used to break out of an inner nested loop, and return control to the outer loop.

```
class BreakInNestedLoops
{
    static void Main(string[] args)
    {

        int[] numbers = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
        char[] letters = { 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j' };

        // Outer loop
        for (int x = 0; x < numbers.Length; x++)
        {
            Console.WriteLine("num = {0}", numbers[x]);

            // Inner loop
            for (int y = 0; y < letters.Length; y++)
            {
                if (y == x)
                {
                    // Return control to outer loop
                    break;
                }
                Console.Write(" {0} ", letters[y]);
            }
            Console.WriteLine();
        }

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}

/*
 * Output:
    num = 0

    num = 1
     a
    num = 2
     a  b
    num = 3
     a  b  c
    num = 4
     a  b  c  d
    num = 5
     a  b  c  d  e
    num = 6
     a  b  c  d  e  f
    num = 7
     a  b  c  d  e  f  g
    num = 8
     a  b  c  d  e  f  g  h
    num = 9
     a  b  c  d  e  f  g  h  i
 */
```

Example

This example demonstrates the use of `break` in a switch statement.

```
class Switch
{
    static void Main()
    {
        Console.Write("Enter your selection (1, 2, or 3): ");
        string s = Console.ReadLine();
        int n = Int32.Parse(s);

        switch (n)
        {
            case 1:
                Console.WriteLine("Current value is {0}", 1);
                break;
            case 2:
                Console.WriteLine("Current value is {0}", 2);
                break;
            case 3:
                Console.WriteLine("Current value is {0}", 3);
                break;
            default:
                Console.WriteLine("Sorry, invalid selection.");
                break;
        }

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/*
Sample Input: 1

Sample Output:
Enter your selection (1, 2, or 3): 1
Current value is 1
*/
```

If you entered `4`, the output would be:

```
Enter your selection (1, 2, or 3): 4
Sorry, invalid selection.
```

C# language specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See Also

- C# Reference
- C# Programming Guide
- C# Keywords
- switch
- Jump Statements
- Iteration Statements

continue (C# Reference)

8/25/2018 • 2 minutes to read • Edit Online

The `continue` statement passes control to the next iteration of the enclosing while, do, for, or foreach statement in which it appears.

Example

In this example, a counter is initialized to count from 1 to 10. By using the `continue` statement in conjunction with the expression `(i < 9)`, the statements between `continue` and the end of the `for` body are skipped.

```
class ContinueTest
{
    static void Main()
    {
        for (int i = 1; i <= 10; i++)
        {
            if (i < 9)
            {
                continue;
            }
            Console.WriteLine(i);
        }

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/*
Output:
9
10
*/
```

C# language specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- C# Reference
- C# Programming Guide
- C# Keywords
- break Statement
- Jump Statements

goto (C# Reference)

8/25/2018 • 2 minutes to read • Edit Online

The `goto` statement transfers the program control directly to a labeled statement.

A common use of `goto` is to transfer control to a specific switch-case label or the default label in a `switch` statement.

The `goto` statement is also useful to get out of deeply nested loops.

Example

The following example demonstrates using `goto` in a switch statement.

```
class SwitchTest
{
    static void Main()
    {
        Console.WriteLine("Coffee sizes: 1=Small 2=Medium 3=Large");
        Console.Write("Please enter your selection: ");
        string s = Console.ReadLine();
        int n = int.Parse(s);
        int cost = 0;
        switch (n)
        {
            case 1:
                cost += 25;
                break;
            case 2:
                cost += 25;
                goto case 1;
            case 3:
                cost += 50;
                goto case 1;
            default:
                Console.WriteLine("Invalid selection.");
                break;
        }
        if (cost != 0)
        {
            Console.WriteLine("Please insert {0} cents.", cost);
        }
        Console.WriteLine("Thank you for your business.");

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/*
Sample Input:  2

Sample Output:
Coffee sizes: 1=Small 2=Medium 3=Large
Please enter your selection: 2
Please insert 50 cents.
Thank you for your business.
*/
```

Example

The following example demonstrates using `goto` to break out from nested loops.

```
public class GotoTest1
{
    static void Main()
    {
        int x = 200, y = 4;
        int count = 0;
        string[,] array = new string[x, y];

        // Initialize the array:
        for (int i = 0; i < x; i++)

            for (int j = 0; j < y; j++)
                array[i, j] = (++count).ToString();

        // Read input:
        Console.Write("Enter the number to search for: ");

        // Input a string:
        string myNumber = Console.ReadLine();

        // Search:
        for (int i = 0; i < x; i++)
        {
            for (int j = 0; j < y; j++)
            {
                if (array[i, j].Equals(myNumber))
                {
                    goto Found;
                }
            }
        }

        Console.WriteLine("The number {0} was not found.", myNumber);
        goto Finish;

    Found:
        Console.WriteLine("The number {0} is found.", myNumber);

    Finish:
        Console.WriteLine("End of search.");


        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/*
Sample Input: 44

Sample Output
Enter the number to search for: 44
The number 44 is found.
End of search.
*/
```

C# language specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- C# Reference
- C# Programming Guide
- C# Keywords
- goto Statement (C++)
- Jump Statements

return (C# Reference)

8/29/2018 • 2 minutes to read • Edit Online

The `return` statement terminates execution of the method in which it appears and returns control to the calling method. It can also return an optional value. If the method is a `void` type, the `return` statement can be omitted.

If the return statement is inside a `try` block, the `finally` block, if one exists, will be executed before control returns to the calling method.

Example

In the following example, the method `CalculateArea()` returns the local variable `area` as a double value.

```
class ReturnTest
{
    static double CalculateArea(int r)
    {
        double area = r * r * Math.PI;
        return area;
    }

    static void Main()
    {
        int radius = 5;
        double result = CalculateArea(radius);
        Console.WriteLine("The area is {0:0.00}", result);

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
// Output: The area is 78.54
```

C# language specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- C# Reference
- C# Programming Guide
- C# Keywords
- return Statement
- Jump Statements

Exception Handling Statements (C# Reference)

8/25/2018 • 2 minutes to read • Edit Online

C# provides built-in support for handling anomalous situations, known as exceptions, which may occur during the execution of your program. These exceptions are handled by code that is outside the normal flow of control.

The following exception handling topics are explained in this section:

- throw
- try-catch
- try-finally
- try-catch-finally

See Also

- C# Reference
- C# Programming Guide
- C# Keywords
- Statement Keywords
- Exceptions and Exception Handling

throw (C# Reference)

8/25/2018 • 3 minutes to read • Edit Online

Signals the occurrence of an exception during program execution.

Remarks

The syntax of `throw` is:

```
throw [e]
```

where `e` is an instance of a class derived from System.Exception. The following example uses the `throw` statement to throw an IndexOutOfRangeException if the argument passed to a method named `GetNumber` does not correspond to a valid index of an internal array.

```
using System;

public class NumberGenerator
{
   int[] numbers = { 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 };

   public int GetNumber(int index)
   {
      if (index < 0 || index >= numbers.Length) {
         throw new IndexOutOfRangeException();
      }
      return numbers[index];
   }
}
```

Method callers then use a `try-catch` or `try-catch-finally` block to handle the thrown exception. The following example handles the exception thrown by the `GetNumber` method.

```
using System;

public class Example
{
   public static void Main()
   {
      var gen = new NumberGenerator();
      int index = 10;
      try {
          int value = gen.GetNumber(index);
          Console.WriteLine($"Retrieved {value}");
      }
      catch (IndexOutOfRangeException e)
      {
         Console.WriteLine($"{e.GetType().Name}: {index} is outside the bounds of the array");
      }
   }
}
// The example displays the following output:
//        IndexOutOfRangeException: 10 is outside the bounds of the array
```

Re-throwing an exception

`throw` can also be used in a `catch` block to re-throw an exception handled in a `catch` block. In this case, `throw` does not take an exception operand. It is most useful when a method passes on an argument from a caller to some other library method, and the library method throws an exception that must be passed on to the caller. For example, the following example re-throws an NullReferenceException that is thrown when attempting to retrieve the first character of an uninitialized string.

```
using System;

public class Sentence
{
   public Sentence(string s)
   {
      Value = s;
   }

   public string Value { get; set; }

   public char GetFirstCharacter()
   {
      try {
         return Value[0];
        }
      catch (NullReferenceException e) {
         throw;
      }
   }
}

public class Example
{
   public static void Main()
   {
      var s = new Sentence(null);
      Console.WriteLine($"The first character is {s.GetFirstCharacter()}");
   }
}
// The example displays the following output:
//    Unhandled Exception: System.NullReferenceException: Object reference not set to an instance of an
object.
//       at Sentence.GetFirstCharacter()
//       at Example.Main()
```

IMPORTANT

You can also use the `throw e` syntax in a `catch` block to instantiate a new exception that you pass on to the caller. In this case, the stack trace of the original exception, which is available from the StackTrace property, is not preserved.

The `throw` expression

Starting with C# 7.0, `throw` can be used as an expression as well as a statement. This allows an exception to be thrown in contexts that were previously unsupported. These include:

- the conditional operator. The following example uses a `throw` expression to throw an ArgumentException if a method is passed an empty string array. Before C# 7.0, this logic would need to appear in an `if` / `else` statement.

```
private static void DisplayFirstNumber(string[] args)
{
   string arg = args.Length >= 1 ? args[0] :
                              throw new ArgumentException("You must supply an argument");
   if (Int64.TryParse(arg, out var number))
      Console.WriteLine($"You entered {number:F0}");
   else
      Console.WriteLine($"{arg} is not a number.");

}
```

- the null-coalescing operator. In the following example, a `throw` expression is used with a null-coalescing operator to throw an exception if the string assigned to a `Name` property is `null`.

```
public string Name
{
    get => name;
    set => name = value ??
        throw new ArgumentNullException("Name cannot be null", nameof(value));
}
```

- an expression-bodied lambda or method. The following example illustrates an expression-bodied method that throws an InvalidCastException because a conversion to a DateTime value is not supported.

```
DateTime ToDateTime(IFormatProvider provider) =>
         throw new InvalidCastException("Conversion to a DateTime is not supported.");
```

C# Language Specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See Also

- C# Reference
- C# Programming Guide
- try-catch
- The try, catch, and throw Statements in C++
- C# Keywords
- Exception Handling Statements
- How to: Explicitly Throw Exceptions

try-catch (C# Reference)

8/25/2018 • 8 minutes to read • Edit Online

The try-catch statement consists of a `try` block followed by one or more `catch` clauses, which specify handlers for different exceptions.

Remarks

When an exception is thrown, the common language runtime (CLR) looks for the `catch` statement that handles this exception. If the currently executing method does not contain such a `catch` block, the CLR looks at the method that called the current method, and so on up the call stack. If no `catch` block is found, then the CLR displays an unhandled exception message to the user and stops execution of the program.

The `try` block contains the guarded code that may cause the exception. The block is executed until an exception is thrown or it is completed successfully. For example, the following attempt to cast a `null` object raises the NullReferenceException exception:

```
object o2 = null;
try
{
    int i2 = (int)o2;   // Error
}
```

Although the `catch` clause can be used without arguments to catch any type of exception, this usage is not recommended. In general, you should only catch those exceptions that you know how to recover from. Therefore, you should always specify an object argument derived from System.Exception For example:

```
catch (InvalidCastException e)
{
}
```

It is possible to use more than one specific `catch` clause in the same try-catch statement. In this case, the order of the `catch` clauses is important because the `catch` clauses are examined in order. Catch the more specific exceptions before the less specific ones. The compiler produces an error if you order your catch blocks so that a later block can never be reached.

Using `catch` arguments is one way to filter for the exceptions you want to handle. You can also use a exception filter that further examines the exception to decide whether to handle it. If the exception filter returns false, then the search for a handler continues.

```
catch (ArgumentException e) when (e.ParamName == "…")
{
}
```

Exception filters are preferable to catching and rethrowing (explained below) because filters leave the stack unharmed. If a later handler dumps the stack, you can see where the exception originally came from, rather than just the last place it was rethrown. A common use of exception filter expressions is logging. You can create a filter that always returns false that also outputs to a log, you can log exceptions as they go by without having to handle them and rethrow.

A throw statement can be used in a `catch` block to re-throw the exception that is caught by the `catch` statement. The following example extracts source information from an IOException exception, and then throws the exception to the parent method.

```
catch (FileNotFoundException e)
{
    // FileNotFoundExceptions are handled here.
}
catch (IOException e)
{
    // Extract some information from this exception, and then
    // throw it to the parent method.
    if (e.Source != null)
        Console.WriteLine("IOException source: {0}", e.Source);
    throw;
}
```

You can catch one exception and throw a different exception. When you do this, specify the exception that you caught as the inner exception, as shown in the following example.

```
catch (InvalidCastException e)
{
    // Perform some action here, and then throw a new exception.
    throw new YourCustomException("Put your error message here.", e);
}
```

You can also re-throw an exception when a specified condition is true, as shown in the following example.

```
catch (InvalidCastException e)
{
    if (e.Data == null)
    {
        throw;
    }
    else
    {
        // Take some action.
    }
}
```

NOTE

It is also possible to use an exception filter to get a similar result in an often cleaner fashion (as well as not modifying the stack, as explained earlier in this document). The following example has a similar behavior for callers as the previous example. The function throws the `InvalidCastException` back to the caller when `e.Data` is `null`.

```
catch (InvalidCastException e) when (e.Data != null)
{
    // Take some action.
}
```

From inside a `try` block, initialize only variables that are declared therein. Otherwise, an exception can occur before the execution of the block is completed. For example, in the following code example, the variable `n` is initialized inside the `try` block. An attempt to use this variable outside the `try` block in the `Write(n)` statement will generate a compiler error.

```
static void Main()
{
    int n;
    try
    {
        // Do not initialize this variable here.
        n = 123;
    }
    catch
    {
    }
    // Error: Use of unassigned local variable 'n'.
    Console.Write(n);
}
```

For more information about catch, see try-catch-finally.

Exceptions in Async Methods

An async method is marked by an async modifier and usually contains one or more await expressions or statements. An await expression applies the await operator to a Task or Task<TResult>.

When control reaches an `await` in the async method, progress in the method is suspended until the awaited task completes. When the task is complete, execution can resume in the method. For more information, see Asynchronous Programming with async and await and Control Flow in Async Programs.

The completed task to which `await` is applied might be in a faulted state because of an unhandled exception in the method that returns the task. Awaiting the task throws an exception. A task can also end up in a canceled state if the asynchronous process that returns it is canceled. Awaiting a canceled task throws an `OperationCanceledException`. For more information about how to cancel an asynchronous process, see Fine-Tuning Your Async Application.

To catch the exception, await the task in a `try` block, and catch the exception in the associated `catch` block. For an example, see the "Example" section.

A task can be in a faulted state because multiple exceptions occurred in the awaited async method. For example, the task might be the result of a call to Task.WhenAll. When you await such a task, only one of the exceptions is caught, and you can't predict which exception will be caught. For an example, see the "Example" section.

Example

In the following example, the `try` block contains a call to the `ProcessString` method that may cause an exception. The `catch` clause contains the exception handler that just displays a message on the screen. When the `throw` statement is called from inside `MyMethod`, the system looks for the `catch` statement and displays the message `Exception caught`.

```
    class TryFinallyTest
{
    static void ProcessString(string s)
    {
        if (s == null)
        {
            throw new ArgumentNullException();
        }
    }

    static void Main()
    {
        string s = null; // For demonstration purposes.

        try
        {
            ProcessString(s);
        }

        catch (Exception e)
        {
            Console.WriteLine("{0} Exception caught.", e);
        }
    }
}
    /*
    Output:
    System.ArgumentNullException: Value cannot be null.
       at TryFinallyTest.Main() Exception caught.
     * */
```

Example

In the following example, two catch blocks are used, and the most specific exception, which comes first, is caught.

To catch the least specific exception, you can replace the throw statement in `ProcessString` with the following statement: `throw new Exception()`.

If you place the least-specific catch block first in the example, the following error message appears: `A previous catch clause already catches all exceptions of this or a super type ('System.Exception')`.

```
class ThrowTest3
{
    static void ProcessString(string s)
    {
        if (s == null)
        {
            throw new ArgumentNullException();
        }
    }

    static void Main()
    {
        try
        {
            string s = null;
            ProcessString(s);
        }
        // Most specific:
        catch (ArgumentNullException e)
        {
            Console.WriteLine("{0} First exception caught.", e);
        }
        // Least specific:
        catch (Exception e)
        {
            Console.WriteLine("{0} Second exception caught.", e);
        }
    }
}
/*
 Output:
 System.ArgumentNullException: Value cannot be null.
 at Test.ThrowTest3.ProcessString(String s) ... First exception caught.
*/
```

Example

The following example illustrates exception handling for async methods. To catch an exception that an async task throws, place the `await` expression in a `try` block, and catch the exception in a `catch` block.

Uncomment the `throw new Exception` line in the example to demonstrate exception handling. The task's `IsFaulted` property is set to `True`, the task's `Exception.InnerException` property is set to the exception, and the exception is caught in the `catch` block.

Uncomment the `throw new OperationCancelledException` line to demonstrate what happens when you cancel an asynchronous process. The task's `IsCanceled` property is set to `true`, and the exception is caught in the `catch` block. Under some conditions that don't apply to this example, the task's `IsFaulted` property is set to `true` and `IsCanceled` is set to `false`.

```csharp
public async Task DoSomethingAsync()
{
    Task<string> theTask = DelayAsync();

    try
    {
        string result = await theTask;
        Debug.WriteLine("Result: " + result);
    }
    catch (Exception ex)
    {
        Debug.WriteLine("Exception Message: " + ex.Message);
    }
    Debug.WriteLine("Task IsCanceled: " + theTask.IsCanceled);
    Debug.WriteLine("Task IsFaulted:  " + theTask.IsFaulted);
    if (theTask.Exception != null)
    {
        Debug.WriteLine("Task Exception Message: "
            + theTask.Exception.Message);
        Debug.WriteLine("Task Inner Exception Message: "
            + theTask.Exception.InnerException.Message);
    }
}

private async Task<string> DelayAsync()
{
    await Task.Delay(100);

    // Uncomment each of the following lines to
    // demonstrate exception handling.

    //throw new OperationCanceledException("canceled");
    //throw new Exception("Something happened.");
    return "Done";
}

// Output when no exception is thrown in the awaited method:
//   Result: Done
//   Task IsCanceled: False
//   Task IsFaulted:  False

// Output when an Exception is thrown in the awaited method:
//   Exception Message: Something happened.
//   Task IsCanceled: False
//   Task IsFaulted:  True
//   Task Exception Message: One or more errors occurred.
//   Task Inner Exception Message: Something happened.

// Output when a OperationCanceledException or TaskCanceledException
// is thrown in the awaited method:
//   Exception Message: canceled
//   Task IsCanceled: True
//   Task IsFaulted:  False
```

Example

The following example illustrates exception handling where multiple tasks can result in multiple exceptions. The `try` block awaits the task that's returned by a call to Task.WhenAll. The task is complete when the three tasks to which WhenAll is applied are complete.

Each of the three tasks causes an exception. The `catch` block iterates through the exceptions, which are found in the `Exception.InnerExceptions` property of the task that was returned by Task.WhenAll.

```csharp
public async Task DoMultipleAsync()
{
    Task theTask1 = ExcAsync(info: "First Task");
    Task theTask2 = ExcAsync(info: "Second Task");
    Task theTask3 = ExcAsync(info: "Third Task");

    Task allTasks = Task.WhenAll(theTask1, theTask2, theTask3);

    try
    {
        await allTasks;
    }
    catch (Exception ex)
    {
        Debug.WriteLine("Exception: " + ex.Message);
        Debug.WriteLine("Task IsFaulted: " + allTasks.IsFaulted);
        foreach (var inEx in allTasks.Exception.InnerExceptions)
        {
            Debug.WriteLine("Task Inner Exception: " + inEx.Message);
        }
    }
}

private async Task ExcAsync(string info)
{
    await Task.Delay(100);

    throw new Exception("Error-" + info);
}

// Output:
//   Exception: Error-First Task
//   Task IsFaulted: True
//   Task Inner Exception: Error-First Task
//   Task Inner Exception: Error-Second Task
//   Task Inner Exception: Error-Third Task
```

C# Language Specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See Also

- C# Reference
- C# Programming Guide
- C# Keywords
- try, throw, and catch Statements (C++)
- Exception Handling Statements
- throw
- try-finally
- How to: Explicitly Throw Exceptions

try-finally (C# Reference)

8/25/2018 • 3 minutes to read • Edit Online

By using a `finally` block, you can clean up any resources that are allocated in a try block, and you can run code even if an exception occurs in the `try` block. Typically, the statements of a `finally` block run when control leaves a `try` statement. The transfer of control can occur as a result of normal execution, of execution of a `break`, `continue`, `goto`, or `return` statement, or of propagation of an exception out of the `try` statement.

Within a handled exception, the associated `finally` block is guaranteed to be run. However, if the exception is unhandled, execution of the `finally` block is dependent on how the exception unwind operation is triggered. That, in turn, is dependent on how your computer is set up.

Usually, when an unhandled exception ends an application, whether or not the `finally` block is run is not important. However, if you have statements in a `finally` block that must be run even in that situation, one solution is to add a `catch` block to the `try`-`finally` statement. Alternatively, you can catch the exception that might be thrown in the `try` block of a `try`-`finally` statement higher up the call stack. That is, you can catch the exception in the method that calls the method that contains the `try`-`finally` statement, or in the method that calls that method, or in any method in the call stack. If the exception is not caught, execution of the `finally` block depends on whether the operating system chooses to trigger an exception unwind operation.

Example

In the following example, an invalid conversion statement causes a `System.InvalidCastException` exception. The exception is unhandled.

```
public class ThrowTestA
{
    static void Main()
    {
        int i = 123;
        string s = "Some string";
        object obj = s;

        try
        {
            // Invalid conversion; obj contains a string, not a numeric type.
            i = (int)obj;

            // The following statement is not run.
            Console.WriteLine("WriteLine at the end of the try block.");
        }
        finally
        {
            // To run the program in Visual Studio, type CTRL+F5. Then
            // click Cancel in the error dialog.
            Console.WriteLine("\nExecution of the finally block after an unhandled\n" +
                "error depends on how the exception unwind operation is triggered.");
            Console.WriteLine("i = {0}", i);
        }
    }
    // Output:
    // Unhandled Exception: System.InvalidCastException: Specified cast is not valid.
    //
    // Execution of the finally block after an unhandled
    // error depends on how the exception unwind operation is triggered.
    // i = 123
}
```

In the following example, an exception from the `TryCast` method is caught in a method farther up the call stack.

```
public class ThrowTestB
{
    static void Main()
    {
        try
        {
            // TryCast produces an unhandled exception.
            TryCast();
        }
        catch (Exception ex)
        {
            // Catch the exception that is unhandled in TryCast.
            Console.WriteLine
                ("Catching the {0} exception triggers the finally block.",
                ex.GetType());

            // Restore the original unhandled exception. You might not
            // know what exception to expect, or how to handle it, so pass
            // it on.
            throw;
        }
    }

    public static void TryCast()
    {
        int i = 123;
        string s = "Some string";
        object obj = s;

        try
        {
            // Invalid conversion; obj contains a string, not a numeric type.
            i = (int)obj;

            // The following statement is not run.
            Console.WriteLine("WriteLine at the end of the try block.");
        }
        finally
        {
            // Report that the finally block is run, and show that the value of
            // i has not been changed.
            Console.WriteLine("\nIn the finally block in TryCast, i = {0}.\n", i);
        }
    }
    // Output:
    // In the finally block in TryCast, i = 123.

    // Catching the System.InvalidCastException exception triggers the finally block.

    // Unhandled Exception: System.InvalidCastException: Specified cast is not valid.
}
```

For more information about `finally`, see try-catch-finally.

C# also contains the using statement, which provides similar functionality for IDisposable objects in a convenient syntax.

C# Language Specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See Also

- C# Reference
- C# Programming Guide
- C# Keywords
- try, throw, and catch Statements (C++)
- Exception Handling Statements
- throw
- try-catch
- How to: Explicitly Throw Exceptions

try-catch-finally (C# Reference)

8/25/2018 • 2 minutes to read • Edit Online

A common usage of `catch` and `finally` together is to obtain and use resources in a `try` block, deal with exceptional circumstances in a `catch` block, and release the resources in the `finally` block.

For more information and examples on re-throwing exceptions, see try-catch and Throwing Exceptions. For more information about the `finally` block, see try-finally.

Example

```
public class EHClass
{
    void ReadFile(int index)
    {
        // To run this code, substitute a valid path from your local machine
        string path = @"c:\users\public\test.txt";
        System.IO.StreamReader file = new System.IO.StreamReader(path);
        char[] buffer = new char[10];
        try
        {
            file.ReadBlock(buffer, index, buffer.Length);
        }
        catch (System.IO.IOException e)
        {
            Console.WriteLine("Error reading from {0}. Message = {1}", path, e.Message);
        }

        finally
        {
            if (file != null)
            {
                file.Close();
            }
        }
        // Do something with buffer...
    }

}
```

C# Language Specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See Also

- C# Reference
- C# Programming Guide
- C# Keywords
- try, throw, and catch Statements (C++)
- Exception Handling Statements
- throw

- How to: Explicitly Throw Exceptions
- using Statement

Checked and Unchecked (C# Reference)

8/25/2018 • 2 minutes to read • Edit Online

C# statements can execute in either checked or unchecked context. In a checked context, arithmetic overflow raises an exception. In an unchecked context, arithmetic overflow is ignored and the result is truncated by discarding any high-order bits that don't fit in the destination type.

- checked Specify checked context.
- unchecked Specify unchecked context.

The following operations are affected by the overflow checking:

- Expressions using the following predefined operators on integral types:

  `++`, `--`, unary `-`, `+`, `-`, `*`, `/`

- Explicit numeric conversions between integral types, or from `float` or `double` to an integral type.

If neither `checked` nor `unchecked` is specified, the default context for non-constant expressions (expressions that are evaluated at run time) is defined by the value of the -checked compiler option. By default the value of that option is unset and arithmetic operations are executed in an unchecked context.

For constant expressions (expressions that can be fully evaluated at compile time), the default context is always checked. Unless a constant expression is explicitly placed in an unchecked context, overflows that occur during the compile-time evaluation of the expression cause compile-time errors.

See Also

- C# Reference
- C# Programming Guide
- C# Keywords
- Statement Keywords

checked (C# Reference)

8/25/2018 • 2 minutes to read • Edit Online

The `checked` keyword is used to explicitly enable overflow checking for integral-type arithmetic operations and conversions.

By default, an expression that contains only constant values causes a compiler error if the expression produces a value that is outside the range of the destination type. If the expression contains one or more non-constant values, the compiler does not detect the overflow. Evaluating the expression assigned to `i2` in the following example does not cause a compiler error.

```
// The following example causes compiler error CS0220 because 2147483647
// is the maximum value for integers.
//int i1 = 2147483647 + 10;

// The following example, which includes variable ten, does not cause
// a compiler error.
int ten = 10;
int i2 = 2147483647 + ten;

// By default, the overflow in the previous statement also does
// not cause a run-time exception. The following line displays
// -2,147,483,639 as the sum of 2,147,483,647 and 10.
Console.WriteLine(i2);
```

By default, these non-constant expressions are not checked for overflow at run time either, and they do not raise overflow exceptions. The previous example displays -2,147,483,639 as the sum of two positive integers.

Overflow checking can be enabled by compiler options, environment configuration, or use of the `checked` keyword. The following examples demonstrate how to use a `checked` expression or a `checked` block to detect the overflow that is produced by the previous sum at run time. Both examples raise an overflow exception.

```
// If the previous sum is attempted in a checked environment, an
// OverflowException error is raised.

// Checked expression.
Console.WriteLine(checked(2147483647 + ten));

// Checked block.
checked
{
    int i3 = 2147483647 + ten;
    Console.WriteLine(i3);
}
```

The unchecked keyword can be used to prevent overflow checking.

Example

This sample shows how to use `checked` to enable overflow checking at run time.

```
class OverFlowTest
{
    // Set maxIntValue to the maximum value for integers.
    static int maxIntValue = 2147483647;

    // Using a checked expression.
    static int CheckedMethod()
    {
        int z = 0;
        try
        {
            // The following line raises an exception because it is checked.
            z = checked(maxIntValue + 10);
        }
        catch (System.OverflowException e)
        {
            // The following line displays information about the error.
            Console.WriteLine("CHECKED and CAUGHT:  " + e.ToString());
        }
        // The value of z is still 0.
        return z;
    }

    // Using an unchecked expression.
    static int UncheckedMethod()
    {
        int z = 0;
        try
        {
            // The following calculation is unchecked and will not
            // raise an exception.
            z = maxIntValue + 10;
        }
        catch (System.OverflowException e)
        {
            // The following line will not be executed.
            Console.WriteLine("UNCHECKED and CAUGHT:  " + e.ToString());
        }
        // Because of the undetected overflow, the sum of 2147483647 + 10 is
        // returned as -2147483639.
        return z;
    }

    static void Main()
    {
        Console.WriteLine("\nCHECKED output value is: {0}",
                          CheckedMethod());
        Console.WriteLine("UNCHECKED output value is: {0}",
                          UncheckedMethod());
    }
    /*
   Output:
   CHECKED and CAUGHT:  System.OverflowException: Arithmetic operation resulted
   in an overflow.
      at ConsoleApplication1.OverFlowTest.CheckedMethod()

   CHECKED output value is: 0
   UNCHECKED output value is: -2147483639
 */
}
```

C# language specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- C# Reference
- C# Programming Guide
- C# Keywords
- Checked and Unchecked
- unchecked

unchecked (C# Reference)

8/25/2018 • 2 minutes to read • Edit Online

The `unchecked` keyword is used to suppress overflow-checking for integral-type arithmetic operations and conversions.

In an unchecked context, if an expression produces a value that is outside the range of the destination type, the overflow is not flagged. For example, because the calculation in the following example is performed in an `unchecked` block or expression, the fact that the result is too large for an integer is ignored, and `int1` is assigned the value -2,147,483,639.

```
unchecked
{
    int1 = 2147483647 + 10;
}
int1 = unchecked(ConstantMax + 10);
```

If the `unchecked` environment is removed, a compilation error occurs. The overflow can be detected at compile time because all the terms of the expression are constants.

Expressions that contain non-constant terms are unchecked by default at compile time and run time. See checked for information about enabling a checked environment.

Because checking for overflow takes time, the use of unchecked code in situations where there is no danger of overflow might improve performance. However, if overflow is a possibility, a checked environment should be used.

Example

This sample shows how to use the `unchecked` keyword.

```
class UncheckedDemo
{
    static void Main(string[] args)
    {
        // int.MaxValue is 2,147,483,647.
        const int ConstantMax = int.MaxValue;
        int int1;
        int int2;
        int variableMax = 2147483647;

        // The following statements are checked by default at compile time. They do not
        // compile.
        //int1 = 2147483647 + 10;
        //int1 = ConstantMax + 10;

        // To enable the assignments to int1 to compile and run, place them inside
        // an unchecked block or expression. The following statements compile and
        // run.
        unchecked
        {
            int1 = 2147483647 + 10;
        }
        int1 = unchecked(ConstantMax + 10);

        // The sum of 2,147,483,647 and 10 is displayed as -2,147,483,639.
        Console.WriteLine(int1);


        // The following statement is unchecked by default at compile time and run
        // time because the expression contains the variable variableMax. It causes
        // overflow but the overflow is not detected. The statement compiles and runs.
        int2 = variableMax + 10;

        // Again, the sum of 2,147,483,647 and 10 is displayed as -2,147,483,639.
        Console.WriteLine(int2);

        // To catch the overflow in the assignment to int2 at run time, put the
        // declaration in a checked block or expression. The following
        // statements compile but raise an overflow exception at run time.
        checked
        {
            //int2 = variableMax + 10;
        }
        //int2 = checked(variableMax + 10);

        // Unchecked sections frequently are used to break out of a checked
        // environment in order to improve performance in a portion of code
        // that is not expected to raise overflow exceptions.
        checked
        {
            // Code that might cause overflow should be executed in a checked
            // environment.
            unchecked
            {
                // This section is appropriate for code that you are confident
                // will not result in overflow, and for which performance is
                // a priority.
            }
            // Additional checked code here.
        }
    }
}
```

C# Language Specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See Also

- C# Reference
- C# Programming Guide
- C# Keywords
- Checked and Unchecked
- checked

fixed Statement (C# Reference)

8/25/2018 • 4 minutes to read • Edit Online

The `fixed` statement prevents the garbage collector from relocating a movable variable. The `fixed` statement is only permitted in an unsafe context. `fixed` can also be used to create fixed size buffers.

The `fixed` statement sets a pointer to a managed variable and "pins" that variable during the execution of the statement. Pointers to movable managed variables are useful only in a `fixed` context. Without a `fixed` context, garbage collection could relocate the variables unpredictably. The C# compiler only lets you assign a pointer to a managed variable in a `fixed` statement.

```
class Point
{
    public int x;
    public int y;
}

unsafe private static void ModifyFixedStorage()
{
    // Variable pt is a managed variable, subject to garbage collection.
    Point pt = new Point();

    // Using fixed allows the address of pt members to be taken,
    // and "pins" pt so that it is not relocated.

    fixed (int* p = &pt.x)
    {
        *p = 1;
    }
}
```

You can initialize a pointer by using an array, a string, a fixed-size buffer, or the address of a variable. The following example illustrates the use of variable addresses, arrays, and strings. For more information about fixed-size buffers, see Fixed Size Buffers.

```
Point point = new Point();
double[] arr = { 0, 1.5, 2.3, 3.4, 4.0, 5.9 };
string str = "Hello World";

// The following two assignments are equivalent. Each assigns the address
// of the first element in array arr to pointer p.

// You can initialize a pointer by using an array.
fixed (double* p = arr) { /*...*/ }

// You can initialize a pointer by using the address of a variable.
fixed (double* p = &arr[0]) { /*...*/ }

// The following assignment initializes p by using a string.
fixed (char* p = str) { /*...*/ }

// The following assignment is not valid, because str[0] is a char,
// which is a value, not a variable.
//fixed (char* p = &str[0]) { /*...*/ }
```

Starting with C# 7.3, the `fixed` statement operates on additional types beyond arrays, strings, fixed-size buffers, or unmanaged variables. Any type that implements a method named `GetPinnableReference` can be pinned. The `GetPinnableReference` must return a `ref` variable to an unmanaged type. See the topic on pointer types for more information. The .NET types System.Span<T> and System.ReadOnlySpan<T> introduced in .NET Core 2.0 make use of this pattern and can be pinned. This is shown in the following example:

```
unsafe private static void FixedSpanExample()
{
    int[] PascalsTriangle = {
                  1,
                1,  1,
              1,  2,  1,
            1,  3,  3,  1,
          1,  4,  6,  4,  1,
        1,  5,  10, 10, 5,  1
    };

    Span<int> RowFive = new Span<int>(PascalsTriangle, 10, 5);

    fixed (int* ptrToRow = RowFive)
    {
        // Sum the numbers 1,4,6,4,1
        var sum = 0;
        for (int i = 0; i < RowFive.Length; i++)
        {
            sum += *(ptrToRow + i);
        }
        Console.WriteLine(sum);
    }
}
```

If you are creating types that should participate in this pattern, see Span<T>.GetPinnableReference() for an example of implementing the pattern.

Multiple pointers can be initialized in one statement if they are all the same type:

```
fixed (byte* ps = srcarray, pd = dstarray) {...}
```

To initialize pointers of different types, simply nest `fixed` statements, as shown in the following example.

```
fixed (int* p1 = &point.x)
{
    fixed (double* p2 = &arr[5])
    {
        // Do something with p1 and p2.
    }
}
```

After the code in the statement is executed, any pinned variables are unpinned and subject to garbage collection. Therefore, do not point to those variables outside the `fixed` statement. The variables declared in the `fixed` statement are scoped to that statement, making this easier:

```
fixed (byte* ps = srcarray, pd = dstarray)
{
   ...
}
// ps and pd are no longer in scope here.
```

Pointers initialized in `fixed` statements are readonly variables. If you want to modify the pointer value, you must declare a second pointer variable, and modify that. The variable declared in the `fixed` statement cannot be modified:

```
fixed (byte* ps = srcarray, pd = dstarray)
{
    byte* pSourceCopy = ps;
    pSourceCopy++; // point to the next element.
    ps++; // invalid: cannot modify ps, as it is declared in the fixed statement.
}
```

In unsafe mode, you can allocate memory on the stack, where it is not subject to garbage collection and therefore does not need to be pinned. For more information, see stackalloc.

```
// Unsafe method: takes a pointer to an int.
unsafe static void SquarePtrParam(int* p)
{
    *p *= *p;
}

unsafe static void SquarePointValue()
{
    Point pt = new Point
    {
        x = 5,
        y = 6
    };
    // Pin pt in place:
    fixed (int* p = &pt.x)
    {
        SquarePtrParam(p);
    }
    // pt now unpinned.
    Console.WriteLine("{0} {1}", pt.x, pt.y);
    /*
    Output:
    25 6
     */
}
```

C# Language Specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See Also

- C# Reference
- C# Programming Guide
- C# Keywords
- unsafe
- Fixed Size Buffers

lock statement (C# Reference)

8/30/2018 • 2 minutes to read • Edit Online

The `lock` statement obtains the mutual-exclusion lock for a given object, executes a statement block, and then releases the lock. While a lock is held, the thread that holds the lock can again obtain and release the lock. Any other thread is blocked from obtaining the lock and waits until the lock is released.

The `lock` statement is of the form

```
lock (x)
{
    // Your code...
}
```

where `x` is an expression of a reference type. It's precisely equivalent to

```
object __lockObj = x;
bool __lockWasTaken = false;
try
{
    System.Threading.Monitor.Enter(__lockObj, ref __lockWasTaken);
    // Your code...
}
finally
{
    if (__lockWasTaken) System.Threading.Monitor.Exit(__lockObj);
}
```

Since the code uses a try...finally block, the lock is released even if an exception is thrown within the body of a `lock` statement.

You can't use the await keyword in the body of a `lock` statement.

Remarks

When you synchronize thread access to shared resource, lock on a dedicated object instance (for example, `private readonly object balanceLock = new object();`) or another instance that is unlikely to be used as a lock object by unrelated parts of the code. Avoid using the same lock object instance for different shared resources, as it might result in deadlock or lock contention. In particular, avoid using

- `this` (might be used by the callers as a lock),
- Type instances (might be obtained by the typeof operator or reflection),
- string instances, including string literals,

as lock objects.

Example

The following example defines an `Account` class that synchronizes access to its private `balance` field by locking on a dedicated `balanceLock` instance. Using the same instance for locking ensures that the `balance` field cannot be updated simultaneously by two threads attempting to call the `Debit` or `Credit` methods simultaneously.

```
using System;
using System.Threading.Tasks;

public class Account
{
    private readonly object balanceLock = new object();
    private decimal balance;

    public Account(decimal initialBalance)
    {
        balance = initialBalance;
    }

    public decimal Debit(decimal amount)
    {
        lock (balanceLock)
        {
            if (balance >= amount)
            {
                Console.WriteLine($"Balance before debit :{balance, 5}");
                Console.WriteLine($"Amount to remove     :{amount, 5}");
                balance = balance - amount;
                Console.WriteLine($"Balance after debit  :{balance, 5}");
                return amount;
            }
            else
            {
                return 0;
            }
        }
    }

    public void Credit(decimal amount)
    {
        lock (balanceLock)
        {
            Console.WriteLine($"Balance before credit:{balance, 5}");
            Console.WriteLine($"Amount to add        :{amount, 5}");
            balance = balance + amount;
            Console.WriteLine($"Balance after credit :{balance, 5}");
        }
    }
}

class AccountTest
{
    static void Main()
    {
        var account = new Account(1000);
        var tasks = new Task[100];
        for (int i = 0; i < tasks.Length; i++)
        {
            tasks[i] = Task.Run(() => RandomlyUpdate(account));
        }
        Task.WaitAll(tasks);
    }

    static void RandomlyUpdate(Account account)
    {
        var rnd = new Random();
        for (int i = 0; i < 10; i++)
        {
            var amount = rnd.Next(1, 100);
            bool doCredit = rnd.NextDouble() < 0.5;
            if (doCredit)
            {
                account.Credit(amount);
            }
            else
```

```
                {
                    account.Debit(amount);
                }
            }
        }
    }
```

C# language specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- System.Threading.Monitor
- System.Threading.SpinLock
- System.Threading.Interlocked
- C# Reference
- C# Keywords
- Statement Keywords
- Interlocked operations
- Overview of synchronization primitives

Method Parameters (C# Reference)

8/25/2018 • 2 minutes to read • Edit Online

Parameters declared for a method without in, ref or out, are passed to the called method by value. That value can be changed in the method, but the changed value will not be retained when control passes back to the calling procedure. By using a method parameter keyword, you can change this behavior.

This section describes the keywords you can use when declaring method parameters:

- params specifies that this parameter may take a variable number of arguments.
- in specifies that this parameter is passed by reference but is only read by the called method.
- ref specifies that this parameter is passed by reference and may be read or written by the called method.
- out specifies that this parameter is passed by reference and is written by the called method.

See Also

- C# Reference
- C# Programming Guide
- C# Keywords

params (C# Reference)

8/29/2018 • 2 minutes to read • Edit Online

By using the `params` keyword, you can specify a method parameter that takes a variable number of arguments.

You can send a comma-separated list of arguments of the type specified in the parameter declaration or an array of arguments of the specified type. You also can send no arguments. If you send no arguments, the length of the `params` list is zero.

No additional parameters are permitted after the `params` keyword in a method declaration, and only one `params` keyword is permitted in a method declaration.

The declared type of the `params` parameter must be a single-dimensional array, as the following example shows. Otherwise, a compiler error CS0225 occurs.

Example

The following example demonstrates various ways in which arguments can be sent to a `params` parameter.

```
public class MyClass
{
    public static void UseParams(params int[] list)
    {
        for (int i = 0; i < list.Length; i++)
        {
            Console.Write(list[i] + " ");
        }
        Console.WriteLine();
    }

    public static void UseParams2(params object[] list)
    {
        for (int i = 0; i < list.Length; i++)
        {
            Console.Write(list[i] + " ");
        }
        Console.WriteLine();
    }

    static void Main()
    {
        // You can send a comma-separated list of arguments of the
        // specified type.
        UseParams(1, 2, 3, 4);
        UseParams2(1, 'a', "test");

        // A params parameter accepts zero or more arguments.
        // The following calling statement displays only a blank line.
        UseParams2();

        // An array argument can be passed, as long as the array
        // type matches the parameter type of the method being called.
        int[] myIntArray = { 5, 6, 7, 8, 9 };
        UseParams(myIntArray);

        object[] myObjArray = { 2, 'b', "test", "again" };
        UseParams2(myObjArray);

        // The following call causes a compiler error because the object
        // array cannot be converted into an integer array.
        //UseParams(myObjArray);

        // The following call does not cause an error, but the entire
        // integer array becomes the first element of the params array.
        UseParams2(myIntArray);
    }
}
/*
Output:
    1 2 3 4
    1 a test

    5 6 7 8 9
    2 b test again
    System.Int32[]
*/
```

C# language specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- C# Reference
- C# Programming Guide
- C# Keywords
- Method Parameters

in parameter modifier (C# Reference)

8/25/2018 • 4 minutes to read • Edit Online

The `in` keyword causes arguments to be passed by reference. It is like the ref or out keywords, except that `in` arguments cannot be modified by the called method. Whereas `ref` arguments may be modified, `out` arguments must be modified by the caller, and those modifications are observable in the calling context.

```
int readonlyArgument = 44;
InArgExample(readonlyArgument);
Console.WriteLine(readonlyArgument);     // value is still 44

void InArgExample(in int number)
{
    // Uncomment the following line to see error CS8331
    //number = 19;
}
```

The preceding example demonstrates that the `in` modifier is usually unnecessary at the call site. It is only required in the method declaration.

NOTE

The `in` keyword can also be used with a generic type parameter to specify that the type parameter is contravariant, as part of a `foreach` statement, or as part of a `join` clause in a LINQ query. For more information on the use of the `in` keyword in these contexts, see in, which provides links to all those uses.

Variables passed as `in` arguments must be initialized before being passed in a method call. However, the called method may not assign a value or modify the argument.

Although the `in`, `ref`, and `out` keywords cause different run-time behavior, they are not considered part of the method signature at compile time. Therefore, methods cannot be overloaded if the only difference is that one method takes a `ref` or `in` argument and the other takes an `out` argument. The following code, for example, will not compile:

```
class CS0663_Example
{
    // Compiler error CS0663: "Cannot define overloaded
    // methods that differ only on in, ref and out".
    public void SampleMethod(in int i) { }
    public void SampleMethod(ref int i) { }
}
```

Overloading based on the presence of `in` is allowed:

```
class InOverloads
{
    public void SampleMethod(in int i) { }
    public void SampleMethod(int i) { }
}
```

Overload resolution rules

You can understand the overload resolution rules for methods with by value vs. `in` arguments by understanding the motivation for `in` arguments. Defining methods using `in` parameters is a potential performance optimization. Some `struct` type arguments may be large in size, and when methods are called in tight loops or critical code paths, the cost of copying those structures is critical. Methods declare `in` parameters to specify that arguments may be passed by reference safely because the called method does not modify the state of that argument. Passing those arguments by reference avoids the (potentially) expensive copy.

Specifying `in` for arguments at the call site is typically optional. There is no semantic difference between passing arguments by value and passing them by reference using the `in` modifier. The `in` modifier at the call site is optional because you don't need to indicate that the argument's value might be changed. You explicitly add the `in` modifier at the call site to ensure the argument is passed by reference, not by value. Explicitly using `in` has the following two effects:

First, specifying `in` at the call site forces the compiler to select a method defined with a matching `in` parameter. Otherwise, when two methods differ only in the presence of `in`, the by value overload is a better match.

Second, specifying `in` declares your intent to pass an argument by reference. The argument used with `in` must represent a location that can be directly referred to. The same general rules for `out` and `ref` arguments apply: You cannot use constants, ordinary properties, or other expressions that produce values. Otherwise, omitting `in` at the call site informs the compiler that you will allow it to create a temporary variable to pass by read-only reference to the method. The compiler creates a temporary variable to overcome several restrictions with `in` arguments:

- A temporary variable allows compile-time constants as `in` parameters.
- A temporary variable allows properties, or other expressions for `in` parameters.
- A temporary variable allows arguments where there is an implicit conversion from the argument type to the parameter type.

In all the preceding instances, the compiler creates a temporary variable that stores the value of the constant, property, or other expression.

The following code illustrates these rules:

```
static void Method(in int argument)
{
    // implementation removed
}

Method(5); // OK, temporary variable created.
Method(5L); // CS1503: no implicit conversion from long to int
short s = 0;
Method(s); // OK, temporary int created with the value 0
Method(in s); // CS1503: cannot convert from in short to in int
int i = 42;
Method(i); // passed by readonly reference
Method(in i); // passed by readonly reference, explicitly using `in`
```

Now, suppose another method using by value arguments was available. The results change as shown in the following code:

```
static void Method(int argument)
{
    // implementation removed
}

static void Method(in int argument)
{
    // implementation removed
}

Method(5); // Calls overload passed by value
Method(5L); // CS1503: no implicit conversion from long to int
short s = 0;
Method(s); // Calls overload passed by value.
Method(in s); // CS1503: cannot convert from in short to in int
int i = 42;
Method(i); // Calls overload passed by value
Method(in i); // passed by readonly reference, explicitly using `in`
```

The only method call where the argument is passed by reference is the final one.

> **NOTE**
>
> The preceding code uses `int` as the argument type for simplicity. Because `int` is no larger than a reference in most modern machines, there is no benefit to passing a single `int` as a readonly reference.

Limitations on `in` parameters

You can't use the `in`, `ref`, and `out` keywords for the following kinds of methods:

- Async methods, which you define by using the async modifier.
- Iterator methods, which include a yield return or `yield break` statement.

C# Language Specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See Also

- C# Reference
- C# Programming Guide
- C# Keywords
- Method Parameters
- Reference Semantics with Value Types

ref (C# Reference)

9/8/2018 • 7 minutes to read • Edit Online

The `ref` keyword indicates a value that is passed by reference. It is used in four different contexts:

- In a method signature and in a method call, to pass an argument to a method by reference. See Passing an argument by reference for more information.
- In a method signature, to return a value to the caller by reference. See Reference return values for more information.
- In a member body, to indicate that a reference return value is stored locally as a reference that the caller intends to modify or, in general, a local variable accesses another value by reference. See Ref locals for more information.
- In a `struct` declaration to declare a `ref struct` or a `ref readonly struct`. For more information, see Reference semantics with value types.

Passing an argument by reference

When used in a method's parameter list, the `ref` keyword indicates that an argument is passed by reference, not by value. The effect of passing by reference is that any change to the argument in the called method is reflected in the calling method. For example, if the caller passes a local variable expression or an array element access expression, and the called method replaces the object to which the ref parameter refers, then the caller's local variable or the array element now refers to the new object when the method returns.

> **NOTE**
>
> Do not confuse the concept of passing by reference with the concept of reference types. The two concepts are not the same. A method parameter can be modified by `ref` regardless of whether it is a value type or a reference type. There is no boxing of a value type when it is passed by reference.

To use a `ref` parameter, both the method definition and the calling method must explicitly use the `ref` keyword, as shown in the following example.

```
void Method(ref int refArgument)
{
    refArgument = refArgument + 44;
}

int number = 1;
Method(ref number);
Console.WriteLine(number);
// Output: 45
```

An argument that is passed to a `ref` or `in` parameter must be initialized before it is passed. This differs from out parameters, whose arguments do not have to be explicitly initialized before they are passed.

Members of a class can't have signatures that differ only by `ref`, `in`, or `out`. A compiler error occurs if the only difference between two members of a type is that one of them has a `ref` parameter and the other has an `out`, or `in` parameter. The following code, for example, doesn't compile.

```
class CS0663_Example
{
    // Compiler error CS0663: "Cannot define overloaded
    // methods that differ only on ref and out".
    public void SampleMethod(out int i) { }
    public void SampleMethod(ref int i) { }
}
```

However, methods can be overloaded when one method has a `ref`, `in`, or `out` parameter and the other has a value parameter, as shown in the following example.

```
class RefOverloadExample
{
    public void SampleMethod(int i) { }
    public void SampleMethod(ref int i) { }
}
```

In other situations that require signature matching, such as hiding or overriding, `in`, `ref`, and `out` are part of the signature and don't match each other.

Properties are not variables. They are methods, and cannot be passed to `ref` parameters.

You can't use the `ref`, `in`, and `out` keywords for the following kinds of methods:

- Async methods, which you define by using the async modifier.
- Iterator methods, which include a yield return or `yield break` statement.

Passing an argument by reference: An example

The previous examples pass value types by reference. You can also use the `ref` keyword to pass reference types by reference. Passing a reference type by reference enables the called method to replace the object to which the reference parameter refers in the caller. The storage location of the object is passed to the method as the value of the reference parameter. If you change the value in the storage location of the parameter (to point to a new object), you also change the storage location to which the caller refers. The following example passes an instance of a reference type as a `ref` parameter.

```
class Product
{
    public Product(string name, int newID)
    {
        ItemName = name;
        ItemID = newID;
    }

    public string ItemName { get; set; }
    public int ItemID { get; set; }
}

private static void ModifyProductsByReference()
{
    // Declare an instance of Product and display its initial values.
    Product item = new Product("Fasteners", 54321);
    System.Console.WriteLine("Original values in Main.  Name: {0}, ID: {1}\n",
        item.ItemName, item.ItemID);

    // Pass the product instance to ChangeByReference.
    ChangeByReference(ref item);
    System.Console.WriteLine("Back in Main.  Name: {0}, ID: {1}\n",
        item.ItemName, item.ItemID);
}

private static void ChangeByReference(ref Product itemRef)
{
    // Change the address that is stored in the itemRef parameter.
    itemRef = new Product("Stapler", 99999);

    // You can change the value of one of the properties of
    // itemRef. The change happens to item in Main as well.
    itemRef.ItemID = 12345;
}
```

For more information about how to pass reference types by value and by reference, see Passing Reference-Type Parameters.

Reference return values

Reference return values (or ref returns) are values that a method returns by reference to the caller. That is, the caller can modify the value returned by a method, and that change is reflected in the state of the object that contains the method.

A reference return value is defined by using the `ref` keyword:

- In the method signature. For example, the following method signature indicates that the `GetCurrentPrice` method returns a Decimal value by reference.

```
public ref decimal GetCurrentPrice()
```

- Between the `return` token and the variable returned in a `return` statement in the method. For example:

```
return ref DecimalArray[0];
```

In order for the caller to modify the object's state, the reference return value must be stored to a variable that is explicitly defined as a ref local.

For an example, see A ref returns and ref locals example

Ref locals

A ref local variable is used to refer to values returned using `return ref`. A ref local variable cannot be initialized to a non-ref return value. In other words, the right hand side of the initialization must be a reference. Any modifications to the value of the ref local are reflected in the state of the object whose method returned the value by reference.

You define a ref local by using the `ref` keyword before the variable declaration, as well as immediately before the call to the method that returns the value by reference.

For example, the following statement defines a ref local value that is returned by a method named `GetEstimatedValue`:

```
ref decimal estValue = ref Building.GetEstimatedValue();
```

You can access a value by reference in the same way. In some cases, accessing a value by reference increases performance by avoiding a potentially expensive copy operation. For example, the following statement shows how one can define a ref local value that is used to reference a value.

```
ref VeryLargeStruct reflocal = ref veryLargeStruct;
```

Note that in both examples the `ref` keyword must be used in both places, or the compiler generates error CS8172, "Cannot initialize a by-reference variable with a value."

A ref returns and ref locals example

The following example defines a `Book` class that has two String fields, `Title` and `Author`. It also defines a `BookCollection` class that includes a private array of `Book` objects. Individual book objects are returned by reference by calling its `GetBookByTitle` method.

```
public class Book
{
    public string Author;
    public string Title;
}

public class BookCollection
{
    private Book[] books = { new Book { Title = "Call of the Wild, The", Author = "Jack London" },
                        new Book { Title = "Tale of Two Cities, A", Author = "Charles Dickens" }
                       };
    private Book nobook = null;

    public ref Book GetBookByTitle(string title)
    {
        for (int ctr = 0; ctr < books.Length; ctr++)
        {
            if (title == books[ctr].Title)
                return ref books[ctr];
        }
        return ref nobook;
    }

    public void ListBooks()
    {
        foreach (var book in books)
        {
            Console.WriteLine($"{book.Title}, by {book.Author}");
        }
        Console.WriteLine();
    }
}
```

When the caller stores the value returned by the `GetBookByTitle` method as a ref local, changes that the caller makes to the return value are reflected in the `BookCollection` object, as the following example shows.

```
var bc = new BookCollection();
bc.ListBooks();

ref var book = ref bc.GetBookByTitle("Call of the Wild, The");
if (book != null)
    book = new Book { Title = "Republic, The", Author = "Plato" };
bc.ListBooks();
// The example displays the following output:
//       Call of the Wild, The, by Jack London
//       Tale of Two Cities, A, by Charles Dickens
//
//       Republic, The, by Plato
//       Tale of Two Cities, A, by Charles Dickens
```

C# language specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- Reference semantics with value types
- Passing Parameters
- Method Parameters

- C# Reference
- C# Programming Guide
- C# Keywords

out parameter modifier (C# Reference)

9/8/2018 • 3 minutes to read • Edit Online

The `out` keyword causes arguments to be passed by reference. It is like the ref keyword, except that `ref` requires that the variable be initialized before it is passed. It is also like the in keyword, except that `in` does not allow the called method to modify the argument value. To use an `out` parameter, both the method definition and the calling method must explicitly use the `out` keyword. For example:

```
int initializeInMethod;
OutArgExample(out initializeInMethod);
Console.WriteLine(initializeInMethod);     // value is now 44

void OutArgExample(out int number)
{
    number = 44;
}
```

NOTE

The `out` keyword can also be used with a generic type parameter to specify that the type parameter is covariant. For more information on the use of the `out` keyword in this context, see out (Generic Modifier).

Variables passed as `out` arguments do not have to be initialized before being passed in a method call. However, the called method is required to assign a value before the method returns.

Although the `in`, `ref`, and `out` keywords cause different run-time behavior, they are not considered part of the method signature at compile time. Therefore, methods cannot be overloaded if the only difference is that one method takes a `ref` or `in` argument and the other takes an `out` argument. The following code, for example, will not compile:

```
class CS0663_Example
{
    // Compiler error CS0663: "Cannot define overloaded
    // methods that differ only on ref and out".
    public void SampleMethod(out int i) { }
    public void SampleMethod(ref int i) { }
}
```

Overloading is legal, however, if one method takes a `ref`, `in`, or `out` argument and the other has none of those modifiers, like this:

```
class OutOverloadExample
{
    public void SampleMethod(int i) { }
    public void SampleMethod(out int i) => i = 5;
}
```

The compiler chooses the best overload by matching the parameter modifiers at the call site to the parameter modifiers used in the method call.

Properties are not variables and therefore cannot be passed as `out` parameters.

You can't use the `in`, `ref`, and `out` keywords for the following kinds of methods:

- Async methods, which you define by using the async modifier.
- Iterator methods, which include a yield return or `yield break` statement.

Declaring `out` arguments

Declaring a method with `out` arguments is useful when you want a method to return multiple values. The following example uses `out` to return three variables with a single method call. Note that the third argument is assigned to null. This enables methods to return values optionally.

```
void Method(out int answer, out string message, out string stillNull)
{
    answer = 44;
    message = "I've been returned";
    stillNull = null;
}

int argNumber;
string argMessage, argDefault;
Method(out argNumber, out argMessage, out argDefault);
Console.WriteLine(argNumber);
Console.WriteLine(argMessage);
Console.WriteLine(argDefault == null);
```

The Try pattern involves returning a `bool` to indicate whether an operation succeeded or failed, and returning the value produced by the operation in an `out` argument. A number of parsing methods, such as the DateTime.TryParse method, use this pattern.

Calling a method with an `out` argument

In C# 6 and earlier, you must declare a variable in a separate statement before you pass it as an `out` argument. The following example declares a variable named `number` before it is passed to the Int32.TryParse method, which attempts to convert a string to a number.

```
string numberAsString = "1640";

int number;
if (Int32.TryParse(numberAsString, out number))
    Console.WriteLine($"Converted '{numberAsString}' to {number}");
else
    Console.WriteLine($"Unable to convert '{numberAsString}'");
// The example displays the following output:
//       Converted '1640' to 1640
```

Starting with C# 7.0, you can declare the `out` variable in the argument list of the method call, rather than in a separate variable declaration. This produces more compact, readable code, and also prevents you from inadvertently assigning a value to the variable before the method call. The following example is like the previous example, except that it defines the `number` variable in the call to the Int32.TryParse method.

```
string numberAsString = "1640";

if (Int32.TryParse(numberAsString, out int number))
    Console.WriteLine($"Converted '{numberAsString}' to {number}");
else
    Console.WriteLine($"Unable to convert '{numberAsString}'");
// The example displays the following output:
//       Converted '1640' to 1640
```

In the previous example, the `number` variable is strongly typed as an `int`. You can also declare an implicitly typed local variable, as the following example does.

```
string numberAsString = "1640";

if (Int32.TryParse(numberAsString, out var number))
    Console.WriteLine($"Converted '{numberAsString}' to {number}");
else
    Console.WriteLine($"Unable to convert '{numberAsString}'");
// The example displays the following output:
//       Converted '1640' to 1640
```

C# Language Specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See Also

- C# Reference
- C# Programming Guide
- C# Keywords
- Method Parameters

Namespace Keywords (C# Reference)

8/25/2018 • 2 minutes to read • Edit Online

This section describes the keywords and operators that are associated with using namespaces:

- namespace
- using
- using static
- . Operator
- :: Operator
- extern alias

See Also

- C# Reference
- C# Programming Guide
- C# Keywords
- Namespaces

namespace (C# Reference)

8/19/2018 • 2 minutes to read • Edit Online

The `namespace` keyword is used to declare a scope that contains a set of related objects. You can use a namespace to organize code elements and to create globally unique types.

```
namespace SampleNamespace
{
    class SampleClass { }

    interface ISampleInterface { }

    struct SampleStruct { }

    enum SampleEnum { a, b }

    delegate void SampleDelegate(int i);

    namespace SampleNamespace.Nested
    {
        class SampleClass2 { }
    }
}
```

Remarks

Within a namespace, you can declare zero or more of the following types:

- another namespace
- class
- interface
- struct
- enum
- delegate

Whether or not you explicitly declare a namespace in a C# source file, the compiler adds a default namespace. This unnamed namespace, sometimes referred to as the global namespace, is present in every file. Any identifier in the global namespace is available for use in a named namespace.

Namespaces implicitly have public access and this is not modifiable. For a discussion of the access modifiers you can assign to elements in a namespace, see Access Modifiers.

It is possible to define a namespace in two or more declarations. For example, the following example defines two classes as part of the `MyCompany` namespace:

```
namespace MyCompany.Proj1
{
    class MyClass
    {
    }
}

namespace MyCompany.Proj1
{
    class MyClass1
    {
    }
}
```

Example

The following example shows how to call a static method in a nested namespace.

```
namespace SomeNameSpace
{
    public class MyClass
    {
        static void Main()
        {
            Nested.NestedNameSpaceClass.SayHello();
        }
    }

    // a nested namespace
    namespace Nested
    {
        public class NestedNameSpaceClass
        {
            public static void SayHello()
            {
                Console.WriteLine("Hello");
            }
        }
    }
}
// Output: Hello
```

Related resources

For more information about using namespaces, see the following topics:

- Namespaces
- Using Namespaces
- How to: Use the Global Namespace Alias

C# language specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- C# Reference

- C# Programming Guide
- C# Keywords
- Namespace Keywords
- using

using (C# Reference)

8/25/2018 • 2 minutes to read • Edit Online

The `using` keyword has two major uses:

- As a directive, when it is used to create an alias for a namespace or to import types defined in other namespaces. See using Directive.
- As a statement, when it defines a scope at the end of which an object will be disposed. See using Statement.

In addition, the using static directive lets you define a type whose static members you can access without specifying a type name.

See Also

- C# Reference
- C# Programming Guide
- C# Keywords
- Namespace Keywords
- Namespaces
- extern

using Directive (C# Reference)

8/25/2018 • 3 minutes to read • Edit Online

The `using` directive has three uses:

- To allow the use of types in a namespace so that you do not have to qualify the use of a type in that namespace:

```
using System.Text;
```

- To allow you to access static members and nested types of a type without having to qualify the access with the type name.

```
using static System.Math;
```

  For more information, see the using static directive.

- To create an alias for a namespace or a type. This is called a *using alias directive*.

```
using Project = PC.MyCompany.Project;
```

The `using` keyword is also used to create *using statements*, which help ensure that IDisposable objects such as files and fonts are handled correctly. See using Statement for more information.

Using Static Type

You can access static members of a type without having to qualify the access with the type name:

```
using static System.Console;
using static System.Math;
class Program
{
    static void Main()
    {
        WriteLine(Sqrt(3*3 + 4*4));
    }
}
```

Remarks

The scope of a `using` directive is limited to the file in which it appears.

The `using` directive can appear:

- At the beginning of a source code file, before any namespace or type definitions.
- In any namespace, but before any namespace or types declared in this namespace.

Otherwise, compiler error CS1529 is generated.

Create a `using` alias directive to make it easier to qualify an identifier to a namespace or type. In any `using` directive, the fully-qualified namespace or type must be used regardless of the `using` directives that come before it. No `using` alias can be used in the declaration of a `using` directive. For example, the following generates a compiler error:

```
using s = System.Text;
using s.RegularExpressions;
```

Create a `using` directive to use the types in a namespace without having to specify the namespace. A `using` directive does not give you access to any namespaces that are nested in the namespace you specify.

Namespaces come in two categories: user-defined and system-defined. User-defined namespaces are namespaces defined in your code. For a list of the system-defined namespaces, see .NET API Browser.

For examples on referencing methods in other assemblies, see Create and Use Assemblies Using the Command Line.

Example 1

The following example shows how to define and use a `using` alias for a namespace:

```
namespace PC
{
    // Define an alias for the nested namespace.
    using Project = PC.MyCompany.Project;
    class A
    {
        void M()
        {
            // Use the alias
            Project.MyClass mc = new Project.MyClass();
        }
    }
    namespace MyCompany
    {
        namespace Project
        {
            public class MyClass { }
        }
    }
}
```

A using alias directive cannot have an open generic type on the right hand side. For example, you cannot create a using alias for a List<T>, but you can create one for a List<int>.

Example 2

The following example shows how to define a `using` directive and a `using` alias for a class:

```
using System;

// Using alias directive for a class.
using AliasToMyClass = NameSpace1.MyClass;

// Using alias directive for a generic class.
using UsingAlias = NameSpace2.MyClass<int>;

namespace NameSpace1
{
    public class MyClass
    {
        public override string ToString()
        {
            return "You are in NameSpace1.MyClass.";
        }
    }

}

namespace NameSpace2
{
    class MyClass<T>
    {
        public override string ToString()
        {
            return "You are in NameSpace2.MyClass.";
        }
    }
}

namespace NameSpace3
{
    // Using directive:
    using NameSpace1;
    // Using directive:
    using NameSpace2;

    class MainClass
    {
        static void Main()
        {
            AliasToMyClass instance1 = new AliasToMyClass();
            Console.WriteLine(instance1);

            UsingAlias instance2 = new UsingAlias();
            Console.WriteLine(instance2);

        }
    }
}
// Output:
//    You are in NameSpace1.MyClass.
//    You are in NameSpace2.MyClass.
```

C# Language Specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See Also

- C# Reference
- C# Programming Guide

- Using Namespaces
- C# Keywords
- Namespace Keywords
- Namespaces
- using Statement

using static Directive (C# Reference)

8/25/2018 • 3 minutes to read • Edit Online

The `using static` directive designates a type whose static members and nested types you can access without specifying a type name. Its syntax is:

```
using static <fully-qualified-type-name>;
```

where *fully-qualified-type-name* is the name of the type whose static members and nested types can be referenced without specifying a type name. If you do not provide a fully qualified type name (the full namespace name along with the type name), C# generates compiler error CS0246: "The type or namespace name 'type/namespace' could not be found (are you missing a using directive or an assembly reference?)".

The `using static` directive applies to any type that has static members (or nested types), even if it also has instance members. However, instance members can only be invoked through the type instance.

The `using static` directive was introduced in C# 6.

Remarks

Ordinarily, when you call a static member, you provide the type name along with the member name. Repeatedly entering the same type name to invoke members of the type can result in verbose, obscure code. For example, the following definition of a `Circle` class references a number of members of the Math class.

```
using System;

public class Circle
{
   public Circle(double radius)
   {
      Radius = radius;
   }

   public double Radius { get; set; }

   public double Diameter
   {
      get { return 2 * Radius; }
   }

   public double Circumference
   {
      get { return 2 * Radius * Math.PI; }
   }

   public double Area
   {
      get { return Math.PI * Math.Pow(Radius, 2); }
   }
}
```

By eliminating the need to explicitly reference the Math class each time a member is referenced, the `using static` directive produces much cleaner code:

```
using System;
using static System.Math;

public class Circle
{
   public Circle(double radius)
   {
      Radius = radius;
   }

   public double Radius { get; set; }

   public double Diameter
   {
      get { return 2 * Radius; }
   }

   public double Circumference
   {
      get { return 2 * Radius * PI; }
   }

   public double Area
   {
      get { return PI * Pow(Radius, 2); }
   }
}
```

`using static` imports only accessible static members and nested types declared in the specified type. Inherited members are not imported. You can import from any named type with a using static directive, including Visual Basic modules. If F# top-level functions appear in metadata as static members of a named type whose name is a valid C# identifier, then the F# functions can be imported.

`using static` makes extension methods declared in the specified type available for extension method lookup. However, the names of the extension methods are not imported into scope for unqualified reference in code.

Methods with the same name imported from different types by different `using static` directives in the same compilation unit or namespace form a method group. Overload resolution within these method groups follows normal C# rules.

Example

The following example uses the `using static` directive to make the static members of the Console, Math, and String classes available without having to specify their type name.

```csharp
using System;
using static System.Console;
using static System.Math;
using static System.String;

class Program
{
   static void Main()
   {
      Write("Enter a circle's radius: ");
      var input = ReadLine();
      if (!IsNullOrEmpty(input) && double.TryParse(input, out var radius)) {
         var c = new Circle(radius);

         string s = "\nInformation about the circle:\n";
         s = s + Format("   Radius: {0:N2}\n", c.Radius);
         s = s + Format("   Diameter: {0:N2}\n", c.Diameter);
         s = s + Format("   Circumference: {0:N2}\n", c.Circumference);
         s = s + Format("   Area: {0:N2}\n", c.Area);
         WriteLine(s);
      }
      else {
         WriteLine("Invalid input...");
      }
   }
}

public class Circle
{
   public Circle(double radius)
   {
      Radius = radius;
   }

   public double Radius { get; set; }

   public double Diameter
   {
      get { return 2 * Radius; }
   }

   public double Circumference
   {
      get { return 2 * Radius * PI; }
   }

   public double Area
   {
      get { return PI * Pow(Radius, 2); }
   }
}
// The example displays the following output:
//       Enter a circle's radius: 12.45
//
//       Information about the circle:
//          Radius: 12.45
//          Diameter: 24.90
//          Circumference: 78.23
//          Area: 486.95
```

In the example, the `using static` directive could also have been applied to the Double type. This would have made it possible to call the TryParse(String, Double) method without specifying a type name. However, this creates less readable code, since it becomes necessary to check the `using static` statements to determine which numeric type's `TryParse` method is called.

See also

- using directive
- C# Reference
- C# Keywords
- Using Namespaces
- Namespace Keywords
- Namespaces

using Statement (C# Reference)

8/25/2018 • 2 minutes to read • Edit Online

Provides a convenient syntax that ensures the correct use of IDisposable objects.

Example

The following example shows how to use the `using` statement.

```
using (Font font1 = new Font("Arial", 10.0f))
{
    byte charset = font1.GdiCharSet;
}
```

Remarks

File and Font are examples of managed types that access unmanaged resources (in this case file handles and device contexts). There are many other kinds of unmanaged resources and class library types that encapsulate them. All such types must implement the IDisposable interface.

When the lifetime of an `IDisposable` object is limited to a single method, you should declare and instantiate it in the `using` statement. The `using` statement calls the Dispose method on the object in the correct way, and (when you use it as shown earlier) it also causes the object itself to go out of scope as soon as Dispose is called. Within the `using` block, the object is read-only and cannot be modified or reassigned.

The `using` statement ensures that Dispose is called even if an exception occurs within the `using` block. You can achieve the same result by putting the object inside a `try` block and then calling Dispose in a `finally` block; in fact, this is how the `using` statement is translated by the compiler. The code example earlier expands to the following code at compile time (note the extra curly braces to create the limited scope for the object):

```
{
  Font font1 = new Font("Arial", 10.0f);
  try
  {
    byte charset = font1.GdiCharSet;
  }
  finally
  {
    if (font1 != null)
      ((IDisposable)font1).Dispose();
  }
}
```

For more information about the `try`-`finally` statement, see the try-finally topic.

Multiple instances of a type can be declared in the `using` statement, as shown in the following example:

```
using (Font font3 = new Font("Arial", 10.0f),
            font4 = new Font("Arial", 10.0f))
{
    // Use font3 and font4.
}
```

You can instantiate the resource object and then pass the variable to the `using` statement, but this is not a best practice. In this case, after control leaves the `using` block, the object remains in scope but probably has no access to its unmanaged resources. In other words, it's not fully initialized anymore. If you try to use the object outside the `using` block, you risk causing an exception to be thrown. For this reason, it's generally better to instantiate the object in the `using` statement and limit its scope to the `using` block.

```
Font font2 = new Font("Arial", 10.0f);
using (font2) // not recommended
{
    // use font2
}
// font2 is still in scope
// but the method call throws an exception
float f = font2.GetHeight();
```

For more information about disposing of `IDisposable` objects, see Using objects that implement IDisposable.

C# Language Specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See Also

- C# Reference
- C# Programming Guide
- C# Keywords
- using Directive
- Garbage Collection
- Using objects that implement IDisposable
- IDisposable interface

extern alias (C# Reference)

8/25/2018 • 2 minutes to read • Edit Online

You might have to reference two versions of assemblies that have the same fully-qualified type names. For example, you might have to use two or more versions of an assembly in the same application. By using an external assembly alias, the namespaces from each assembly can be wrapped inside root-level namespaces named by the alias, which enables them to be used in the same file.

> **NOTE**
>
> The extern keyword is also used as a method modifier, declaring a method written in unmanaged code.

To reference two assemblies with the same fully-qualified type names, an alias must be specified at a command prompt, as follows:

`/r:GridV1=grid.dll`

`/r:GridV2=grid20.dll`

This creates the external aliases `GridV1` and `GridV2`. To use these aliases from within a program, reference them by using the `extern` keyword. For example:

`extern alias GridV1;`

`extern alias GridV2;`

Each extern alias declaration introduces an additional root-level namespace that parallels (but does not lie within) the global namespace. Thus types from each assembly can be referred to without ambiguity by using their fully qualified name, rooted in the appropriate namespace-alias.

In the previous example, `GridV1::Grid` would be the grid control from `grid.dll`, and `GridV2::Grid` would be the grid control from `grid20.dll`.

C# Language Specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See Also

- C# Reference
- C# Programming Guide
- C# Keywords
- Namespace Keywords
- :: Operator
- /reference (C# Compiler Options)

Operator Keywords (C# Reference)

8/25/2018 • 2 minutes to read • Edit Online

Used to perform miscellaneous actions such as creating objects, checking the run-time type of an object, obtaining the size of a type, and other actions. This section introduces the following keywords:

- as Converts an object to a compatible type.
- await Suspends an async method until an awaited task is completed.
- is Checks the run-time type of an object.
- new
  - new Operator Creates objects.
  - new Modifier Hides an inherited member.
  - new Constraint Qualifies a type parameter.
- nameof Obtains the simple (unqualified) string name of a variable, type, or member.
- sizeof Obtains the size of a type.
- typeof Obtains the **System.Type** object for a type.
- true
  - true Operator Returns the boolean value true to indicate true and returns false otherwise.
  - true Literal Represents the boolean value true.
- false
  - false Operator Returns the Boolean value true to indicate false and returns false otherwise.
  - false Literal Represents the boolean value false.
- stackalloc Allocates a block of memory on the stack.

The following keywords, which can be used as operators and as statements, are covered in the Statements section:

- checked Specifies checked context.
- unchecked Specifies unchecked context.

See Also

- C# Reference
- C# Programming Guide
- C# Keywords
- C# Operators

as (C# Reference)

8/25/2018 • 2 minutes to read • Edit Online

You can use the `as` operator to perform certain types of conversions between compatible reference types or nullable types. The following code shows an example.

```
class csrefKeywordsOperators
{
    class Base
    {
        public override string  ToString()
        {
           return "Base";
        }
    }
    class Derived : Base
    { }

    class Program
    {
        static void Main()
        {

            Derived d = new Derived();

            Base b = d as Base;
            if (b != null)
            {
                Console.WriteLine(b.ToString());
            }

        }
    }
}
```

Remarks

The `as` operator is like a cast operation. However, if the conversion isn't possible, `as` returns `null` instead of raising an exception. Consider the following example:

```
expression as type
```

The code is equivalent to the following expression except that the `expression` variable is evaluated only one time.

```
expression is type ? (type)expression : (type)null
```

Note that the `as` operator performs only reference conversions, nullable conversions, and boxing conversions. The `as` operator can't perform other conversions, such as user-defined conversions, which should instead be performed by using cast expressions.

Example

```
class ClassA { }
class ClassB { }

class MainClass
{
    static void Main()
    {
        object[] objArray = new object[6];
        objArray[0] = new ClassA();
        objArray[1] = new ClassB();
        objArray[2] = "hello";
        objArray[3] = 123;
        objArray[4] = 123.4;
        objArray[5] = null;

        for (int i = 0; i < objArray.Length; ++i)
        {
            string s = objArray[i] as string;
            Console.Write("{0}:", i);
            if (s != null)
            {
                Console.WriteLine("'" + s + "'");
            }
            else
            {
                Console.WriteLine("not a string");
            }
        }
    }
}
/*
Output:
0:not a string
1:not a string
2:'hello'
3:not a string
4:not a string
5:not a string
*/
```

C# Language Specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See Also

- C# Reference
- C# Programming Guide
- C# Keywords
- is
- ?: Operator
- Operator Keywords

await (C# Reference)

8/25/2018 • 4 minutes to read • Edit Online

The `await` operator is applied to a task in an asynchronous method to insert a suspension point in the execution of the method until the awaited task completes. The task represents ongoing work.

`await` can only be used in an asynchronous method modified by the async keyword. Such a method, defined by using the `async` modifier and usually containing one or more `await` expressions, is referred to as an *async method*.

> **NOTE**
>
> The `async` and `await` keywords were introduced in C# 5. For an introduction to async programming, see Asynchronous Programming with async and await.

The task to which the `await` operator is applied typically is returned by a call to a method that implements the Task-Based Asynchronous Pattern. They include methods that return Task, Task<TResult>, and `System.Threading.Tasks.ValueType<TResult>` objects.

In the following example, the HttpClient.GetByteArrayAsync method returns a `Task<byte[]>`. The task is a promise to produce the actual byte array when the task is complete. The `await` operator suspends execution until the work of the GetByteArrayAsync method is complete. In the meantime, control is returned to the caller of `GetPageSizeAsync`. When the task finishes execution, the `await` expression evaluates to a byte array.

```
using System;
using System.Net.Http;
using System.Threading;
using System.Threading.Tasks;

public class Example
{
   public static void Main()
   {
      string[] args = Environment.GetCommandLineArgs();
      if (args.Length > 1)
         GetPageSizeAsync(args[1]).Wait();
      else
         Console.WriteLine("Enter at least one URL on the command line.");
   }

   private static async Task GetPageSizeAsync(string url)
   {
       var client = new HttpClient();
       var uri = new Uri(Uri.EscapeUriString(url));
       byte[] urlContents = await client.GetByteArrayAsync(uri);
       Console.WriteLine($"{url}: {urlContents.Length/2:N0} characters");
   }
}
// The following call from the command line:
//    await1 http://docs.microsoft.com
// displays output like the following:
//   http://docs.microsoft.com: 7,967 characters
```

> **IMPORTANT**
>
> For the complete example, see Walkthrough: Accessing the Web by Using Async and Await. You can download the sample from Developer Code Samples on the Microsoft website. The example is in the AsyncWalkthrough_HttpClient project.

As shown in the previous example, if `await` is applied to the result of a method call that returns a `Task<TResult>`, then the type of the `await` expression is `TResult`. If `await` is applied to the result of a method call that returns a `Task`, then the type of the `await` expression is `void`. The following example illustrates the difference.

```
// await keyword used with a method that returns a Task<TResult>.
TResult result = await AsyncMethodThatReturnsTaskTResult();

// await keyword used with a method that returns a Task.
await AsyncMethodThatReturnsTask();

// await keyword used with a method that returns a ValueTask<TResult>.
TResult result = await AsyncMethodThatReturnsValueTaskTResult();
```

An `await` expression does not block the thread on which it is executing. Instead, it causes the compiler to sign up the rest of the async method as a continuation on the awaited task. Control then returns to the caller of the async method. When the task completes, it invokes its continuation, and execution of the async method resumes where it left off.

An `await` expression can occur only in the body of its enclosing method, lambda expression, or anonymous method, which must be marked with an `async` modifier. The term *await* serves as a keyword only in that context. Elsewhere, it is interpreted as an identifier. Within the method, lambda expression, or anonymous method, an `await` expression cannot occur in the body of a synchronous function, in a query expression, in the block of a lock statement, or in an unsafe context.

Exceptions

Most async methods return a Task or Task<TResult>. The properties of the returned task carry information about its status and history, such as whether the task is complete, whether the async method caused an exception or was canceled, and what the final result is. The `await` operator accesses those properties by calling methods on the object returned by the `GetAwaiter` method.

If you await a task-returning async method that causes an exception, the `await` operator rethrows the exception.

If you await a task-returning async method that's canceled, the `await` operator rethrows an OperationCanceledException.

A single task that is in a faulted state can reflect multiple exceptions. For example, the task might be the result of a call to Task.WhenAll. When you await such a task, the await operation rethrows only one of the exceptions. However, you can't predict which of the exceptions is rethrown.

For examples of error handling in async methods, see try-catch.

Example

The following example returns the total number of characters in the pages whose URLs are passed to it as command line arguments. The example calls the `GetPageLengthsAsync` method, which is marked with the `async` keyword. The `GetPageLengthsAsync` method in turn uses the `await` keyword to await calls to the HttpClient.GetStringAsync method.

```
using System;
using System.Net.Http;
using System.Threading;
using System.Threading.Tasks;

class Example
{
   static void Main()
   {
      string[] args = Environment.GetCommandLineArgs();
      if (args.Length < 2)
         throw new ArgumentNullException("No URLs specified on the command line.");

      long characters = GetPageLengthsAsync(args).Result;
      Console.WriteLine($"{args.Length - 1} pages, {characters:N0} characters");
   }

   private static async Task<long> GetPageLengthsAsync(string[] args)
   {
      var client = new HttpClient();
      long pageLengths = 0;

      for (int ctr = 1; ctr < args.Length; ctr++) {
         var uri = new Uri(Uri.EscapeUriString(args[ctr]));
         string pageContents = await client.GetStringAsync(uri);
         Interlocked.Add(ref pageLengths, pageContents.Length);
      }
      return pageLengths;
   }
}
```

Because the use of `async` and `await` in an application entry point is not supported, we cannot apply the `async` attribute to the `Main` method, nor can we await the `GetPageLengthsAsync` method call. We can ensure that the `Main` method waits for the async operation to complete by retrieving the value of the Task<TResult>.Result property. For tasks that do not return a value, you can call the Task.Wait method.

See also

- Asynchronous Programming with async and await
- Walkthrough: Accessing the Web by Using Async and Await
- async

is (C# Reference)

9/12/2018 • 10 minutes to read • Edit Online

Checks if an object is compatible with a given type, or (starting with C# 7.0) tests an expression against a pattern.

Testing for type compatibility

The `is` keyword evaluates type compatibility at runtime. It determines whether an object instance or the result of an expression can be converted to a specified type. It has the syntax

```
expr is type
```

where *expr* is an expression that evaluates to an instance of some type, and *type* is the name of the type to which the result of *expr* is to be converted. The `is` statement is `true` if *expr* is non-null and the object that results from evaluating the expression can be converted to *type*; otherwise, it returns `false`.

For example, the following code determines if `obj` can be cast to an instance of the `Person` type:

```
if (obj is Person) {
   // Do something if obj is a Person.
}
```

The `is` statement is true if:

- *expr* is an instance of the same type as *type*.
- *expr* is an instance of a type that derives from *type*. In other words, the result of *expr* can be upcast to an instance of *type*.
- *expr* has a compile-time type that is a base class of *type*, and *expr* has a runtime type that is *type* or is derived from *type*. The *compile-time type* of a variable is the variable's type as defined in its declaration. The *runtime type* of a variable is the type of the instance that is assigned to that variable.
- *expr* is an instance of a type that implements the *type* interface.

The following example shows that the `is` expression evaluates to `true` for each of these conversions.

```
using System;

public class Class1 : IFormatProvider
{
   public object GetFormat(Type t)
   {
      if (t.Equals(this.GetType()))
         return this;
      return null;
   }
}

public class Class2 : Class1
{
   public int Value { get; set; }
}

public class Example
{
   public static void Main()
   {
      var cl1 = new Class1();
      Console.WriteLine(cl1 is IFormatProvider);
      Console.WriteLine(cl1 is Object);
      Console.WriteLine(cl1 is Class1);
      Console.WriteLine(cl1 is Class2);
      Console.WriteLine();

      var cl2 = new Class2();
      Console.WriteLine(cl2 is IFormatProvider);
      Console.WriteLine(cl2 is Class2);
      Console.WriteLine(cl2 is Class1);
      Console.WriteLine();

      Class1 cl = cl2;
      Console.WriteLine(cl is Class1);
      Console.WriteLine(cl is Class2);
   }
}
// The example displays the following output:
//     True
//     True
//     True
//     False
//
//     True
//     True
//     True
//
//     True
//     True
```

The `is` keyword generates a compile-time warning if the expression is known to always be either `true` or `false`. It only considers reference conversions, boxing conversions, and unboxing conversions; it does not consider user-defined conversions or conversions defined by a type's implicit and explicit operators. The following example generates warnings because the result of the conversion is known at compile-time. Note that the `is` expression for conversions from `int` to `long` and `double` return false, since these conversions are handled by the implicit operator.

```
Console.WriteLine(3 is int);
Console.WriteLine();

int value = 6;
Console.WriteLine(value is long);
Console.WriteLine(value is double);
Console.WriteLine(value is object);
Console.WriteLine(value is ValueType);
Console.WriteLine(value is int);
// Compilation generates the following compiler warnings:
//   is2.cs(8,25): warning CS0183: The given expression is always of the provided ('int') type
//   is2.cs(12,25): warning CS0184: The given expression is never of the provided ('long') type
//   is2.cs(13,25): warning CS0184: The given expression is never of the provided ('double') type
//   is2.cs(14,25): warning CS0183: The given expression is always of the provided ('object') type
//   is2.cs(15,25): warning CS0183: The given expression is always of the provided ('ValueType') type
//   is2.cs(16,25): warning CS0183: The given expression is always of the provided ('int') type
```

`expr` can be any expression that returns a value, with the exception of anonymous methods and lambda expressions. The following example uses `is` to evaluate the return value of a method call.

```
using System;

public class Example
{
   public static void Main()
   {
      double number1 = 12.63;
      if (Math.Ceiling(number1) is double)
         Console.WriteLine("The expression returns a double.");
      else if (Math.Ceiling(number1) is decimal)
         Console.WriteLine("The expression returns a decimal.");

      decimal number2 = 12.63m;
      if (Math.Ceiling(number2) is double)
         Console.WriteLine("The expression returns a double.");
      else if (Math.Ceiling(number2) is decimal)
         Console.WriteLine("The expression returns a decimal.");

   }
}
// The example displays the following output:
//     The expression returns a double.
//     The expression returns a decimal.
```

Starting with C# 7.0, you can use pattern matching with the type pattern to write more concise code that uses the `is` statement.

Pattern matching with `is`

Starting with C# 7.0, the `is` and switch statements support pattern matching. The `is` keyword supports the following patterns:

- Type pattern, which tests whether an expression can be converted to a specified type and, if it can be, casts it to a variable of that type.
- Constant pattern, which tests whether an expression evaluates to a specified constant value.
- var pattern, a match that always succeeds and binds the value of an expression to a new local variable.

Type pattern

When using the type pattern to perform pattern matching, `is` tests whether an expression can be converted to a specified type and, if it can be, casts it to a variable of that type. It is a straightforward extension of the `is` statement that enables concise type evaluation and conversion. The general form of the `is` type pattern is:

```
expr is type varname
```

where *expr* is an expression that evaluates to an instance of some type, *type* is the name of the type to which the result of *expr* is to be converted, and *varname* is the object to which the result of *expr* is converted if the `is` test is `true`.

The `is` expression is `true` if *expr* is not `null`, and any of the following is true:

- *expr* is an instance of the same type as *type*.
- *expr* is an instance of a type that derives from *type*. In other words, the result of *expr* can be upcast to an instance of *type*.
- *expr* has a compile-time type that is a base class of *type*, and *expr* has a runtime type that is *type* or is derived from *type*. The *compile-time type* of a variable is the variable's type as defined in its declaration. The *runtime type* of a variable is the type of the instance that is assigned to that variable.
- *expr* is an instance of a type that implements the *type* interface.

If *expr* is `true` and `is` is used with an `if` statement, *varname* is assigned and has local scope within the `if` statement only.

The following example uses the `is` type pattern to provide the implementation of a type's IComparable.CompareTo(Object) method.

```
using System;

public class Employee : IComparable
{
    public String Name { get; set; }
    public int Id { get; set; }

    public int CompareTo(Object o)
    {
        if (o is Employee e)
        {
            return Name.CompareTo(e.Name);
        }
        throw new ArgumentException("o is not an Employee object.");
    }
}
```

Without pattern matching, this code might be written as follows. The use of type pattern matching produces more compact, readable code by eliminating the need to test whether the result of a conversion is a `null`.

```csharp
using System;

public class Employee : IComparable
{
    public String Name { get; set; }
    public int Id { get; set; }

    public int CompareTo(Object o)
    {
        var e = o as Employee;
        if (e == null)
        {
           throw new ArgumentException("o is not an Employee object.");
        }
        return Name.CompareTo(e.Name);
    }
}
```

The `is` type pattern also produces more compact code when determining the type of a value type. The following example uses the `is` type pattern to determine whether an object is a `Person` or a `Dog` instance before displaying the value of an appropriate property.

```
using System;

public class Example
{
   public static void Main()
   {
      Object o = new Person("Jane");
      ShowValue(o);

      o = new Dog("Alaskan Malamute");
      ShowValue(o);
   }

   public static void ShowValue(object o)
   {
      if (o is Person p) {
         Console.WriteLine(p.Name);
      }
      else if (o is Dog d) {
         Console.WriteLine(d.Breed);
      }
   }
}

public struct Person
{
   public string Name { get; set; }

   public Person(string name) : this()
   {
      Name = name;
   }
}

public struct Dog
{
   public string Breed { get; set; }

   public Dog(string breedName) : this()
   {
      Breed = breedName;
   }
}
// The example displays the following output:
// Jane
// Alaskan Malamute
```

The equivalent code without pattern matching requires a separate assignment that includes an explicit cast.

```csharp
using System;

public class Example
{
   public static void Main()
   {
      Object o = new Person("Jane");
      ShowValue(o);

      o = new Dog("Alaskan Malamute");
      ShowValue(o);
   }

   public static void ShowValue(object o)
   {
      if (o is Person) {
         Person p = (Person) o;
         Console.WriteLine(p.Name);
      }
      else if (o is Dog) {
         Dog d = (Dog) o;
         Console.WriteLine(d.Breed);
      }
   }
}

public struct Person
{
   public string Name { get; set; }

   public Person(string name) : this()
   {
      Name = name;
   }
}

public struct Dog
{
   public string Breed { get; set; }

   public Dog(string breedName) : this()
   {
      Breed = breedName;
   }
}
// The example displays the following output:
//       Jane
//       Alaskan Malamute
```

Constant pattern

When performing pattern matching with the constant pattern, `is` tests whether an expression equals a specified constant. In C# 6 and earlier versions, the constant pattern is supported by the switch statement. Starting with C# 7.0, it is supported by the `is` statement as well. Its syntax is:

```
expr is constant
```

where *expr* is the expression to evaluate, and *constant* is the value to test for. *constant* can be any of the following constant expressions:

- A literal value.
- The name of a declared `const` variable.

- An enumeration constant.

The constant expression is evaluated as follows:

- If *expr* and *constant* are integral types, the C# equality operator determines whether the expression returns `true` (that is, whether `expr == constant`).
- Otherwise, the value of the expression is determined by a call to the static Object.Equals(expr, constant) method.

The following example combines the type and constant patterns to test whether an object is a `Dice` instance and, if it is, to determine whether the value of a dice roll is 6.

```
using System;

public class Dice
{
    Random rnd = new Random();
    public Dice()
    {

    }
    public int Roll()
    {
        return rnd.Next(1, 7);
    }
}

class Program
{
    static void Main(string[] args)
    {
        var d1 = new Dice();
        ShowValue(d1);
    }

    private static void ShowValue(object o)
    {
        const int HIGH_ROLL = 6;

        if (o is Dice d && d.Roll() is HIGH_ROLL)
            Console.WriteLine($"The value is {HIGH_ROLL}!");
        else
            Console.WriteLine($"The dice roll is not a {HIGH_ROLL}!");
     }
}
// The example displays output like the following:
//      The value is 6!
```

Checking for `null` can be performed using the constant pattern. The `null` keyword is supported by the `is` statement. Its syntax is:

```
expr is null
```

The following example shows a comparison of `null` checks:

```
using System;

class Program
{
    static void Main(string[] args)
    {
        object o = null;

        if (o is null)
        {
            Console.WriteLine("o does not have a value");
        }
        else
        {
            Console.WriteLine($"o is {o}");
        }

        int? x = 10;

        if (x is null)
        {
            Console.WriteLine("x does not have a value");
        }
        else
        {
            Console.WriteLine($"x is {x.Value}");
        }

        // 'null' check comparison
        Console.WriteLine($"'is' constant pattern 'null' check result : { o is null }");
        Console.WriteLine($"object.ReferenceEquals 'null' check result : { object.ReferenceEquals(o, null)
}");
        Console.WriteLine($"Equality operator (==) 'null' check result : { o == null }");
    }

    // The example displays the following output:
    // o does not have a value
    // x is 10
    // 'is' constant pattern 'null' check result : True
    // object.ReferenceEquals 'null' check result : True
    // Equality operator (==) 'null' check result : True
}
```

var pattern

A pattern match with the var pattern always succeeds. Its syntax is

```
expr is var varname
```

where the value of *expr* is always assigned to a local variable named *varname*. *varname* is a static variable of the same type as *expr*. The following example uses the var pattern to assign an expression to a variable named `obj`. It then displays the value and the type of `obj`.

```
using System;

class Program
{
    static void Main()
   {
      object[] items = { new Book("The Tempest"), new Person("John") };
      foreach (var item in items) {
        if (item is var obj)
          Console.WriteLine($"Type: {obj.GetType().Name}, Value: {obj}");
      }
   }
}

class Book
{
    public Book(string title)
    {
       Title = title;
    }

    public string Title { get; set; }

    public override string ToString()
    {
       return Title;
    }
}

class Person
{
   public Person(string name)
   {
      Name = name;
   }

   public string Name
   { get; set; }

   public override string ToString()
   {
      return Name;
   }
}
// The example displays the following output:
//       Type: Book, Value: The Tempest
//       Type: Person, Value: John
```

Note that if *expr* is `null`, the `is` expression still is true and assigns `null` to *varname*.

C# Language Specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- C# Reference
- C# Keywords
- typeof
- as
- Operator Keywords

new (C# Reference)

8/25/2018 • 2 minutes to read • Edit Online

In C#, the `new` keyword can be used as an operator, a modifier, or a constraint.

new Operator
Used to create objects and invoke constructors.

new Modifier
Used to hide an inherited member from a base class member.

new Constraint
Used to restrict types that might be used as arguments for a type parameter in a generic declaration.

See Also

- C# Reference
- C# Programming Guide
- C# Keywords

new operator (C# Reference)

8/19/2018 • 2 minutes to read • Edit Online

Used to create objects and invoke constructors. For example:

```
Class1 obj  = new Class1();
```

It is also used to create instances of anonymous types:

```
var query = from cust in customers
            select new { Name = cust.Name, Address = cust.PrimaryAddress };
```

The `new` operator is also used to invoke the default constructor for value types. For example:

```
int i = new int();
```

In the preceding statement, `i` is initialized to `0`, which is the default value for the type `int`. The statement has the same effect as the following:

```
int i = 0;
```

For a complete list of default values, see Default Values Table.

Remember that it is an error to declare a default constructor for a struct because every value type implicitly has a public default constructor. It is possible to declare parameterized constructors on a struct type to set its initial values, but this is only necessary if values other than the default are required.

Both value-type objects such as structs and reference-type objects such as classes are destroyed automatically, but value-type objects are destroyed when their containing context is destroyed, whereas reference-type objects are destroyed by the garbage collector at an unspecified time after the last reference to them is removed. For types that contain resources such as file handles, or network connections, it is desirable to employ deterministic cleanup to ensure that the resources they contain are released as soon as possible. For more information, see using Statement.

The `new` operator cannot be overloaded.

If the `new` operator fails to allocate memory, it throws the exception, OutOfMemoryException.

Example

In the following example, a `struct` object and a class object are created and initialized by using the `new` operator and then assigned values. The default and the assigned values are displayed.

```
struct SampleStruct
{
   public int x;
   public int y;

   public SampleStruct(int x, int y)
   {
      this.x = x;
      this.y = y;
   }
}

class SampleClass
{
   public string name;
   public int id;

   public SampleClass() {}

   public SampleClass(int id, string name)
   {
      this.id = id;
      this.name = name;
   }
}

class ProgramClass
{
   static void Main()
   {
      // Create objects using default constructors:
      SampleStruct Location1 = new SampleStruct();
      SampleClass Employee1 = new SampleClass();

      // Display values:
      Console.WriteLine("Default values:");
      Console.WriteLine("   Struct members: {0}, {1}",
             Location1.x, Location1.y);
      Console.WriteLine("   Class members: {0}, {1}",
             Employee1.name, Employee1.id);

      // Create objects using parameterized constructors:
      SampleStruct Location2 = new SampleStruct(10, 20);
      SampleClass Employee2 = new SampleClass(1234, "Cristina Potra");

      // Display values:
      Console.WriteLine("Assigned values:");
      Console.WriteLine("   Struct members: {0}, {1}",
             Location2.x, Location2.y);
      Console.WriteLine("   Class members: {0}, {1}",
             Employee2.name, Employee2.id);
   }
}
/*
Output:
Default values:
   Struct members: 0, 0
   Class members: , 0
Assigned values:
   Struct members: 10, 20
   Class members: Cristina Potra, 1234
*/
```

Notice in the example that the default value of a string is `null`. Therefore, it is not displayed.

C# language specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- C# Reference
- C# Programming Guide
- C# Keywords
- Operator Keywords
- new
- Anonymous Types

new modifier (C# Reference)

8/19/2018 • 3 minutes to read • Edit Online

When used as a declaration modifier, the `new` keyword explicitly hides a member that is inherited from a base class. When you hide an inherited member, the derived version of the member replaces the base class version. Although you can hide members without using the `new` modifier, you get a compiler warning. If you use `new` to explicitly hide a member, it suppresses this warning.

To hide an inherited member, declare it in the derived class by using the same member name, and modify it with the `new` keyword. For example:

```
public class BaseC
{
    public int x;
    public void Invoke() { }
}
public class DerivedC : BaseC
{
    new public void Invoke() { }
}
```

In this example, `BaseC.Invoke` is hidden by `DerivedC.Invoke`. The field `x` is not affected because it is not hidden by a similar name.

Name hiding through inheritance takes one of the following forms:

Generally, a constant, field, property, or type that is introduced in a class or struct hides all base class members that share its name. There are special cases. For example, if you declare a new field with name `N` to have a type that is not invocable, and a base type declares `N` to be a method, the new field does not hide the base declaration in invocation syntax. See the C# 5.0 language specification for details (see section "Member Lookup" in section "Expressions").

A method introduced in a class or struct hides properties, fields, and types that share that name in the base class. It also hides all base class methods that have the same signature.

An indexer introduced in a class or struct hides all base class indexers that have the same signature.

It is an error to use both `new` and override on the same member, because the two modifiers have mutually exclusive meanings. The `new` modifier creates a new member with the same name and causes the original member to become hidden. The `override` modifier extends the implementation for an inherited member.

Using the `new` modifier in a declaration that does not hide an inherited member generates a warning.

Example

In this example, a base class, `BaseC`, and a derived class, `DerivedC`, use the same field name `x`, which hides the value of the inherited field. The example demonstrates the use of the `new` modifier. It also demonstrates how to access the hidden members of the base class by using their fully qualified names.

```
public class BaseC
{
    public static int x = 55;
    public static int y = 22;
}

public class DerivedC : BaseC
{
    // Hide field 'x'.
    new public static int x = 100;

    static void Main()
    {
        // Display the new value of x:
        Console.WriteLine(x);

        // Display the hidden value of x:
        Console.WriteLine(BaseC.x);

        // Display the unhidden member y:
        Console.WriteLine(y);
    }
}
/*
Output:
100
55
22
*/
```

Example

In this example, a nested class hides a class that has the same name in the base class. The example demonstrates how to use the `new` modifier to eliminate the warning message and how to access the hidden class members by using their fully qualified names.

```
public class BaseC
{
    public class NestedC
    {
        public int x = 200;
        public int y;
    }
}

public class DerivedC : BaseC
{
    // Nested type hiding the base type members.
    new public class NestedC
    {
        public int x = 100;
        public int y;
        public int z;
    }

    static void Main()
    {
        // Creating an object from the overlapping class:
        NestedC c1  = new NestedC();

        // Creating an object from the hidden class:
        BaseC.NestedC c2 = new BaseC.NestedC();

        Console.WriteLine(c1.x);
        Console.WriteLine(c2.x);
    }
}
/*
Output:
100
200
*/
```

If you remove the `new` modifier, the program will still compile and run, but you will get the following warning:

```
The keyword new is required on 'MyDerivedC.x' because it hides inherited member 'MyBaseC.x'.
```

C# language specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- C# Reference
- C# Programming Guide
- C# Keywords
- Operator Keywords
- Modifiers
- Versioning with the Override and New Keywords
- Knowing When to Use Override and New Keywords

new constraint (C# Reference)

8/19/2018 • 2 minutes to read • Edit Online

The `new` constraint specifies that any type argument in a generic class declaration must have a public parameterless constructor. To use the new constraint, the type cannot be abstract.

Example

Apply the `new` constraint to a type parameter when your generic class creates new instances of the type, as shown in the following example:

```
class ItemFactory<T> where T : new()
{
    public T GetNewItem()
    {
        return new T();
    }
}
```

Example

When you use the `new()` constraint with other constraints, it must be specified last:

```
public class ItemFactory2<T>
    where T : IComparable, new()
{
}
```

For more information, see Constraints on Type Parameters.

C# language specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- System.Collections.Generic
- C# Reference
- C# Programming Guide
- C# Keywords
- Operator Keywords
- Generics

sizeof (C# Reference)

8/30/2018 • 2 minutes to read • Edit Online

Used to obtain the size in bytes for an unmanaged type.

Unmanaged types include:

- The simple types that are listed in the following table:

EXPRESSION	CONSTANT VALUE
`sizeof(sbyte)`	1
`sizeof(byte)`	1
`sizeof(short)`	2
`sizeof(ushort)`	2
`sizeof(int)`	4
`sizeof(uint)`	4
`sizeof(long)`	8
`sizeof(ulong)`	8
`sizeof(char)`	2 (Unicode)
`sizeof(float)`	4
`sizeof(double)`	8
`sizeof(decimal)`	16
`sizeof(bool)`	1

- Enum types.
- Pointer types.
- User-defined structs that do not contain any instance fields or auto-implemented instance properties that are reference types or constructed types.

The following example shows how to retrieve the size of an `int`:

```
// Constant value 4:
int intSize = sizeof(int);
```

Remarks

Starting with version 2.0 of C#, applying `sizeof` to simple or enum types no longer requires that code be compiled in an unsafe context.

The `sizeof` operator cannot be overloaded. The values returned by the `sizeof` operator are of type `int`. The previous table shows the constant values that are substituted for `sizeof` expressions that have certain simple types as operands.

For all other types, including structs, the `sizeof` operator can be used only in unsafe code blocks. Although you can use the Marshal.SizeOf method, the value returned by this method is not always the same as the value returned by `sizeof`. Marshal.SizeOf returns the size after the type has been marshaled, whereas `sizeof` returns the size as it has been allocated by the common language runtime, including any padding.

Example

```
class MainClass
{
    // unsafe not required for primitive types
    static void Main()
    {
        Console.WriteLine("The size of short is {0}.", sizeof(short));
        Console.WriteLine("The size of int is {0}.", sizeof(int));
        Console.WriteLine("The size of long is {0}.", sizeof(long));
    }
}
/*
Output:
    The size of short is 2.
    The size of int is 4.
    The size of long is 8.
*/
```

C# Language Specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See Also

- C# Reference
- C# Programming Guide
- C# Keywords
- Operator Keywords
- enum
- Unsafe Code and Pointers
- Structs
- Constants

typeof (C# Reference)

8/29/2018 • 2 minutes to read • Edit Online

Used to obtain the `System.Type` object for a type. A `typeof` expression takes the following form:

```
System.Type type = typeof(int);
```

Remarks

To obtain the run-time type of an expression, you can use the .NET Framework method GetType, as in the following example:

```
int i = 0;
System.Type type = i.GetType();
```

The `typeof` operator cannot be overloaded.

The `typeof` operator can also be used on open generic types. Types with more than one type parameter must have the appropriate number of commas in the specification. The following example shows how to determine whether the return type of a method is a generic IEnumerable<T>. Type.GetInterface will return `null` if the return type is not an IEnumerable<T> generic type.

```
using System;
using System.Collections.Generic;
using System.Reflection;

class Example
{
    static void Main()
    {
        MethodInfo method = typeof(string).GetMethod("Copy");
        Type t = method.ReturnType.GetInterface(typeof(IEnumerable<>).Name);
        if (t != null)
            Console.WriteLine(t);
        else
            Console.WriteLine("The return type is not IEnumerable<T>.");
        /*
        Output:
        System.Collections.Generic.IEnumerable`1[System.Char]
        */
    }
}
```

Example

```
public class ExampleClass
{
   public int sampleMember;
   public void SampleMethod() {}

   static void Main()
   {
      Type t = typeof(ExampleClass);
      // Alternatively, you could use
      // ExampleClass obj = new ExampleClass();
      // Type t = obj.GetType();

      Console.WriteLine("Methods:");
      System.Reflection.MethodInfo[] methodInfo = t.GetMethods();

      foreach (System.Reflection.MethodInfo mInfo in methodInfo)
         Console.WriteLine(mInfo.ToString());

      Console.WriteLine("Members:");
      System.Reflection.MemberInfo[] memberInfo = t.GetMembers();

      foreach (System.Reflection.MemberInfo mInfo in memberInfo)
         Console.WriteLine(mInfo.ToString());
   }
}
/*
 Output:
    Methods:
    Void SampleMethod()
    System.String ToString()
    Boolean Equals(System.Object)
    Int32 GetHashCode()
    System.Type GetType()
    Members:
    Void SampleMethod()
    System.String ToString()
    Boolean Equals(System.Object)
    Int32 GetHashCode()
    System.Type GetType()
    Void .ctor()
    Int32 sampleMember
*/
```

Example

This sample uses the GetType method to determine the type that is used to contain the result of a numeric calculation. This depends on the storage requirements of the resulting number.

```
class GetTypeTest
{
    static void Main()
    {
        int radius = 3;
        Console.WriteLine("Area = {0}", radius * radius * Math.PI);
        Console.WriteLine("The type is {0}",
                          (radius * radius * Math.PI).GetType()
        );
    }
}
/*
Output:
Area = 28.2743338823081
The type is System.Double
*/
```

C# Language Specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See Also

- System.Type
- C# Reference
- C# Programming Guide
- C# Keywords
- is
- Operator Keywords

true (C# Reference)

8/25/2018 • 2 minutes to read • Edit Online

Used as an overloaded operator or as a literal:

true Operator

true Literal

See Also

- C# Reference
- C# Programming Guide
- C# Keywords

true Operator (C# Reference)

8/25/2018 • 3 minutes to read • Edit Online

Returns the bool value `true` to indicate that an operand is true and returns `false` otherwise.

Prior to C# 2.0, the `true` and `false` operators were used to create user-defined nullable value types that were compatible with types such as `SqlBool`. However, the language now provides built-in support for nullable value types, and whenever possible you should use those instead of overloading the `true` and `false` operators. For more information, see Nullable Types.

With nullable Booleans, the expression `a != b` is not necessarily equal to `!(a == b)` because one or both of the values might be null. You need to overload both the `true` and `false` operators separately to correctly identify the null values in the expression. The following example shows how to overload and use the `true` and `false` operators.

```
// For example purposes only. Use the built-in nullable bool
// type (bool?) whenever possible.
public struct DBBool
{
    // The three possible DBBool values.
    public static readonly DBBool Null = new DBBool(0);
    public static readonly DBBool False = new DBBool(-1);
    public static readonly DBBool True = new DBBool(1);
    // Private field that stores –1, 0, 1 for False, Null, True.
    sbyte value;
    // Private instance constructor. The value parameter must be –1, 0, or 1.
    DBBool(int value)
    {
        this.value = (sbyte)value;
    }
    // Properties to examine the value of a DBBool. Return true if this
    // DBBool has the given value, false otherwise.
    public bool IsNull { get { return value == 0; } }
    public bool IsFalse { get { return value < 0; } }
    public bool IsTrue { get { return value > 0; } }
    // Implicit conversion from bool to DBBool. Maps true to DBBool.True and
    // false to DBBool.False.
    public static implicit operator DBBool(bool x)
    {
        return x ? True : False;
    }
    // Explicit conversion from DBBool to bool. Throws an exception if the
    // given DBBool is Null; otherwise returns true or false.
    public static explicit operator bool(DBBool x)
    {
        if (x.value == 0) throw new InvalidOperationException();
        return x.value > 0;
    }
    // Equality operator. Returns Null if either operand is Null; otherwise
    // returns True or False.
    public static DBBool operator ==(DBBool x, DBBool y)
    {
        if (x.value == 0 || y.value == 0) return Null;
        return x.value == y.value ? True : False;
    }
    // Inequality operator. Returns Null if either operand is Null; otherwise
    // returns True or False.
    public static DBBool operator !=(DBBool x, DBBool y)
    {
        if (x.value == 0 || y.value == 0) return Null;
```

```
            return x.value != y.value ? True : False;
        }
        // Logical negation operator. Returns True if the operand is False, Null
        // if the operand is Null, or False if the operand is True.
        public static DBBool operator !(DBBool x)
        {
            return new DBBool(-x.value);
        }
        // Logical AND operator. Returns False if either operand is False,
        // Null if either operand is Null, otherwise True.
        public static DBBool operator &(DBBool x, DBBool y)
        {
            return new DBBool(x.value < y.value ? x.value : y.value);
        }
        // Logical OR operator. Returns True if either operand is True,
        // Null if either operand is Null, otherwise False.
        public static DBBool operator |(DBBool x, DBBool y)
        {
            return new DBBool(x.value > y.value ? x.value : y.value);
        }
        // Definitely true operator. Returns true if the operand is True, false
        // otherwise.
        public static bool operator true(DBBool x)
        {
            return x.value > 0;
        }
        // Definitely false operator. Returns true if the operand is False, false
        // otherwise.
        public static bool operator false(DBBool x)
        {
            return x.value < 0;
        }
        public override bool Equals(object obj)
        {
            if (!(obj is DBBool)) return false;
            return value == ((DBBool)obj).value;
        }
        public override int GetHashCode()
        {
            return value;
        }
        public override string ToString()
        {
            if (value > 0) return "DBBool.True";
            if (value < 0) return "DBBool.False";
            return "DBBool.Null";
        }
    }
```

A type that overloads the `true` and `false` operators can be used for the controlling expression in if, do, while, and for statements and in conditional expressions.

If a type defines operator `true`, it must also define operator false.

A type cannot directly overload the conditional logical operators (&& and ||), but an equivalent effect can be achieved by overloading the regular logical operators and operators `true` and `false`.

C# Language Specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See Also

- C# Reference
- C# Programming Guide
- C# Keywords
- C# Operators
- false

true Literal (C# Reference)

8/25/2018 • 2 minutes to read • Edit Online

Represents the boolean value true.

Example

```
class TrueTest
{
    static void Main()
    {
        bool a = true;
        Console.WriteLine( a ? "yes" : "no" );
    }
}
/*
Output:
yes
*/
```

C# Language Specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See Also

- C# Reference
- C# Programming Guide
- C# Keywords
- false

false (C# Reference)

8/25/2018 • 2 minutes to read • Edit Online

Used as an overloaded operator or as a literal:

- false Operator
- false Literal

See Also

- C# Reference
- C# Programming Guide
- C# Keywords

false operator (C# Reference)

8/25/2018 • 3 minutes to read • Edit Online

Returns the bool value `true` to indicate that an operand is `false` and returns `false` otherwise.

Prior to C# 2.0, the `true` and `false` operators were used to create user-defined nullable value types that were compatible with types such as `SqlBool`. However, the language now provides built-in support for nullable value types, and whenever possible you should use those instead of overloading the `true` and `false` operators. For more information, see Nullable Types.

With nullable Booleans, the expression `a != b` is not necessarily equal to `!(a == b)` because one or both of the values might be null. You have to overload both the `true` and `false` operators separately to correctly handle the null values in the expression. The following example shows how to overload and use the `true` and `false` operators.

```
// For example purposes only. Use the built-in nullable bool
// type (bool?) whenever possible.
public struct DBBool
{
    // The three possible DBBool values.
    public static readonly DBBool Null = new DBBool(0);
    public static readonly DBBool False = new DBBool(-1);
    public static readonly DBBool True = new DBBool(1);
    // Private field that stores -1, 0, 1 for False, Null, True.
    sbyte value;
    // Private instance constructor. The value parameter must be -1, 0, or 1.
    DBBool(int value)
    {
        this.value = (sbyte)value;
    }
    // Properties to examine the value of a DBBool. Return true if this
    // DBBool has the given value, false otherwise.
    public bool IsNull { get { return value == 0; } }
    public bool IsFalse { get { return value < 0; } }
    public bool IsTrue { get { return value > 0; } }
    // Implicit conversion from bool to DBBool. Maps true to DBBool.True and
    // false to DBBool.False.
    public static implicit operator DBBool(bool x)
    {
        return x ? True : False;
    }
    // Explicit conversion from DBBool to bool. Throws an exception if the
    // given DBBool is Null; otherwise returns true or false.
    public static explicit operator bool(DBBool x)
    {
        if (x.value == 0) throw new InvalidOperationException();
        return x.value > 0;
    }
    // Equality operator. Returns Null if either operand is Null; otherwise
    // returns True or False.
    public static DBBool operator ==(DBBool x, DBBool y)
    {
        if (x.value == 0 || y.value == 0) return Null;
        return x.value == y.value ? True : False;
    }
    // Inequality operator. Returns Null if either operand is Null; otherwise
    // returns True or False.
    public static DBBool operator !=(DBBool x, DBBool y)
    {
        if (x.value == 0 || y.value == 0) return Null;
```

```
        return x.value != y.value ? True : False;
    }
    // Logical negation operator. Returns True if the operand is False, Null
    // if the operand is Null, or False if the operand is True.
    public static DBBool operator !(DBBool x)
    {
        return new DBBool(-x.value);
    }
    // Logical AND operator. Returns False if either operand is False,
    // Null if either operand is Null, otherwise True.
    public static DBBool operator &(DBBool x, DBBool y)
    {
        return new DBBool(x.value < y.value ? x.value : y.value);
    }
    // Logical OR operator. Returns True if either operand is True,
    // Null if either operand is Null, otherwise False.
    public static DBBool operator |(DBBool x, DBBool y)
    {
        return new DBBool(x.value > y.value ? x.value : y.value);
    }
    // Definitely true operator. Returns true if the operand is True, false
    // otherwise.
    public static bool operator true(DBBool x)
    {
        return x.value > 0;
    }
    // Definitely false operator. Returns true if the operand is False, false
    // otherwise.
    public static bool operator false(DBBool x)
    {
        return x.value < 0;
    }
    public override bool Equals(object obj)
    {
        if (!(obj is DBBool)) return false;
        return value == ((DBBool)obj).value;
    }
    public override int GetHashCode()
    {
        return value;
    }
    public override string ToString()
    {
        if (value > 0) return "DBBool.True";
        if (value < 0) return "DBBool.False";
        return "DBBool.Null";
    }
}
```

A type that overloads the `true` and `false` operators can be used for the controlling expression in if, do, while, and for statements and in conditional expressions.

If a type defines operator `false`, it must also define operator true.

A type cannot directly overload the conditional logical operators && and ||, but an equivalent effect can be achieved by overloading the regular logical operators and operators `true` and `false`.

C# language specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- C# Reference
- C# Programming Guide
- C# Keywords
- C# Operators
- true

false literal (C# Reference)

8/25/2018 • 2 minutes to read • Edit Online

Represents the boolean value false.

Example

```
class TestClass
{
    static void Main()
    {
        bool a = false;
        Console.WriteLine( a ? "yes" : "no" );
    }
}
// Output: no
```

C# language specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- C# Reference
- C# Programming Guide
- C# Keywords
- true

stackalloc (C# Reference)

8/25/2018 • 2 minutes to read • Edit Online

The `stackalloc` keyword is used in an unsafe code context to allocate a block of memory on the stack.

```
int* block = stackalloc int[100];
```

Remarks

The keyword is valid only in local variable initializers. The following code causes compiler errors.

```
int* block;
// The following assignment statement causes compiler errors. You
// can use stackalloc only when declaring and initializing a local
// variable.
block = stackalloc int[100];
```

Beginning with C# 7.3, you can use array initializer syntax for `stackalloc` arrays. All the following declarations declare an array with three elements whose values are the integers `1`, `2`, and `3`:

```
// Valid starting with C# 7.3
int* first = stackalloc int[3] { 1, 2, 3 };
int* second = stackalloc int[] { 1, 2, 3 };
int* third = stackalloc[] { 1, 2, 3 };
```

Because pointer types are involved, `stackalloc` requires an unsafe context. For more information, see Unsafe Code and Pointers

`stackalloc` is like _alloca in the C run-time library.

Examples

The following example calculates and displays the first 20 numbers in the Fibonacci sequence. Each number is the sum of the previous two numbers. In the code, a block of memory of sufficient size to contain 20 elements of type `int` is allocated on the stack, not the heap. The address of the block is stored in the pointer `fib`. This memory is not subject to garbage collection and therefore does not have to be pinned (by using fixed). The lifetime of the memory block is limited to the lifetime of the method that defines it. You cannot free the memory before the method returns.

```
const int arraySize = 20;
int* fib = stackalloc int[arraySize];
int* p = fib;
// The sequence begins with 1, 1.
*p++ = *p++ = 1;
for (int i = 2; i < arraySize; ++i, ++p)
{
    // Sum the previous two numbers.
    *p = p[-1] + p[-2];
}
for (int i = 0; i < arraySize; ++i)
{
    Console.WriteLine(fib[i]);
}
/* Output:
   1
   1
   2
   3
   5
   8
   13
   21
   34
   55
   89
   144
   233
   377
   610
   987
   1597
   2584
   4181
   6765
 */
```

The following example initializes a `stackalloc` array of integers to a bit mask with one bit set in each element. This demonstrates the new initializer syntax available starting in C# 7.3:

```
int* mask = stackalloc[] {
    0b_0000_0000_0000_0001,
    0b_0000_0000_0000_0010,
    0b_0000_0000_0000_0100,
    0b_0000_0000_0000_1000,
    0b_0000_0000_0001_0000,
    0b_0000_0000_0010_0000,
    0b_0000_0000_0100_0000,
    0b_0000_0000_1000_0000,
    0b_0000_0001_0000_0000,
    0b_0000_0010_0000_0000,
    0b_0000_0100_0000_0000,
    0b_0000_1000_0000_0000,
    0b_0001_0000_0000_0000,
    0b_0010_0000_0000_0000,
    0b_0100_0000_0000_0000,
    0b_1000_0000_0000_0000
};

for (int i = 0; i < 16; i++)
    Console.WriteLine(mask[i]);
/* Output:
   1
   2
   4
   8
   16
   32
   64
   128
   256
   512
   1024
   2048
   4096
   8192
   16384
   32768
 */
```

Security

Unsafe code is less secure than safe alternatives. However, the use of `stackalloc` automatically enables buffer overrun detection features in the common language runtime (CLR). If a buffer overrun is detected, the process is terminated as quickly as possible to minimize the chance that malicious code is executed.

C# Language Specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See Also

- C# Reference
- C# Programming Guide
- C# Keywords
- Operator Keywords
- Unsafe Code and Pointers

nameof (C# Reference)

8/25/2018 • 3 minutes to read • Edit Online

Used to obtain the simple (unqualified) string name of a variable, type, or member.

When reporting errors in code, hooking up model-view-controller (MVC) links, firing property changed events, etc., you often want to capture the string name of a method. Using `nameof` helps keep your code valid when renaming definitions. Before, you had to use string literals to refer to definitions, which is brittle when renaming code elements because tools do not know to check these string literals.

A `nameof` expression has this form:

```
if (x == null) throw new ArgumentNullException(nameof(x));
WriteLine(nameof(person.Address.ZipCode)); // prints "ZipCode"
```

Key Use Cases

These examples show the key use cases for `nameof`.

Validate parameters:

```
void f(string s) {
   if (s == null) throw new ArgumentNullException(nameof(s));
}
```

MVC Action links:

```
<%= Html.ActionLink("Sign up",
            @typeof(UserController),
            @nameof(UserController.SignUp))
%>
```

INotifyPropertyChanged:

```
int p {
   get { return this.p; }
   set { this.p = value; PropertyChanged(this, new PropertyChangedEventArgs(nameof(this.p)); } // nameof(p)
works too
}
```

XAML dependency property:

```
public static DependencyProperty AgeProperty = DependencyProperty.Register(nameof(Age), typeof(int),
typeof(C));
```

Logging:

```
void f(int i) {
   Log(nameof(f), "method entry");
}
```

Attributes:

```
[DebuggerDisplay("={" + nameof(GetString) + "()}")]
class C {
   string GetString() { }
}
```

Examples

Some C# examples:

```
using Stuff = Some.Cool.Functionality
class C {
    static int Method1 (string x, int y) {}
    static int Method1 (string x, string y) {}
    int Method2 (int z) {}
    string f<T>() => nameof(T);
}

var c = new C()

class Test {
    static void Main (string[] args) {
        Console.WriteLine(nameof(C)); // -> "C"
        Console.WriteLine(nameof(C.Method1)); // -> "Method1"
        Console.WriteLine(nameof(C.Method2)); // -> "Method2"
        Console.WriteLine(nameof(c.Method1)); // -> "Method1"
        Console.WriteLine(nameof(c.Method2)); // -> "Method2"
        // Console.WriteLine(nameof(z)); -> "z" [inside of Method2 ok, inside Method1 is a compiler error]
        Console.WriteLine(nameof(Stuff)); // -> "Stuff"
        // Console.WriteLine(nameof(T)); -> "T" [works inside of method but not in attributes on the method]
        Console.WriteLine(nameof(f)); // -> "f"
        // Console.WriteLine(nameof(f<T>)); -> [syntax error]
        // Console.WriteLine(nameof(f<>)); -> [syntax error]
        // Console.WriteLine(nameof(Method2())); -> [error "This expression does not have a name"]
    }
}
```

Remarks

The argument to `nameof` must be a simple name, qualified name, member access, base access with a specified member, or this access with a specified member. The argument expression identifies a code definition, but it is never evaluated.

Because the argument needs to be an expression syntactically, there are many things disallowed that are not useful to list. The following are worth mentioning that produce errors: predefined types (for example, `int` or `void`), nullable types (`Point?`), array types (`Customer[,]`), pointer types (`Buffer*`), qualified alias (`A::B`), and unbound generic types (`Dictionary<,>`), preprocessing symbols (`DEBUG`), and labels (`loop:`).

If you need to get the fully-qualified name, you can use the `typeof` expression along with `nameof`. For example:

```
class C {
    void f(int i) {
        Log($"{typeof(C)}.{nameof(f)}", "method entry");
    }
}
```

Unfortunately `typeof` is not a constant expression like `nameof`, so `typeof` cannot be used in conjunction with `nameof` in all the same places as `nameof`. For example, the following would cause a CS0182 compile error:

```
[DebuggerDisplay("={" + typeof(C) + nameof(GetString) + "()}")]
class C {
   string GetString() { }
}
```

In the examples you see that you can use a type name and access an instance method name. You do not need to have an instance of the type, as required in evaluated expressions. Using the type name can be very convenient in some situations, and since you are just referring to the name and not using instance data, you do not need to contrive an instance variable or expression.

You can reference the members of a class in attribute expressions on the class.

There is no way to get a signatures information such as "`Method1 (str, str)`". One way to do that is to use an Expression, `Expression e = () => A.B.Method1("s1", "s2")`, and pull the MemberInfo from the resulting expression tree.

Language Specifications

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See Also

- C# Reference
- C# Programming Guide
- typeof

Conversion Keywords (C# Reference)

8/25/2018 • 2 minutes to read • Edit Online

This section describes keywords used in type conversions:

- explicit
- implicit
- operator

See Also

- C# Reference
- C# Programming Guide
- C# Keywords

explicit (C# Reference)

8/29/2018 • 2 minutes to read • Edit Online

The `explicit` keyword declares a user-defined type conversion operator that must be invoked with a cast.

The following example defines the operator that converts from a `Fahrenheit` class to a `Celsius` class. The operator must be defined either inside a `Fahrenheit` class or a `Celsius` class:

```
public static explicit operator Celsius(Fahrenheit fahr)
{
    return new Celsius((5.0f / 9.0f) * (fahr.Degrees - 32));
}
```

You invoke the defined conversion operator with a cast, as the following example shows:

```
Fahrenheit fahr = new Fahrenheit(100.0f);
Console.Write($"{fahr.Degrees} Fahrenheit");
Celsius c = (Celsius)fahr;
```

The conversion operator converts from a source type to a target type. The source type provides the conversion operator. Unlike implicit conversion, explicit conversion operators must be invoked by means of a cast. If a conversion operation can cause exceptions or lose information, you should mark it `explicit`. This prevents the compiler from silently invoking the conversion operation with possibly unforeseen consequences.

Omitting the cast results in compile-time error CS0266.

For more information, see Using Conversion Operators.

Example

The following example provides a `Fahrenheit` and a `Celsius` class, each of which provides an explicit conversion operator to the other class.

```
class Celsius
{
    public Celsius(float temp)
    {
        Degrees = temp;
    }

    public float Degrees { get; }

    public static explicit operator Fahrenheit(Celsius c)
    {
        return new Fahrenheit((9.0f / 5.0f) * c.Degrees + 32);
    }
}

class Fahrenheit
{
    public Fahrenheit(float temp)
    {
        Degrees = temp;
    }

    public float Degrees { get; }

    public static explicit operator Celsius(Fahrenheit fahr)
    {
        return new Celsius((5.0f / 9.0f) * (fahr.Degrees - 32));
    }
}

class MainClass
{
    static void Main()
    {
        Fahrenheit fahr = new Fahrenheit(100.0f);
        Console.Write($"{fahr.Degrees} Fahrenheit");
        Celsius c = (Celsius)fahr;

        Console.Write($" = {c.Degrees} Celsius");
        Fahrenheit fahr2 = (Fahrenheit)c;
        Console.WriteLine($" = {fahr2.Degrees} Fahrenheit");
    }
}
// Output:
// 100 Fahrenheit = 37.77778 Celsius = 100 Fahrenheit
```

Example

The following example defines a struct, `Digit`, that represents a single decimal digit. An operator is defined for conversions from `byte` to `Digit`, but because not all bytes can be converted to a `Digit`, the conversion is explicit.

```
struct Digit
{
    byte value;
    public Digit(byte value)
    {
        if (value > 9)
        {
            throw new ArgumentException();
        }
        this.value = value;
    }

    // Define explicit byte-to-Digit conversion operator:
    public static explicit operator Digit(byte b)
    {
        Digit d = new Digit(b);
        Console.WriteLine("conversion occurred");
        return d;
    }
}

class ExplicitTest
{
    static void Main()
    {
        try
        {
            byte b = 3;
            Digit d = (Digit)b; // explicit conversion
        }
        catch (Exception e)
        {
            Console.WriteLine("{0} Exception caught.", e);
        }
    }
}
/*
Output:
conversion occurred
*/
```

C# language specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- C# Reference
- C# Programming Guide
- C# Keywords
- implicit
- operator (C# Reference)
- How to: Implement User-Defined Conversions Between Structs
- Chained user-defined explicit conversions in C#

implicit (C# Reference)

8/25/2018 • 2 minutes to read • Edit Online

The `implicit` keyword is used to declare an implicit user-defined type conversion operator. Use it to enable implicit conversions between a user-defined type and another type, if the conversion is guaranteed not to cause a loss of data.

Example

```
class Digit
{
    public Digit(double d) { val = d; }
    public double val;
    // ...other members

    // User-defined conversion from Digit to double
    public static implicit operator double(Digit d)
    {
        return d.val;
    }
    //  User-defined conversion from double to Digit
    public static implicit operator Digit(double d)
    {
        return new Digit(d);
    }
}

class Program
{
    static void Main(string[] args)
    {
        Digit dig = new Digit(7);
        //This call invokes the implicit "double" operator
        double num = dig;
        //This call invokes the implicit "Digit" operator
        Digit dig2 = 12;
        Console.WriteLine("num = {0} dig2 = {1}", num, dig2.val);
        Console.ReadLine();
    }
}
```

By eliminating unnecessary casts, implicit conversions can improve source code readability. However, because implicit conversions do not require programmers to explicitly cast from one type to the other, care must be taken to prevent unexpected results. In general, implicit conversion operators should never throw exceptions and never lose information so that they can be used safely without the programmer's awareness. If a conversion operator cannot meet those criteria, it should be marked `explicit`. For more information, see Using Conversion Operators.

C# language specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- C# Reference
- C# Programming Guide
- C# Keywords
- explicit
- operator (C# Reference)
- How to: Implement User-Defined Conversions Between Structs

operator (C# Reference)

8/28/2018 • 2 minutes to read • Edit Online

Use the `operator` keyword to overload a built-in operator or to provide a user-defined conversion in a class or struct declaration.

To overload an operator on a custom class or struct, you create an operator declaration in the corresponding type. The operator declaration that overloads a built-in C# operator must satisfy the following rules:

- It includes both a `public` and a `static` modifier.
- It includes `operator X` where `X` is the name or symbol of the operator being overloaded.
- Unary operators have one parameter, and binary operators have two parameters. In each case, at least one parameter must be the same type as the class or struct that declares the operator.

For information about how to define conversion operators, see the explicit and implicit keyword articles.

For an overview of the C# operators that can be overloaded, see the Overloadable operators article.

Example

The following example defines a `Fraction` type that represents fractional numbers. It overloads the `+` and `*` operators to perform fractional addition and multiplication, and also provides a conversion operator that converts a `Fraction` type to a `double` type.

```
class Fraction
{
    int num, den;
    public Fraction(int num, int den)
    {
        this.num = num;
        this.den = den;
    }

    // overload operator +
    public static Fraction operator +(Fraction a, Fraction b)
    {
        return new Fraction(a.num * b.den + b.num * a.den,
           a.den * b.den);
    }

    // overload operator *
    public static Fraction operator *(Fraction a, Fraction b)
    {
        return new Fraction(a.num * b.num, a.den * b.den);
    }

    // user-defined conversion from Fraction to double
    public static implicit operator double(Fraction f)
    {
        return (double)f.num / f.den;
    }
}

class Test
{
    static void Main()
    {
        Fraction a = new Fraction(1, 2);
        Fraction b = new Fraction(3, 7);
        Fraction c = new Fraction(2, 3);
        Console.WriteLine((double)(a * b + c));
    }
}
/*
Output
0.880952380952381
*/
```

C# language specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- C# Reference
- C# Programming Guide
- C# Keywords
- implicit
- explicit
- Overloadable operators
- How to: Implement User-Defined Conversions Between Structs

Access Keywords (C# Reference)

8/25/2018 • 2 minutes to read • Edit Online

This section introduces the following access keywords:

- base

  Accesses the members of the base class.

- this

  Refers to the current instance of the class.

See Also

- C# Reference
- C# Programming Guide
- Access Modifiers
- C# Keywords

base (C# Reference)

8/25/2018 • 2 minutes to read • Edit Online

The `base` keyword is used to access members of the base class from within a derived class:

- Call a method on the base class that has been overridden by another method.
- Specify which base-class constructor should be called when creating instances of the derived class.

A base class access is permitted only in a constructor, an instance method, or an instance property accessor.

It is an error to use the `base` keyword from within a static method.

The base class that is accessed is the base class specified in the class declaration. For example, if you specify `class ClassB : ClassA`, the members of ClassA are accessed from ClassB, regardless of the base class of ClassA.

Example

In this example, both the base class, `Person`, and the derived class, `Employee`, have a method named `Getinfo`. By using the `base` keyword, it is possible to call the `Getinfo` method on the base class, from within the derived class.

```
public class Person
{
    protected string ssn = "444-55-6666";
    protected string name = "John L. Malgraine";

    public virtual void GetInfo()
    {
        Console.WriteLine("Name: {0}", name);
        Console.WriteLine("SSN: {0}", ssn);
    }
}
class Employee : Person
{
    public string id = "ABC567EFG";
    public override void GetInfo()
    {
        // Calling the base class GetInfo method:
        base.GetInfo();
        Console.WriteLine("Employee ID: {0}", id);
    }
}

class TestClass
{
    static void Main()
    {
        Employee E = new Employee();
        E.GetInfo();
    }
}
/*
Output
Name: John L. Malgraine
SSN: 444-55-6666
Employee ID: ABC567EFG
*/
```

For additional examples, see new, virtual, and override.

Example

This example shows how to specify the base-class constructor called when creating instances of a derived class.

```
public class BaseClass
{
    int num;

    public BaseClass()
    {
        Console.WriteLine("in BaseClass()");
    }

    public BaseClass(int i)
    {
        num = i;
        Console.WriteLine("in BaseClass(int i)");
    }

    public int GetNum()
    {
        return num;
    }
}

public class DerivedClass : BaseClass
{
    // This constructor will call BaseClass.BaseClass()
    public DerivedClass() : base()
    {
    }

    // This constructor will call BaseClass.BaseClass(int i)
    public DerivedClass(int i) : base(i)
    {
    }

    static void Main()
    {
        DerivedClass md = new DerivedClass();
        DerivedClass md1 = new DerivedClass(1);
    }
}
/*
Output:
in BaseClass()
in BaseClass(int i)
*/
```

C# language specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- C# Reference
- C# Programming Guide
- C# Keywords
- this

this (C# Reference)

8/25/2018 • 2 minutes to read • Edit Online

The `this` keyword refers to the current instance of the class and is also used as a modifier of the first parameter of an extension method.

> **NOTE**
>
> This article discusses the use of `this` with class instances. For more information about its use in extension methods, see Extension Methods.

The following are common uses of `this` :

- To qualify members hidden by similar names, for example:

```
public Employee(string name, string alias)
{
    // Use this to qualify the fields, name and alias:
    this.name = name;
    this.alias = alias;
}
```

- To pass an object as a parameter to other methods, for example:

```
CalcTax(this);
```

- To declare indexers, for example:

```
public int this[int param]
{
    get { return array[param]; }
    set { array[param] = value; }
}
```

Static member functions, because they exist at the class level and not as part of an object, do not have a `this` pointer. It is an error to refer to `this` in a static method.

Example

In this example, `this` is used to qualify the `Employee` class members, `name` and `alias` , which are hidden by similar names. It is also used to pass an object to the method `CalcTax` , which belongs to another class.

```
class Employee
{
    private string name;
    private string alias;
    private decimal salary = 3000.00m;

    // Constructor:
    public Employee(string name, string alias)
    {
        // Use this to qualify the fields, name and alias:
        this.name = name;
        this.alias = alias;
    }
    // Printing method:
    public void printEmployee()
    {
        Console.WriteLine("Name: {0}\nAlias: {1}", name, alias);
        // Passing the object to the CalcTax method by using this:
        Console.WriteLine("Taxes: {0:C}", Tax.CalcTax(this));
    }

    public decimal Salary
    {
        get { return salary; }
    }
}

class Tax
{
    public static decimal CalcTax(Employee E)
    {
        return 0.08m * E.Salary;
    }
}

class MainClass
{
    static void Main()
    {
        // Create objects:
        Employee E1 = new Employee("Mingda Pan", "mpan");

        // Display results:
        E1.printEmployee();
    }
}
/*
Output:
    Name: Mingda Pan
    Alias: mpan
    Taxes: $240.00
 */
```

C# Language Specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See Also

- C# Reference
- C# Programming Guide
- C# Keywords

- base
- Methods

Literal Keywords (C# Reference)

8/25/2018 • 2 minutes to read • Edit Online

C# has the following literal keywords:

- null
- true
- false
- default

See Also

- C# Reference
- C# Programming Guide
- C# Keywords

null (C# Reference)

8/19/2018 • 2 minutes to read • Edit Online

The `null` keyword is a literal that represents a null reference, one that does not refer to any object. `null` is the default value of reference-type variables. Ordinary value types cannot be null. However, C# 2.0 introduced nullable value types. See Nullable Types.

The following example demonstrates some behaviors of the null keyword:

```
class Program
{
    class MyClass
    {
        public void MyMethod() { }
    }

    static void Main(string[] args)
    {
        // Set a breakpoint here to see that mc = null.
        // However, the compiler considers it "unassigned."
        // and generates a compiler error if you try to
        // use the variable.
        MyClass mc;

        // Now the variable can be used, but...
        mc = null;

        // ... a method call on a null object raises
        // a run-time NullReferenceException.
        // Uncomment the following line to see for yourself.
        // mc.MyMethod();

        // Now mc has a value.
        mc = new MyClass();

        // You can call its method.
        mc.MyMethod();

        // Set mc to null again. The object it referenced
        // is no longer accessible and can now be garbage-collected.
        mc = null;

        // A null string is not the same as an empty string.
        string s = null;
        string t = String.Empty; // Logically the same as ""

        // Equals applied to any null object returns false.
        bool b = (t.Equals(s));
        Console.WriteLine(b);

        // Equality operator also returns false when one
        // operand is null.
        Console.WriteLine("Empty string {0} null string", s == t ? "equals": "does not equal");

        // Returns true.
        Console.WriteLine("null == null is {0}", null == null);


        // A value type cannot be null
        // int i = null; // Compiler error!

        // Use a nullable value type instead:
        int? i = null;

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();

    }
}
```

C# language specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- C# Reference
- C# Programming Guide
- C# Keywords
- Literal Keywords
- Default Values Table
- Nothing

default (C# Reference)

8/25/2018 • 2 minutes to read • Edit Online

The `default` keyword can be used in the `switch` statement or in a default value expression:

- The switch statement: Specifies the default label.
- Default value expressions: Produces the default value of a type.

See also

- C# Reference
- C# Programming Guide
- C# Keywords

Contextual Keywords (C# Reference)

8/25/2018 • 2 minutes to read • Edit Online

A contextual keyword is used to provide a specific meaning in the code, but it is not a reserved word in C#. The following contextual keywords are introduced in this section:

KEYWORD	DESCRIPTION
add	Defines a custom event accessor that is invoked when client code subscribes to the event.
async	Indicates that the modified method, lambda expression, or anonymous method is asynchronous.
await	Suspends an async method until an awaited task is completed.
dynamic	Defines a reference type that enables operations in which it occurs to bypass compile-time type checking.
get	Defines an accessor method for a property or an indexer.
global	Specifies the default global namespace, which is otherwise unnamed.
partial	Defines partial classes, structs, and interfaces throughout the same compilation unit.
remove	Defines a custom event accessor that is invoked when client code unsubscribes from the event.
set	Defines an accessor method for a property or an indexer.
value	Used to set accessors and to add or remove event handlers.
var	Enables the type of a variable declared at method scope to be determined by the compiler.
when	Specifies a filter condition for a `catch` block or the `case` label of a `switch` statement.
where	Adds constraints to a generic declaration. (See also where).
yield	Used in an iterator block to return a value to the enumerator object or to signal the end of iteration.

All query keywords introduced in C# 3.0 are also contextual. For more information, see Query Keywords (LINQ).

See Also

- C# Reference
- C# Programming Guide

- C# Keywords

add (C# Reference)

8/25/2018 • 2 minutes to read • Edit Online

The `add` contextual keyword is used to define a custom event accessor that is invoked when client code subscribes to your event. If you supply a custom `add` accessor, you must also supply a remove accessor.

Example

The following example shows an event that has custom `add` and remove accessors. For the full example, see How to: Implement Interface Events.

```
class Events : IDrawingObject
{
    event EventHandler PreDrawEvent;

    event EventHandler IDrawingObject.OnDraw
    {
        add
        {
            lock (PreDrawEvent)
            {
                PreDrawEvent += value;
            }
        }
        remove
        {
            lock (PreDrawEvent)
            {
                PreDrawEvent -= value;
            }
        }
    }

}
```

You do not typically need to provide your own custom event accessors. The accessors that are automatically generated by the compiler when you declare an event are sufficient for most scenarios.

See Also

- Events

get (C# Reference)

8/25/2018 • 2 minutes to read • Edit Online

The `get` keyword defines an *accessor* method in a property or indexer that returns the property value or the indexer element. For more information, see Properties, Auto-Implemented Properties and Indexers.

The following example defines both a `get` and a `set` accessor for a property named `Seconds`. It uses a private field named `_seconds` to back the property value.

```
class TimePeriod
{
     private double _seconds;

     public double Seconds
     {
         get { return _seconds; }
         set { _seconds = value; }
     }
}
```

Often, the `get` accessor consists of a single statement that returns a value, as it did in the previous example. Starting with C# 7.0, you can implement the `get` accessor as an expression-bodied member. The following example implements both the `get` and the `set` accessor as expression-bodied members.

```
class TimePeriod
{
    private double _seconds;

    public double Seconds
    {
        get => _seconds;
        set => _seconds = value;
    }
}
```

For simple cases in which a property's `get` and `set` accessors perform no other operation than setting or retrieving a value in a private backing field, you can take advantage of the C# compiler's support for auto-implemented properties. The following example implements `Hours` as an auto-implemented property.

```
class TimePeriod2
{
    public double Hours { get; set; }
}
```

C# Language Specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See Also

- C# Reference
- C# Programming Guide
- C# Keywords
- Properties

global (C# Reference)

8/25/2018 • 2 minutes to read • Edit Online

The `global` contextual keyword, when it comes before the :: operator, refers to the global namespace, which is the default namespace for any C# program and is otherwise unnamed. For more information, see How to: Use the Global Namespace Alias.

Example

The following example shows how to use the `global` contextual keyword to specify that the class `TestApp` is defined in the global namespace:

```
class TestClass : global::TestApp { }
```

See also

- Namespaces

partial type (C# Reference)

8/29/2018 • 2 minutes to read • Edit Online

Partial type definitions allow for the definition of a class, struct, or interface to be split into multiple files.

In *File1.cs*:

```
namespace PC
{
    partial class A
    {
        int num = 0;
        void MethodA() { }
        partial void MethodC();
    }
}
```

In *File2.cs* the declaration:

```
namespace PC
{
    partial class A
    {
        void MethodB() { }
        partial void MethodC() { }
    }
}
```

Remarks

Splitting a class, struct or interface type over several files can be useful when you are working with large projects, or with automatically generated code such as that provided by the Windows Forms Designer. A partial type may contain a partial method. For more information, see Partial Classes and Methods.

C# language specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- C# Reference
- C# Programming Guide
- Modifiers
- Introduction to Generics

partial method (C# Reference)

8/29/2018 • 2 minutes to read • Edit Online

A partial method has its signature defined in one part of a partial type, and its implementation defined in another part of the type. Partial methods enable class designers to provide method hooks, similar to event handlers, that developers may decide to implement or not. If the developer does not supply an implementation, the compiler removes the signature at compile time. The following conditions apply to partial methods:

- Signatures in both parts of the partial type must match.
- The method must return void.
- No access modifiers are allowed. Partial methods are implicitly private.

The following example shows a partial method defined in two parts of a partial class:

```
namespace PM
{
    partial class A
    {
        partial void OnSomethingHappened(string s);
    }

    // This part can be in a separate file.
    partial class A
    {
        // Comment out this method and the program
        // will still compile.
        partial void OnSomethingHappened(String s)
        {
            Console.WriteLine("Something happened: {0}", s);
        }
    }
}
```

For more information, see Partial Classes and Methods.

See also

- C# Reference
- partial type

remove (C# Reference)

8/29/2018 • 2 minutes to read • Edit Online

The `remove` contextual keyword is used to define a custom event accessor that is invoked when client code unsubscribes from your event. If you supply a custom `remove` accessor, you must also supply an add accessor.

Example

The following example shows an event with custom add and `remove` accessors. For the full example, see How to: Implement Interface Events.

```
class Events : IDrawingObject
{
    event EventHandler PreDrawEvent;

    event EventHandler IDrawingObject.OnDraw
    {
        add
        {
            lock (PreDrawEvent)
            {
                PreDrawEvent += value;
            }
        }
        remove
        {
            lock (PreDrawEvent)
            {
                PreDrawEvent -= value;
            }
        }
    }

}
```

You do not typically need to provide your own custom event accessors. The accessors that are automatically generated by the compiler when you declare an event are sufficient for most scenarios.

See also

- Events

set (C# Reference)

8/25/2018 • 2 minutes to read • Edit Online

The `set` keyword defines an *accessor* method in a property or indexer that assigns a value to the property or the indexer element. For more information and examples, see Properties, Auto-Implemented Properties, and Indexers.

The following example defines both a `get` and a `set` accessor for a property named `Seconds`. It uses a private field named `_seconds` to back the property value.

```
class TimePeriod
{
     private double _seconds;

     public double Seconds
     {
         get { return _seconds; }
         set { _seconds = value; }
     }
}
```

Often, the `set` accessor consists of a single statement that returns a value, as it did in the previous example. Starting with C# 7.0, you can implement the `set` accessor as an expression-bodied member. The following example implements both the `get` and the `set` accessors as expression-bodied members.

```
class TimePeriod
{
    private double _seconds;

    public double Seconds
    {
        get => _seconds;
        set => _seconds = value;
    }
}
```

For simple cases in which a property's `get` and `set` accessors perform no other operation than setting or retrieving a value in a private backing field, you can take advantage of the C# compiler's support for auto-implemented properties. The following example implements `Hours` as an auto-implemented property.

```
class TimePeriod2
{
    public double Hours { get; set; }
}
```

C# Language Specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See Also

- C# Reference
- C# Programming Guide
- C# Keywords
- Properties

when (C# Reference)

8/25/2018 • 3 minutes to read • Edit Online

You can use the `when` contextual keyword to specify a filter condition in two contexts:

- In the `catch` statement of a try/catch or try/catch/finally block.
- In the `case` label of a switch statement.

`when` in a `catch` statement

Starting with C# 6, `When` can be used in a `catch` statement to specify a condition that must be true for the handler for a specific exception to execute. Its syntax is:

```
catch ExceptionType [e] when (expr)
```

where *expr* is an expression that evaluates to a Boolean value. If it returns `true`, the exception handler executes; if `false`, it does not.

The following example uses the `when` keyword to conditionally execute handlers for an HttpRequestException depending on the text of the exception message.

```
using System;
using System.Net.Http;
using System.Threading.Tasks;

class Program
{
   static void Main()
   {
      Console.WriteLine(MakeRequest().Result);
   }

   public static async Task<string> MakeRequest()
   {
       var client = new System.Net.Http.HttpClient();
       var streamTask = client.GetStringAsync("https://localHost:10000");
       try {
           var responseText = await streamTask;
           return responseText;
       }
       catch (HttpRequestException e) when (e.Message.Contains("301")) {
           return "Site Moved";
       }
       catch (HttpRequestException e) when (e.Message.Contains("404")) {
           return "Page Not Found";
       }
       catch (HttpRequestException e) {
           return e.Message;
       }
   }
}
```

`when` in a `switch` statement

Starting with C# 7.0, `case` labels no longer need be mutually exclusive, and the order in which `case` labels appear in a `switch` statement can determine which switch block executes. The `when` keyword can be used to specify a filter condition that causes its associated case label to be true only if the filter condition is also true. Its syntax is:

```
case (expr) when (when-condition):
```

where *expr* is a constant pattern or type pattern that is compared to the match expression, and *when-condition* is any Boolean expression.

The following example uses the `when` keyword to test for `Shape` objects that have an area of zero, as well as to test for a variety of `Shape` objects that have an area greater than zero.

```
using System;

public abstract class Shape
{
   public abstract double Area { get; }
   public abstract double Circumference { get; }
}

public class Rectangle : Shape
{
   public Rectangle(double length, double width)
   {
      Length = length;
      Width = width;
   }

   public double Length { get; set; }
   public double Width { get; set; }

   public override double Area
   {
      get { return Math.Round(Length * Width,2); }
   }

   public override double Circumference
   {
      get { return (Length + Width) * 2; }
   }
}

public class Square : Rectangle
{
   public Square(double side) : base(side, side)
   {
      Side = side;
   }

   public double Side { get; set; }
}

public class Example
{
   public static void Main()
   {
      Shape sh = null;
      Shape[] shapes = { new Square(10), new Rectangle(5, 7),
                         new Rectangle(10, 10), sh, new Square(0) };
      foreach (var shape in shapes)
         ShowShapeInfo(shape);
   }

   private static void ShowShapeInfo(Object obj)
   {
```

```
   {
      switch (obj)
      {
         case Shape shape when shape.Area == 0:
            Console.WriteLine($"The shape: {shape.GetType().Name} with no dimensions");
            break;
         case Square sq when sq.Area > 0:
            Console.WriteLine("Information about the square:");
            Console.WriteLine($"   Length of a side: {sq.Side}");
            Console.WriteLine($"   Area: {sq.Area}");
            break;
         case Rectangle r when r.Area > 0:
            Console.WriteLine("Information about the rectangle:");
            Console.WriteLine($"   Dimensions: {r.Length} x {r.Width}");
            Console.WriteLine($"   Area: {r.Area}");
            break;
         case Shape shape:
            Console.WriteLine($"A {shape.GetType().Name} shape");
            break;
         case null:
            Console.WriteLine($"The {nameof(obj)} variable is uninitialized.");
            break;
         default:
            Console.WriteLine($"The {nameof(obj)} variable does not represent a Shape.");
            break;
      }
   }
}
// The example displays the following output:
//       Information about the square:
//          Length of a side: 10
//          Area: 100
//       Information about the rectangle:
//          Dimensions: 5 x 7
//          Area: 35
//       Information about the rectangle:
//          Dimensions: 10 x 10
//          Area: 100
//       The obj variable is uninitialized.
//       The shape: Square with no dimensions
```

See also

- switch statement
- try/catch statement
- try/catch/finally statement

where (generic type constraint) (C# Reference)

8/25/2018 • 3 minutes to read • Edit Online

The `where` clause in a generic definition specifies constraints on the types that are used as arguments for type parameters in a generic type, method, delegate, or local function. Constraints can specify interfaces, base classes, or require a generic type to be a reference, value or unmanaged type. They declare capabilities that the type argument must possess.

For example, you can declare a generic class, `MyGenericClass`, such that the type parameter `T` implements the IComparable<T> interface:

```
public class AGenericClass<T> where T : IComparable<T> { }
```

NOTE

For more information on the where clause in a query expression, see where clause.

The `where` clause can also include a base class constraint. The base class constraint states that a type to be used as a type argument for that generic type has the specified class as a base class (or is that base class) to be used as a type argument for that generic type. If the base class constraint is used, it must appear before any other constraints on that type parameter. Some types are disallowed as a base class constraint: Object, Array, and ValueType. Prior to C# 7.3, Enum, Delegate, and MulticastDelegate were also disallowed as base class constraints. The following example shows the types that can now be specified as a base class:

```
public class UsingEnum<T> where T : System.Enum { }

public class UsingDelegate<T> where T : System.Delegate { }

public class Multicaster<T> where T : System.MulticastDelegate { }
```

The `where` clause can specify that the type is a `class` or a `struct`. The `struct` constraint removes the need to specify a base class constraint of `System.ValueType`. The `System.ValueType` type may not be used as a base class constraint. The following example shows both the `class` and `struct` constraints:

```
class MyClass<T, U>
    where T : class
    where U : struct
{ }
```

The `where` clause may also include an `unmanaged` constraint. The `unmanaged` constraint limits the type parameter to types known as **unmanaged types**. An **unmanaged type** is a type that isn't a reference type and doesn't contain reference type fields at any level of nesting. The `unmanaged` constraint makes it easier to write low-level interop code in C#. This constraint enables reusable routines across all unmanaged types. The `unmanaged` constraint can't be combined with the `class` or `struct` constraint. The `unmanaged` constraint enforces that the type must be a `struct`:

```
class UnManagedWrapper<T>
    where T : unmanaged
{ }
```

The `where` clause may also include a constructor constraint, `new()`. That constraint makes it possible to create an instance of a type parameter using the `new` operator. The new() Constraint lets the compiler know that any type argument supplied must have an accessible parameterless--or default-- constructor. For example:

```
public class MyGenericClass<T> where T : IComparable<T>, new()
{
    // The following line is not possible without new() constraint:
    T item = new T();
}
```

The `new()` constraint appears last in the `where` clause. The `new()` constraint can't be combined with the `struct` or `unmanaged` constraints. All types satisfying those constraints must have an accessible parameterless constructor, making the `new()` constraint redundant.

With multiple type parameters, use one `where` clause for each type parameter, for example:

```
public interface IMyInterface { }

namespace CodeExample
{
    class Dictionary<TKey, TVal>
        where TKey : IComparable<TKey>
        where TVal : IMyInterface
    {
        public void Add(TKey key, TVal val) { }
    }
}
```

You can also attach constraints to type parameters of generic methods, as shown in the following example:

```
public void MyMethod<T>(T t) where T : IMyInterface { }
```

Notice that the syntax to describe type parameter constraints on delegates is the same as that of methods:

```
delegate T MyDelegate<T>() where T : new();
```

For information on generic delegates, see Generic Delegates.

For details on the syntax and use of constraints, see Constraints on Type Parameters.

C# language specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- C# Reference
- C# Programming Guide
- Introduction to Generics

- new Constraint
- Constraints on Type Parameters

value (C# Reference)

8/25/2018 • 2 minutes to read • Edit Online

The contextual keyword `value` is used in the set accessor in ordinary property declarations. It is similar to an input parameter on a method. The word `value` references the value that client code is attempting to assign to the property. In the following example, `MyDerivedClass` has a property called `Name` that uses the `value` parameter to assign a new string to the backing field `name`. From the point of view of client code, the operation is written as a simple assignment.

```
class MyBaseClass
{
    // virtual auto-implemented property. Overrides can only
    // provide specialized behavior if they implement get and set accessors.
    public virtual string Name { get; set; }

    // ordinary virtual property with backing field
    private int num;
    public virtual int Number
    {
        get { return num; }
        set { num = value; }
    }
}


class MyDerivedClass : MyBaseClass
{
    private string name;

   // Override auto-implemented property with ordinary property
   // to provide specialized accessor behavior.
    public override string Name
    {
        get
        {
            return name;
        }
        set
        {
            if (value != String.Empty)
            {
                name = value;
            }
            else
            {
                name = "Unknown";
            }
        }
    }

}
```

For more information about the use of `value`, see Properties.

C# Language Specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See Also

- C# Reference
- C# Programming Guide
- C# Keywords

yield (C# Reference)

8/25/2018 • 4 minutes to read • Edit Online

When you use the `yield` keyword in a statement, you indicate that the method, operator, or `get` accessor in which it appears is an iterator. Using `yield` to define an iterator removes the need for an explicit extra class (the class that holds the state for an enumeration, see IEnumerator<T> for an example) when you implement the IEnumerable and IEnumerator pattern for a custom collection type.

The following example shows the two forms of the `yield` statement.

```
yield return <expression>;
yield break;
```

Remarks

You use a `yield return` statement to return each element one at a time.

You consume an iterator method by using a foreach statement or LINQ query. Each iteration of the `foreach` loop calls the iterator method. When a `yield return` statement is reached in the iterator method, `expression` is returned, and the current location in code is retained. Execution is restarted from that location the next time that the iterator function is called.

You can use a `yield break` statement to end the iteration.

For more information about iterators, see Iterators.

Iterator Methods and get Accessors

The declaration of an iterator must meet the following requirements:

- The return type must be IEnumerable, IEnumerable<T>, IEnumerator, or IEnumerator<T>.
- The declaration can't have any in ref or out parameters.

The `yield` type of an iterator that returns IEnumerable or IEnumerator is `object`. If the iterator returns IEnumerable<T> or IEnumerator<T>, there must be an implicit conversion from the type of the expression in the `yield return` statement to the generic type parameter .

You can't include a `yield return` or `yield break` statement in methods that have the following characteristics:

- Anonymous methods. For more information, see Anonymous Methods.
- Methods that contain unsafe blocks. For more information, see unsafe.

Exception Handling

A `yield return` statement can't be located in a try-catch block. A `yield return` statement can be located in the try block of a try-finally statement.

A `yield break` statement can be located in a try block or a catch block but not a finally block.

If the `foreach` body (outside of the iterator method) throws an exception, a `finally` block in the iterator method is executed.

Technical Implementation

The following code returns an `IEnumerable<string>` from an iterator method and then iterates through its elements.

```
IEnumerable<string> elements = MyIteratorMethod();
foreach (string element in elements)
{
   ...
}
```

The call to `MyIteratorMethod` doesn't execute the body of the method. Instead the call returns an `IEnumerable<string>` into the `elements` variable.

On an iteration of the `foreach` loop, the MoveNext method is called for `elements`. This call executes the body of `MyIteratorMethod` until the next `yield return` statement is reached. The expression returned by the `yield return` statement determines not only the value of the `element` variable for consumption by the loop body but also the Current property of `elements`, which is an `IEnumerable<string>`.

On each subsequent iteration of the `foreach` loop, the execution of the iterator body continues from where it left off, again stopping when it reaches a `yield return` statement. The `foreach` loop completes when the end of the iterator method or a `yield break` statement is reached.

Example

The following example has a `yield return` statement that's inside a `for` loop. Each iteration of the `foreach` statement body in the `Main` method creates a call to the `Power` iterator function. Each call to the iterator function proceeds to the next execution of the `yield return` statement, which occurs during the next iteration of the `for` loop.

The return type of the iterator method is IEnumerable, which is an iterator interface type. When the iterator method is called, it returns an enumerable object that contains the powers of a number.

```
public class PowersOf2
{
    static void Main()
    {
        // Display powers of 2 up to the exponent of 8:
        foreach (int i in Power(2, 8))
        {
            Console.Write("{0} ", i);
        }
    }

    public static System.Collections.Generic.IEnumerable<int> Power(int number, int exponent)
    {
        int result = 1;

        for (int i = 0; i < exponent; i++)
        {
            result = result * number;
            yield return result;
        }
    }

    // Output: 2 4 8 16 32 64 128 256
}
```

Example

The following example demonstrates a `get` accessor that is an iterator. In the example, each `yield return` statement returns an instance of a user-defined class.

```
public static class GalaxyClass
{
    public static void ShowGalaxies()
    {
        var theGalaxies = new Galaxies();
        foreach (Galaxy theGalaxy in theGalaxies.NextGalaxy)
        {
            Debug.WriteLine(theGalaxy.Name + " " + theGalaxy.MegaLightYears.ToString());
        }
    }

    public class Galaxies
    {

        public System.Collections.Generic.IEnumerable<Galaxy> NextGalaxy
        {
            get
            {
                yield return new Galaxy { Name = "Tadpole", MegaLightYears = 400 };
                yield return new Galaxy { Name = "Pinwheel", MegaLightYears = 25 };
                yield return new Galaxy { Name = "Milky Way", MegaLightYears = 0 };
                yield return new Galaxy { Name = "Andromeda", MegaLightYears = 3 };
            }
        }

    }

    public class Galaxy
    {
        public String Name { get; set; }
        public int MegaLightYears { get; set; }
    }
}
```

C# Language Specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See Also

- C# Reference
- C# Programming Guide
- foreach, in
- Iterators

Query keywords (C# Reference)

8/29/2018 • 2 minutes to read • Edit Online

This section contains the contextual keywords used in query expressions.

In this section

CLAUSE	DESCRIPTION
from	Specifies a data source and a range variable (similar to an iteration variable).
where	Filters source elements based on one or more Boolean expressions separated by logical AND and OR operators (`&&` or `||`).
select	Specifies the type and shape that the elements in the returned sequence will have when the query is executed.
group	Groups query results according to a specified key value.
into	Provides an identifier that can serve as a reference to the results of a join, group or select clause.
orderby	Sorts query results in ascending or descending order based on the default comparer for the element type.
join	Joins two data sources based on an equality comparison between two specified matching criteria.
let	Introduces a range variable to store sub-expression results in a query expression.
in	Contextual keyword in a join clause.
on	Contextual keyword in a join clause.
equals	Contextual keyword in a join clause.
by	Contextual keyword in a group clause.
ascending	Contextual keyword in an orderby clause.
descending	Contextual keyword in an orderby clause.

See also

- C# Keywords
- LINQ (Language-Integrated Query)
- LINQ Query Expressions

- Getting Started with LINQ in C#

from clause (C# Reference)

8/25/2018 • 5 minutes to read • Edit Online

A query expression must begin with a `from` clause. Additionally, a query expression can contain sub-queries, which also begin with a `from` clause. The `from` clause specifies the following:

- The data source on which the query or sub-query will be run.
- A local *range variable* that represents each element in the source sequence.

Both the range variable and the data source are strongly typed. The data source referenced in the `from` clause must have a type of IEnumerable, IEnumerable<T>, or a derived type such as IQueryable<T>.

In the following example, `numbers` is the data source and `num` is the range variable. Note that both variables are strongly typed even though the var keyword is used.

```
class LowNums
{
    static void Main()
    {
        // A simple data source.
        int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };

        // Create the query.
        // lowNums is an IEnumerable<int>
        var lowNums = from num in numbers
            where num < 5
            select num;

        // Execute the query.
        foreach (int i in lowNums)
        {
            Console.Write(i + " ");
        }
    }
}
// Output: 4 1 3 2 0
```

The range variable

The compiler infers the type of the range variable when the data source implements IEnumerable<T>. For example, if the source has a type of `IEnumerable<Customer>`, then the range variable is inferred to be `Customer`. The only time that you must specify the type explicitly is when the source is a non-generic `IEnumerable` type such as ArrayList. For more information, see How to: Query an ArrayList with LINQ.

In the previous example `num` is inferred to be of type `int`. Because the range variable is strongly typed, you can call methods on it or use it in other operations. For example, instead of writing `select num`, you could write `select num.ToString()` to cause the query expression to return a sequence of strings instead of integers. Or you could write `select n + 10` to cause the expression to return the sequence 14, 11, 13, 12, 10. For more information, see select clause.

The range variable is like an iteration variable in a foreach statement except for one very important difference: a range variable never actually stores data from the source. It just a syntactic convenience that enables the query to describe what will occur when the query is executed. For more information, see Introduction to LINQ Queries (C#).

Compound from clauses

In some cases, each element in the source sequence may itself be either a sequence or contain a sequence. For example, your data source may be an `IEnumerable<Student>` where each student object in the sequence contains a list of test scores. To access the inner list within each `Student` element, you can use compound `from` clauses. The technique is like using nested foreach statements. You can add where or orderby clauses to either `from` clause to filter the results. The following example shows a sequence of `Student` objects, each of which contains an inner `List` of integers representing test scores. To access the inner list, use a compound `from` clause. You can insert clauses between the two `from` clauses if necessary.

```
class CompoundFrom
{
    // The element type of the data source.
    public class Student
    {
        public string LastName { get; set; }
        public List<int> Scores {get; set;}
    }

    static void Main()
    {

        // Use a collection initializer to create the data source. Note that
        // each element in the list contains an inner sequence of scores.
        List<Student> students = new List<Student>
        {
           new Student {LastName="Omelchenko", Scores= new List<int> {97, 72, 81, 60}},
           new Student {LastName="O'Donnell", Scores= new List<int> {75, 84, 91, 39}},
           new Student {LastName="Mortensen", Scores= new List<int> {88, 94, 65, 85}},
           new Student {LastName="Garcia", Scores= new List<int> {97, 89, 85, 82}},
           new Student {LastName="Beebe", Scores= new List<int> {35, 72, 91, 70}}
        };

        // Use a compound from to access the inner sequence within each element.
        // Note the similarity to a nested foreach statement.
        var scoreQuery = from student in students
                         from score in student.Scores
                            where score > 90
                            select new { Last = student.LastName, score };

        // Execute the queries.
        Console.WriteLine("scoreQuery:");
        // Rest the mouse pointer on scoreQuery in the following line to
        // see its type. The type is IEnumerable<'a>, where 'a is an
        // anonymous type defined as new {string Last, int score}. That is,
        // each instance of this anonymous type has two members, a string
        // (Last) and an int (score).
        foreach (var student in scoreQuery)
        {
            Console.WriteLine("{0} Score: {1}", student.Last, student.score);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/*
scoreQuery:
Omelchenko Score: 97
O'Donnell Score: 91
Mortensen Score: 94
Garcia Score: 97
Beebe Score: 91
*/
```

Using Multiple from Clauses to Perform Joins

A compound `from` clause is used to access inner collections in a single data source. However, a query can also contain multiple `from` clauses that generate supplemental queries from independent data sources. This technique enables you to perform certain types of join operations that are not possible by using the join clause.

The following example shows how two `from` clauses can be used to form a complete cross join of two data sources.

```
class CompoundFrom2
{
    static void Main()
    {
        char[] upperCase = { 'A', 'B', 'C' };
        char[] lowerCase = { 'x', 'y', 'z' };

        // The type of joinQuery1 is IEnumerable<'a>, where 'a
        // indicates an anonymous type. This anonymous type has two
        // members, upper and lower, both of type char.
        var joinQuery1 =
            from upper in upperCase
            from lower in lowerCase
            select new { upper, lower };

        // The type of joinQuery2 is IEnumerable<'a>, where 'a
        // indicates an anonymous type. This anonymous type has two
        // members, upper and lower, both of type char.
        var joinQuery2 =
            from lower in lowerCase
            where lower != 'x'
            from upper in upperCase
            select new { lower, upper };


        // Execute the queries.
        Console.WriteLine("Cross join:");
        // Rest the mouse pointer on joinQuery1 to verify its type.
        foreach (var pair in joinQuery1)
        {
            Console.WriteLine("{0} is matched to {1}", pair.upper, pair.lower);
        }

        Console.WriteLine("Filtered non-equijoin:");
        // Rest the mouse pointer over joinQuery2 to verify its type.
        foreach (var pair in joinQuery2)
        {
            Console.WriteLine("{0} is matched to {1}", pair.lower, pair.upper);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
        Cross join:
        A is matched to x
        A is matched to y
        A is matched to z
        B is matched to x
        B is matched to y
        B is matched to z
        C is matched to x
        C is matched to y
        C is matched to z
        Filtered non-equijoin:
        y is matched to A
        y is matched to B
        y is matched to C
        z is matched to A
        z is matched to B
        z is matched to C
        */
```

For more information about join operations that use multiple `from` clauses, see Perform left outer joins.

See also

- Query Keywords (LINQ)
- Language Integrated Query (LINQ)

where clause (C# Reference)

8/25/2018 • 3 minutes to read • Edit Online

The `where` clause is used in a query expression to specify which elements from the data source will be returned in the query expression. It applies a Boolean condition (*predicate*) to each source element (referenced by the range variable) and returns those for which the specified condition is true. A single query expression may contain multiple `where` clauses and a single clause may contain multiple predicate subexpressions.

Example

In the following example, the `where` clause filters out all numbers except those that are less than five. If you remove the `where` clause, all numbers from the data source would be returned. The expression `num < 5` is the predicate that is applied to each element.

```
class WhereSample
{
    static void Main()
    {
        // Simple data source. Arrays support IEnumerable<T>.
        int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };

        // Simple query with one predicate in where clause.
        var queryLowNums =
            from num in numbers
            where num < 5
            select num;

        // Execute the query.
        foreach (var s in queryLowNums)
        {
            Console.Write(s.ToString() + " ");
        }
    }
}
//Output: 4 1 3 2 0
```

Example

Within a single `where` clause, you can specify as many predicates as necessary by using the && and || operators. In the following example, the query specifies two predicates in order to select only the even numbers that are less than five.

```
class WhereSample2
{
    static void Main()
    {
        // Data source.
        int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };

        // Create the query with two predicates in where clause.
        var queryLowNums2 =
            from num in numbers
            where num < 5 && num % 2 == 0
            select num;

        // Execute the query
        foreach (var s in queryLowNums2)
        {
            Console.Write(s.ToString() + " ");
        }
        Console.WriteLine();

        // Create the query with two where clause.
        var queryLowNums3 =
            from num in numbers
            where num < 5
            where num % 2 == 0
            select num;

        // Execute the query
        foreach (var s in queryLowNums3)
        {
            Console.Write(s.ToString() + " ");
        }

    }
}
// Output:
// 4 2 0
// 4 2 0
```

Example

A `where` clause may contain one or more methods that return Boolean values. In the following example, the `where` clause uses a method to determine whether the current value of the range variable is even or odd.

```
class WhereSample3
{
    static void Main()
    {
        // Data source
        int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };

        // Create the query with a method call in the where clause.
        // Note: This won't work in LINQ to SQL unless you have a
        // stored procedure that is mapped to a method by this name.
        var queryEvenNums =
            from num in numbers
            where IsEven(num)
            select num;

         // Execute the query.
        foreach (var s in queryEvenNums)
        {
            Console.Write(s.ToString() + " ");
        }
    }

    // Method may be instance method or static method.
    static bool IsEven(int i)
    {
        return i % 2 == 0;
    }
}
//Output: 4 8 6 2 0
```

Remarks

The `where` clause is a filtering mechanism. It can be positioned almost anywhere in a query expression, except it cannot be the first or last clause. A `where` clause may appear either before or after a group clause depending on whether you have to filter the source elements before or after they are grouped.

If a specified predicate is not valid for the elements in the data source, a compile-time error will result. This is one benefit of the strong type-checking provided by LINQ.

At compile time the `where` keyword is converted into a call to the Where Standard Query Operator method.

See Also

- Query Keywords (LINQ)
- from clause
- select clause
- Filtering Data
- LINQ Query Expressions
- Getting Started with LINQ in C#

select clause (C# Reference)

8/25/2018 • 6 minutes to read • Edit Online

In a query expression, the `select` clause specifies the type of values that will be produced when the query is executed. The result is based on the evaluation of all the previous clauses and on any expressions in the `select` clause itself. A query expression must terminate with either a `select` clause or a group clause.

The following example shows a simple `select` clause in a query expression.

```
class SelectSample1
{
    static void Main()
    {
        //Create the data source
        List<int> Scores = new List<int>() { 97, 92, 81, 60 };

        // Create the query.
        IEnumerable<int> queryHighScores =
            from score in Scores
            where score > 80
            select score;

        // Execute the query.
        foreach (int i in queryHighScores)
        {
            Console.Write(i + " ");
        }
    }
}
//Output: 97 92 81
```

The type of the sequence produced by the `select` clause determines the type of the query variable `queryHighScores`. In the simplest case, the `select` clause just specifies the range variable. This causes the returned sequence to contain elements of the same type as the data source. For more information, see Type Relationships in LINQ Query Operations. However, the `select` clause also provides a powerful mechanism for transforming (or *projecting*) source data into new types. For more information, see Data Transformations with LINQ (C#).

Example

The following example shows all the different forms that a `select` clause may take. In each query, note the relationship between the `select` clause and the type of the *query variable* (`studentQuery1`, `studentQuery2`, and so on).

```
    class SelectSample2
    {
        // Define some classes
        public class Student
        {
            public string First { get; set; }
            public string Last { get; set; }
            public int ID { get; set; }
            public List<int> Scores;
            public ContactInfo GetContactInfo(SelectSample2 app, int id)
            {
                ContactInfo cInfo =
```

```
                (from ci in app.contactList
                 where ci.ID == id
                 select ci)
                 .FirstOrDefault();

            return cInfo;
        }

        public override string ToString()
        {
            return First + " " + Last + ":" + ID;
        }
    }

    public class ContactInfo
    {
        public int ID { get; set; }
        public string Email { get; set; }
        public string Phone { get; set; }
        public override string ToString() { return Email + "," + Phone; }
    }

    public class ScoreInfo
    {
        public double Average { get; set; }
        public int ID { get; set; }
    }

    // The primary data source
    List<Student> students = new List<Student>()
    {
         new Student {First="Svetlana", Last="Omelchenko", ID=111, Scores= new List<int>() {97, 92, 81,
60}},
         new Student {First="Claire", Last="O'Donnell", ID=112, Scores= new List<int>() {75, 84, 91, 39}},
         new Student {First="Sven", Last="Mortensen", ID=113, Scores= new List<int>() {88, 94, 65, 91}},
         new Student {First="Cesar", Last="Garcia", ID=114, Scores= new List<int>() {97, 89, 85, 82}},
    };

    // Separate data source for contact info.
    List<ContactInfo> contactList = new List<ContactInfo>()
    {
        new ContactInfo {ID=111, Email="SvetlanO@Contoso.com", Phone="206-555-0108"},
        new ContactInfo {ID=112, Email="ClaireO@Contoso.com", Phone="206-555-0298"},
        new ContactInfo {ID=113, Email="SvenMort@Contoso.com", Phone="206-555-1130"},
        new ContactInfo {ID=114, Email="CesarGar@Contoso.com", Phone="206-555-0521"}
    };


    static void Main(string[] args)
    {
        SelectSample2 app = new SelectSample2();

        // Produce a filtered sequence of unmodified Students.
        IEnumerable<Student> studentQuery1 =
            from student in app.students
            where student.ID > 111
            select student;

        Console.WriteLine("Query1: select range_variable");
        foreach (Student s in studentQuery1)
        {
            Console.WriteLine(s.ToString());
        }

        // Produce a filtered sequence of elements that contain
        // only one property of each Student.
        IEnumerable<String> studentQuery2 =
            from student in app.students
            where student.ID > 111
```

```
        select student.Last;

    Console.WriteLine("\r\n studentQuery2: select range_variable.Property");
    foreach (string s in studentQuery2)
    {
        Console.WriteLine(s);
    }

    // Produce a filtered sequence of objects created by
    // a method call on each Student.
    IEnumerable<ContactInfo> studentQuery3 =
        from student in app.students
        where student.ID > 111
        select student.GetContactInfo(app, student.ID);

    Console.WriteLine("\r\n studentQuery3: select range_variable.Method");
    foreach (ContactInfo ci in studentQuery3)
    {
        Console.WriteLine(ci.ToString());
    }

    // Produce a filtered sequence of ints from
    // the internal array inside each Student.
    IEnumerable<int> studentQuery4 =
        from student in app.students
        where student.ID > 111
        select student.Scores[0];

    Console.WriteLine("\r\n studentQuery4: select range_variable[index]");
    foreach (int i in studentQuery4)
    {
        Console.WriteLine("First score = {0}", i);
    }

    // Produce a filtered sequence of doubles
    // that are the result of an expression.
    IEnumerable<double> studentQuery5 =
        from student in app.students
        where student.ID > 111
        select student.Scores[0] * 1.1;

    Console.WriteLine("\r\n studentQuery5: select expression");
    foreach (double d in studentQuery5)
    {
        Console.WriteLine("Adjusted first score = {0}", d);
    }

    // Produce a filtered sequence of doubles that are
    // the result of a method call.
    IEnumerable<double> studentQuery6 =
        from student in app.students
        where student.ID > 111
        select student.Scores.Average();

    Console.WriteLine("\r\n studentQuery6: select expression2");
    foreach (double d in studentQuery6)
    {
        Console.WriteLine("Average = {0}", d);
    }

    // Produce a filtered sequence of anonymous types
    // that contain only two properties from each Student.
    var studentQuery7 =
        from student in app.students
        where student.ID > 111
        select new { student.First, student.Last };

    Console.WriteLine("\r\n studentQuery7: select new anonymous type");
    foreach (var item in studentQuery7)
```

```
            foreach (var item in studentQuery7)
            {
                Console.WriteLine("{0}, {1}", item.Last, item.First);
            }

            // Produce a filtered sequence of named objects that contain
            // a method return value and a property from each Student.
            // Use named types if you need to pass the query variable
            // across a method boundary.
            IEnumerable<ScoreInfo> studentQuery8 =
                from student in app.students
                where student.ID > 111
                select new ScoreInfo
                {
                    Average = student.Scores.Average(),
                    ID = student.ID
                };

            Console.WriteLine("\r\n studentQuery8: select new named type");
            foreach (ScoreInfo si in studentQuery8)
            {
                Console.WriteLine("ID = {0}, Average = {1}", si.ID, si.Average);
            }

            // Produce a filtered sequence of students who appear on a contact list
            // and whose average is greater than 85.
            IEnumerable<ContactInfo> studentQuery9 =
                from student in app.students
                where student.Scores.Average() > 85
                join ci in app.contactList on student.ID equals ci.ID
                select ci;

            Console.WriteLine("\r\n studentQuery9: select result of join clause");
            foreach (ContactInfo ci in studentQuery9)
            {
                Console.WriteLine("ID = {0}, Email = {1}", ci.ID, ci.Email);
            }

            // Keep the console window open in debug mode
            Console.WriteLine("Press any key to exit.");
            Console.ReadKey();
            }
        }
    /* Output
        Query1: select range_variable
        Claire O'Donnell:112
        Sven Mortensen:113
        Cesar Garcia:114

        studentQuery2: select range_variable.Property
        O'Donnell
        Mortensen
        Garcia

        studentQuery3: select range_variable.Method
        ClaireO@Contoso.com,206-555-0298
        SvenMort@Contoso.com,206-555-1130
        CesarGar@Contoso.com,206-555-0521

        studentQuery4: select range_variable[index]
        First score = 75
        First score = 88
        First score = 97

        studentQuery5: select expression
        Adjusted first score = 82.5
        Adjusted first score = 96.8
        Adjusted first score = 106.7

        studentQuery6: select expression2
```

```
        studentQuery6: select expression2
        Average = 72.25
        Average = 84.5
        Average = 88.25

        studentQuery7: select new anonymous type
        O'Donnell, Claire
        Mortensen, Sven
        Garcia, Cesar

        studentQuery8: select new named type
        ID = 112, Average = 72.25
        ID = 113, Average = 84.5
        ID = 114, Average = 88.25

        studentQuery9: select result of join clause
        ID = 114, Email = CesarGar@Contoso.com
*/
```

As shown in `studentQuery8` in the previous example, sometimes you might want the elements of the returned sequence to contain only a subset of the properties of the source elements. By keeping the returned sequence as small as possible you can reduce the memory requirements and increase the speed of the execution of the query. You can accomplish this by creating an anonymous type in the `select` clause and using an object initializer to initialize it with the appropriate properties from the source element. For an example of how to do this, see Object and Collection Initializers.

Remarks

At compile time, the `select` clause is translated to a method call to the Select standard query operator.

See Also

- C# Reference
- Query Keywords (LINQ)
- from clause
- partial (Method) (C# Reference)
- Anonymous Types
- LINQ Query Expressions
- Getting Started with LINQ in C#

group clause (C# Reference)

8/25/2018 • 8 minutes to read • Edit Online

The `group` clause returns a sequence of IGrouping<TKey,TElement> objects that contain zero or more items that match the key value for the group. For example, you can group a sequence of strings according to the first letter in each string. In this case, the first letter is the key and has a type char, and is stored in the `Key` property of each IGrouping<TKey,TElement> object. The compiler infers the type of the key.

You can end a query expression with a `group` clause, as shown in the following example:

```
// Query variable is an IEnumerable<IGrouping<char, Student>>
var studentQuery1 =
    from student in students
    group student by student.Last[0];
```

If you want to perform additional query operations on each group, you can specify a temporary identifier by using the into contextual keyword. When you use `into`, you must continue with the query, and eventually end it with either a `select` statement or another `group` clause, as shown in the following excerpt:

```
// Group students by the first letter of their last name
// Query variable is an IEnumerable<IGrouping<char, Student>>
var studentQuery2 =
    from student in students
    group student by student.Last[0] into g
    orderby g.Key
    select g;
```

More complete examples of the use of `group` with and without `into` are provided in the Example section of this article.

Enumerating the results of a group query

Because the IGrouping<TKey,TElement> objects produced by a `group` query are essentially a list of lists, you must use a nested foreach loop to access the items in each group. The outer loop iterates over the group keys, and the inner loop iterates over each item in the group itself. A group may have a key but no elements. The following is the `foreach` loop that executes the query in the previous code examples:

```
// Iterate group items with a nested foreach. This IGrouping encapsulates
// a sequence of Student objects, and a Key of type char.
// For convenience, var can also be used in the foreach statement.
foreach (IGrouping<char, Student> studentGroup in studentQuery2)
{
     Console.WriteLine(studentGroup.Key);
     // Explicit type for student could also be used here.
     foreach (var student in studentGroup)
     {
         Console.WriteLine("   {0}, {1}", student.Last, student.First);
     }
 }
```

Key types

Group keys can be any type, such as a string, a built-in numeric type, or a user-defined named type or anonymous type.

Grouping by string

The previous code examples used a `char`. A string key could easily have been specified instead, for example the complete last name:

```
// Same as previous example except we use the entire last name as a key.
// Query variable is an IEnumerable<IGrouping<string, Student>>
 var studentQuery3 =
     from student in students
     group student by student.Last;
```

Grouping by bool

The following example shows the use of a bool value for a key to divide the results into two groups. Note that the value is produced by a sub-expression in the `group` clause.

```
class GroupSample1
{
    // The element type of the data source.
    public class Student
    {
        public string First { get; set; }
        public string Last { get; set; }
        public int ID { get; set; }
        public List<int> Scores;
    }

    public static List<Student> GetStudents()
    {
        // Use a collection initializer to create the data source. Note that each element
        //  in the list contains an inner sequence of scores.
        List<Student> students = new List<Student>
        {
           new Student {First="Svetlana", Last="Omelchenko", ID=111, Scores= new List<int> {97, 72, 81, 60}},
           new Student {First="Claire", Last="O'Donnell", ID=112, Scores= new List<int> {75, 84, 91, 39}},
           new Student {First="Sven", Last="Mortensen", ID=113, Scores= new List<int> {99, 89, 91, 95}},
           new Student {First="Cesar", Last="Garcia", ID=114, Scores= new List<int> {72, 81, 65, 84}},
           new Student {First="Debra", Last="Garcia", ID=115, Scores= new List<int> {97, 89, 85, 82}}
        };

        return students;

    }

    static void Main()
    {
        // Obtain the data source.
        List<Student> students = GetStudents();

        // Group by true or false.
        // Query variable is an IEnumerable<IGrouping<bool, Student>>
        var booleanGroupQuery =
            from student in students
            group student by student.Scores.Average() >= 80; //pass or fail!

        // Execute the query and access items in each group
        foreach (var studentGroup in booleanGroupQuery)
        {
            Console.WriteLine(studentGroup.Key == true ? "High averages" : "Low averages");
            foreach (var student in studentGroup)
            {
                Console.WriteLine("   {0}, {1}:{2}", student.Last, student.First, student.Scores.Average());
            }
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
  Low averages
   Omelchenko, Svetlana:77.5
   O'Donnell, Claire:72.25
   Garcia, Cesar:75.5
  High averages
   Mortensen, Sven:93.5
   Garcia, Debra:88.25
*/
```

Grouping by numeric range

The next example uses an expression to create numeric group keys that represent a percentile range. Note the use of let as a convenient location to store a method call result, so that you don't have to call the method two times in the `group` clause. Note also in the `group` clause that to avoid a "divide by zero" exception the code checks to make sure that the student doesn't have an average of zero. For more information about how to safely use methods in query expressions, see How to: Handle Exceptions in Query Expressions.

```
class GroupSample2
{
    // The element type of the data source.
    public class Student
    {
        public string First { get; set; }
        public string Last { get; set; }
        public int ID { get; set; }
        public List<int> Scores;
    }

    public static List<Student> GetStudents()
    {
        // Use a collection initializer to create the data source. Note that each element
        //  in the list contains an inner sequence of scores.
        List<Student> students = new List<Student>
        {
           new Student {First="Svetlana", Last="Omelchenko", ID=111, Scores= new List<int> {97, 72, 81, 60}},
           new Student {First="Claire", Last="O'Donnell", ID=112, Scores= new List<int> {75, 84, 91, 39}},
           new Student {First="Sven", Last="Mortensen", ID=113, Scores= new List<int> {99, 89, 91, 95}},
           new Student {First="Cesar", Last="Garcia", ID=114, Scores= new List<int> {72, 81, 65, 84}},
           new Student {First="Debra", Last="Garcia", ID=115, Scores= new List<int> {97, 89, 85, 82}}
        };

        return students;

    }

    // This method groups students into percentile ranges based on their
    // grade average. The Average method returns a double, so to produce a whole
    // number it is necessary to cast to int before dividing by 10.
    static void Main()
    {
        // Obtain the data source.
        List<Student> students = GetStudents();

        // Write the query.
        var studentQuery =
            from student in students
            let avg = (int)student.Scores.Average()
            group student by (avg == 0 ? 0 : avg / 10) into g
            orderby g.Key
            select g;

        // Execute the query.
        foreach (var studentGroup in studentQuery)
        {
            int temp = studentGroup.Key * 10;
            Console.WriteLine("Students with an average between {0} and {1}", temp, temp + 10);
            foreach (var student in studentGroup)
            {
                Console.WriteLine("   {0}, {1}:{2}", student.Last, student.First, student.Scores.Average());
            }
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
     Students with an average between 70 and 80
```

```
        Omelchenko, Svetlana:77.5
        O'Donnell, Claire:72.25
        Garcia, Cesar:75.5
      Students with an average between 80 and 90
        Garcia, Debra:88.25
      Students with an average between 90 and 100
        Mortensen, Sven:93.5
  */
```

Grouping by composite keys

Use a composite key when you want to group elements according to more than one key. You create a composite key by using an anonymous type or a named type to hold the key element. In the following example, assume that a class `Person` has been declared with members named `surname` and `city`. The `group` clause causes a separate group to be created for each set of persons with the same last name and the same city.

```
group person by new {name = person.surname, city = person.city};
```

Use a named type if you must pass the query variable to another method. Create a special class using auto-implemented properties for the keys, and then override the Equals and GetHashCode methods. You can also use a struct, in which case you do not strictly have to override those methods. For more information see How to: Implement a Lightweight Class with Auto-Implemented Properties and How to: Query for Duplicate Files in a Directory Tree. The latter article has a code example that demonstrates how to use a composite key with a named type.

Example

The following example shows the standard pattern for ordering source data into groups when no additional query logic is applied to the groups. This is called a grouping without a continuation. The elements in an array of strings are grouped according to their first letter. The result of the query is an IGrouping<TKey,TElement> type that contains a public `Key` property of type `char` and an IEnumerable<T> collection that contains each item in the grouping.

The result of a `group` clause is a sequence of sequences. Therefore, to access the individual elements within each returned group, use a nested `foreach` loop inside the loop that iterates the group keys, as shown in the following example.

```
class GroupExample1
{
    static void Main()
    {
        // Create a data source.
        string[] words = { "blueberry", "chimpanzee", "abacus", "banana", "apple", "cheese" };

        // Create the query.
        var wordGroups =
            from w in words
            group w by w[0];

        // Execute the query.
        foreach (var wordGroup in wordGroups)
        {
            Console.WriteLine("Words that start with the letter '{0}':", wordGroup.Key);
            foreach (var word in wordGroup)
            {
                Console.WriteLine(word);
            }
        }

        // Keep the console window open in debug mode
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
      Words that start with the letter 'b':
        blueberry
        banana
      Words that start with the letter 'c':
        chimpanzee
        cheese
      Words that start with the letter 'a':
        abacus
        apple
     */
```

Example

This example shows how to perform additional logic on the groups after you have created them, by using a *continuation* with `into`. For more information, see into. The following example queries each group to select only those whose key value is a vowel.

```
class GroupClauseExample2
{
    static void Main()
    {
        // Create the data source.
        string[] words2 = { "blueberry", "chimpanzee", "abacus", "banana", "apple", "cheese", "elephant",
"umbrella", "anteater" };

        // Create the query.
        var wordGroups2 =
            from w in words2
            group w by w[0] into grps
            where (grps.Key == 'a' || grps.Key == 'e' || grps.Key == 'i'
                   || grps.Key == 'o' || grps.Key == 'u')
            select grps;

        // Execute the query.
        foreach (var wordGroup in wordGroups2)
        {
            Console.WriteLine("Groups that start with a vowel: {0}", wordGroup.Key);
            foreach (var word in wordGroup)
            {
                Console.WriteLine("   {0}", word);
            }
        }

        // Keep the console window open in debug mode
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
    Groups that start with a vowel: a
        abacus
        apple
        anteater
    Groups that start with a vowel: e
        elephant
    Groups that start with a vowel: u
        umbrella
*/
```

Remarks

At compile time, `group` clauses are translated into calls to the GroupBy method.

See also

- IGrouping<TKey,TElement>
- GroupBy
- ThenBy
- ThenByDescending
- Query Keywords
- Language Integrated Query (LINQ)
- Create a nested group
- Group query results
- Perform a subquery on a grouping operation

into (C# Reference)

8/25/2018 • 2 minutes to read • Edit Online

The `into` contextual keyword can be used to create a temporary identifier to store the results of a group, join or select clause into a new identifier. This identifier can itself be a generator for additional query commands. When used in a `group` or `select` clause, the use of the new identifier is sometimes referred to as a *continuation*.

Example

The following example shows the use of the `into` keyword to enable a temporary identifier `fruitGroup` which has an inferred type of `IGrouping`. By using the identifier, you can invoke the Count method on each group and select only those groups that contain two or more words.

```
class IntoSample1
{
    static void Main()
    {

        // Create a data source.
        string[] words = { "apples", "blueberries", "oranges", "bananas", "apricots"};

        // Create the query.
        var wordGroups1 =
            from w in words
            group w by w[0] into fruitGroup
            where fruitGroup.Count() >= 2
            select new { FirstLetter = fruitGroup.Key, Words = fruitGroup.Count() };

        // Execute the query. Note that we only iterate over the groups,
        // not the items in each group
        foreach (var item in wordGroups1)
        {
            Console.WriteLine(" {0} has {1} elements.", item.FirstLetter, item.Words);
        }

        // Keep the console window open in debug mode
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
   a has 2 elements.
   b has 2 elements.
*/
```

The use of `into` in a `group` clause is only necessary when you want to perform additional query operations on each group. For more information, see group clause.

For an example of the use of `into` in a `join` clause, see join clause.

See also

- Query Keywords (LINQ)
- LINQ Query Expressions
- group clause

orderby clause (C# Reference)

8/24/2018 • 2 minutes to read • Edit Online

In a query expression, the `orderby` clause causes the returned sequence or subsequence (group) to be sorted in either ascending or descending order. Multiple keys can be specified in order to perform one or more secondary sort operations. The sorting is performed by the default comparer for the type of the element. The default sort order is ascending. You can also specify a custom comparer. However, it is only available by using method-based syntax. For more information, see Sorting Data.

Example

In the following example, the first query sorts the words in alphabetical order starting from A, and second query sorts the same words in descending order. (The `ascending` keyword is the default sort value and can be omitted.)

```
class OrderbySample1
{
    static void Main()
    {
        // Create a delicious data source.
        string[] fruits = { "cherry", "apple", "blueberry" };

        // Query for ascending sort.
        IEnumerable<string> sortAscendingQuery =
            from fruit in fruits
            orderby fruit //"ascending" is default
            select fruit;

        // Query for descending sort.
        IEnumerable<string> sortDescendingQuery =
            from w in fruits
            orderby w descending
            select w;

        // Execute the query.
        Console.WriteLine("Ascending:");
        foreach (string s in sortAscendingQuery)
        {
            Console.WriteLine(s);
        }

        // Execute the query.
        Console.WriteLine(Environment.NewLine + "Descending:");
        foreach (string s in sortDescendingQuery)
        {
            Console.WriteLine(s);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
Ascending:
apple
blueberry
cherry

Descending:
cherry
blueberry
apple
*/
```

Example

The following example performs a primary sort on the students' last names, and then a secondary sort on their first names.

```
class OrderbySample2
{
    // The element type of the data source.
    public class Student
    {
        public string First { get; set; }
        public string Last { get; set; }
        public int ID { get; set; }
    }
```

```
    public static List<Student> GetStudents()
    {
        // Use a collection initializer to create the data source. Note that each element
        //  in the list contains an inner sequence of scores.
        List<Student> students = new List<Student>
        {
           new Student {First="Svetlana", Last="Omelchenko", ID=111},
           new Student {First="Claire", Last="O'Donnell", ID=112},
           new Student {First="Sven", Last="Mortensen", ID=113},
           new Student {First="Cesar", Last="Garcia", ID=114},
           new Student {First="Debra", Last="Garcia", ID=115}
        };

        return students;

    }
    static void Main(string[] args)
    {
        // Create the data source.
        List<Student> students = GetStudents();

        // Create the query.
        IEnumerable<Student> sortedStudents =
            from student in students
            orderby student.Last ascending, student.First ascending
            select student;

        // Execute the query.
        Console.WriteLine("sortedStudents:");
        foreach (Student student in sortedStudents)
            Console.WriteLine(student.Last + " " + student.First);

        // Now create groups and sort the groups. The query first sorts the names
        // of all students so that they will be in alphabetical order after they are
        // grouped. The second orderby sorts the group keys in alpha order.
        var sortedGroups =
            from student in students
            orderby student.Last, student.First
            group student by student.Last[0] into newGroup
            orderby newGroup.Key
            select newGroup;

        // Execute the query.
        Console.WriteLine(Environment.NewLine + "sortedGroups:");
        foreach (var studentGroup in sortedGroups)
        {
            Console.WriteLine(studentGroup.Key);
            foreach (var student in studentGroup)
            {
                Console.WriteLine("   {0}, {1}", student.Last, student.First);
            }
        }

        // Keep the console window open in debug mode
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
sortedStudents:
Garcia Cesar
Garcia Debra
Mortensen Sven
O'Donnell Claire
Omelchenko Svetlana

sortedGroups:
G
   Garcia, Cesar
```

```
    Garcia, Cesar
    Garcia, Debra
M
    Mortensen, Sven
O
    O'Donnell, Claire
    Omelchenko, Svetlana
*/
```

Remarks

At compile time, the `orderby` clause is translated to a call to the OrderBy method. Multiple keys in the `orderby` clause translate to ThenBy method calls.

See also

- C# Reference
- Query Keywords (LINQ)
- Language Integrated Query (LINQ)
- group clause
- Getting Started with LINQ in C#

join clause (C# Reference)

8/19/2018 • 9 minutes to read • Edit Online

The `join` clause is useful for associating elements from different source sequences that have no direct relationship in the object model. The only requirement is that the elements in each source share some value that can be compared for equality. For example, a food distributor might have a list of suppliers of a certain product, and a list of buyers. A `join` clause can be used, for example, to create a list of the suppliers and buyers of that product who are all in the same specified region.

A `join` clause takes two source sequences as input. The elements in each sequence must either be or contain a property that can be compared to a corresponding property in the other sequence. The `join` clause compares the specified keys for equality by using the special `equals` keyword. All joins performed by the `join` clause are equijoins. The shape of the output of a `join` clause depends on the specific type of join you are performing. The following are three most common join types:

- Inner join
- Group join
- Left outer join

Inner join

The following example shows a simple inner equijoin. This query produces a flat sequence of "product name / category" pairs. The same category string will appear in multiple elements. If an element from `categories` has no matching `products`, that category will not appear in the results.

```
var innerJoinQuery =
    from category in categories
    join prod in products on category.ID equals prod.CategoryID
    select new { ProductName = prod.Name, Category = category.Name }; //produces flat sequence
```

For more information, see Perform inner joins.

Group join

A `join` clause with an `into` expression is called a group join.

```
var innerGroupJoinQuery =
    from category in categories
    join prod in products on category.ID equals prod.CategoryID into prodGroup
    select new { CategoryName = category.Name, Products = prodGroup };
```

A group join produces a hierarchical result sequence, which associates elements in the left source sequence with one or more matching elements in the right side source sequence. A group join has no equivalent in relational terms; it is essentially a sequence of object arrays.

If no elements from the right source sequence are found to match an element in the left source, the `join` clause will produce an empty array for that item. Therefore, the group join is still basically an inner-equijoin except that the result sequence is organized into groups.

If you just select the results of a group join, you can access the items, but you cannot identify the key that they match on. Therefore, it is generally more useful to select the results of the group join into a new type that also has the key name, as shown in the previous example.

You can also, of course, use the result of a group join as the generator of another subquery:

```
var innerGroupJoinQuery2 =
    from category in categories
    join prod in products on category.ID equals prod.CategoryID into prodGroup
    from prod2 in prodGroup
    where prod2.UnitPrice > 2.50M
    select prod2;
```

For more information, see Perform grouped joins.

Left outer join

In a left outer join, all the elements in the left source sequence are returned, even if no matching elements are in the right sequence. To perform a left outer join in LINQ, use the `DefaultIfEmpty` method in combination with a group join to specify a default right-side element to produce if a left-side element has no matches. You can use `null` as the default value for any reference type, or you can specify a user-defined default type. In the following example, a user-defined default type is shown:

```
var leftOuterJoinQuery =
    from category in categories
    join prod in products on category.ID equals prod.CategoryID into prodGroup
    from item in prodGroup.DefaultIfEmpty(new Product { Name = String.Empty, CategoryID = 0 })
    select new { CatName = category.Name, ProdName = item.Name };
```

For more information, see Perform left outer joins.

The equals operator

A `join` clause performs an equijoin. In other words, you can only base matches on the equality of two keys. Other types of comparisons such as "greater than" or "not equals" are not supported. To make clear that all joins are equijoins, the `join` clause uses the `equals` keyword instead of the `==` operator. The `equals` keyword can only be used in a `join` clause and it differs from the `==` operator in one important way. With `equals`, the left key consumes the outer source sequence, and the right key consumes the inner source. The outer source is only in scope on the left side of `equals` and the inner source sequence is only in scope on the right side.

Non-equijoins

You can perform non-equijoins, cross joins, and other custom join operations by using multiple `from` clauses to introduce new sequences independently into a query. For more information, see Perform custom join operations.

Joins on object collections vs. relational tables

In a LINQ query expression, join operations are performed on object collections. Object collections cannot be "joined" in exactly the same way as two relational tables. In LINQ, explicit `join` clauses are only required when two source sequences are not tied by any relationship. When working with LINQ to SQL, foreign key tables are represented in the object model as properties of the primary table. For example, in the Northwind database, the Customer table has a foreign key relationship with the Orders table. When you map the tables to the object model, the Customer class has an Orders property that contains the collection of Orders associated with that Customer. In effect, the join has already been done for you.

For more information about querying across related tables in the context of LINQ to SQL, see How to: Map

Database Relationships.

Composite keys

You can test for equality of multiple values by using a composite key. For more information, see Join by using composite keys. Composite keys can be also used in a `group` clause.

Example

The following example compares the results of an inner join, a group join, and a left outer join on the same data sources by using the same matching keys. Some extra code is added to these examples to clarify the results in the console display.

```
class JoinDemonstration
{
    #region Data

    class Product
    {
        public string Name { get; set; }
        public int CategoryID { get; set; }
    }

    class Category
    {
        public string Name { get; set; }
        public int ID { get; set; }
    }

    // Specify the first data source.
    List<Category> categories = new List<Category>()
    {
        new Category(){Name="Beverages", ID=001},
        new Category(){ Name="Condiments", ID=002},
        new Category(){ Name="Vegetables", ID=003},
        new Category() {  Name="Grains", ID=004},
        new Category() {  Name="Fruit", ID=005}
    };

    // Specify the second data source.
    List<Product> products = new List<Product>()
   {
      new Product{Name="Cola",  CategoryID=001},
      new Product{Name="Tea",  CategoryID=001},
      new Product{Name="Mustard", CategoryID=002},
      new Product{Name="Pickles", CategoryID=002},
      new Product{Name="Carrots", CategoryID=003},
      new Product{Name="Bok Choy", CategoryID=003},
      new Product{Name="Peaches", CategoryID=005},
      new Product{Name="Melons", CategoryID=005},
    };
    #endregion


    static void Main(string[] args)
    {
        JoinDemonstration app = new JoinDemonstration();

        app.InnerJoin();
        app.GroupJoin();
        app.GroupInnerJoin();
        app.GroupJoin3();
        app.LeftOuterJoin();
        app.LeftOuterJoin2();
```

```
        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }

    void InnerJoin()
    {
        // Create the query that selects
        // a property from each element.
        var innerJoinQuery =
           from category in categories
           join prod in products on category.ID equals prod.CategoryID
           select new { Category = category.ID, Product = prod.Name };

        Console.WriteLine("InnerJoin:");
        // Execute the query. Access results
        // with a simple foreach statement.
        foreach (var item in innerJoinQuery)
        {
            Console.WriteLine("{0,-10}{1}", item.Product, item.Category);
        }
        Console.WriteLine("InnerJoin: {0} items in 1 group.", innerJoinQuery.Count());
        Console.WriteLine(System.Environment.NewLine);

    }

    void GroupJoin()
    {
        // This is a demonstration query to show the output
        // of a "raw" group join. A more typical group join
        // is shown in the GroupInnerJoin method.
        var groupJoinQuery =
           from category in categories
           join prod in products on category.ID equals prod.CategoryID into prodGroup
           select prodGroup;

        // Store the count of total items (for demonstration only).
        int totalItems = 0;

        Console.WriteLine("Simple GroupJoin:");

        // A nested foreach statement is required to access group items.
        foreach (var prodGrouping in groupJoinQuery)
        {
            Console.WriteLine("Group:");
            foreach (var item in prodGrouping)
            {
                totalItems++;
                Console.WriteLine("   {0,-10}{1}", item.Name, item.CategoryID);
            }
        }
        Console.WriteLine("Unshaped GroupJoin: {0} items in {1} unnamed groups", totalItems,
groupJoinQuery.Count());
        Console.WriteLine(System.Environment.NewLine);
    }

    void GroupInnerJoin()
    {
        var groupJoinQuery2 =
            from category in categories
            orderby category.ID
            join prod in products on category.ID equals prod.CategoryID into prodGroup
            select new
            {
                Category = category.Name,
                Products = from prod2 in prodGroup
                           orderby prod2.Name
                           select prod2
            };
```

```
        };

        //Console.WriteLine("GroupInnerJoin:");
        int totalItems = 0;

        Console.WriteLine("GroupInnerJoin:");
        foreach (var productGroup in groupJoinQuery2)
        {
            Console.WriteLine(productGroup.Category);
            foreach (var prodItem in productGroup.Products)
            {
                totalItems++;
                Console.WriteLine("  {0,-10} {1}", prodItem.Name, prodItem.CategoryID);
            }
        }
        Console.WriteLine("GroupInnerJoin: {0} items in {1} named groups", totalItems,
groupJoinQuery2.Count());
        Console.WriteLine(System.Environment.NewLine);
    }

    void GroupJoin3()
    {

        var groupJoinQuery3 =
            from category in categories
            join product in products on category.ID equals product.CategoryID into prodGroup
            from prod in prodGroup
            orderby prod.CategoryID
            select new { Category = prod.CategoryID, ProductName = prod.Name };

        //Console.WriteLine("GroupInnerJoin:");
        int totalItems = 0;

        Console.WriteLine("GroupJoin3:");
        foreach (var item in groupJoinQuery3)
        {
            totalItems++;
            Console.WriteLine("   {0}:{1}", item.ProductName, item.Category);
        }

        Console.WriteLine("GroupJoin3: {0} items in 1 group", totalItems, groupJoinQuery3.Count());
        Console.WriteLine(System.Environment.NewLine);
    }

    void LeftOuterJoin()
    {
        // Create the query.
        var leftOuterQuery =
           from category in categories
           join prod in products on category.ID equals prod.CategoryID into prodGroup
           select prodGroup.DefaultIfEmpty(new Product() { Name = "Nothing!", CategoryID = category.ID });

        // Store the count of total items (for demonstration only).
        int totalItems = 0;

        Console.WriteLine("Left Outer Join:");

        // A nested foreach statement  is required to access group items
        foreach (var prodGrouping in leftOuterQuery)
        {
            Console.WriteLine("Group:", prodGrouping.Count());
            foreach (var item in prodGrouping)
            {
                totalItems++;
                Console.WriteLine("  {0,-10}{1}", item.Name, item.CategoryID);
            }
        }
        Console.WriteLine("LeftOuterJoin: {0} items in {1} groups", totalItems, leftOuterQuery.Count());
        Console.WriteLine(System.Environment.NewLine);
    }
```

```
    }

    void LeftOuterJoin2()
    {
        // Create the query.
        var leftOuterQuery2 =
           from category in categories
           join prod in products on category.ID equals prod.CategoryID into prodGroup
           from item in prodGroup.DefaultIfEmpty()
           select new { Name = item == null ? "Nothing!" : item.Name, CategoryID = category.ID };

        Console.WriteLine("LeftOuterJoin2: {0} items in 1 group", leftOuterQuery2.Count());
        // Store the count of total items
        int totalItems = 0;

        Console.WriteLine("Left Outer Join 2:");

        // Groups have been flattened.
        foreach (var item in leftOuterQuery2)
        {
            totalItems++;
            Console.WriteLine("{0,-10}{1}", item.Name, item.CategoryID);
        }
        Console.WriteLine("LeftOuterJoin2: {0} items in 1 group", totalItems);
    }
}
/*Output:

InnerJoin:
Cola      1
Tea       1
Mustard   2
Pickles   2
Carrots   3
Bok Choy  3
Peaches   5
Melons    5
InnerJoin: 8 items in 1 group.


Unshaped GroupJoin:
Group:
    Cola      1
    Tea       1
Group:
    Mustard   2
    Pickles   2
Group:
    Carrots   3
    Bok Choy  3
Group:
Group:
    Peaches   5
    Melons    5
Unshaped GroupJoin: 8 items in 5 unnamed groups


GroupInnerJoin:
Beverages
    Cola       1
    Tea        1
Condiments
    Mustard    2
    Pickles    2
Vegetables
    Bok Choy   3
    Carrots    3
Grains
Fruit
```

```
    Melons     5
    Peaches    5
GroupInnerJoin: 8 items in 5 named groups


GroupJoin3:
    Cola:1
    Tea:1
    Mustard:2
    Pickles:2
    Carrots:3
    Bok Choy:3
    Peaches:5
    Melons:5
GroupJoin3: 8 items in 1 group


Left Outer Join:
Group:
    Cola      1
    Tea       1
Group:
    Mustard   2
    Pickles   2
Group:
    Carrots   3
    Bok Choy  3
Group:
    Nothing!  4
Group:
    Peaches   5
    Melons    5
LeftOuterJoin: 9 items in 5 groups


LeftOuterJoin2: 9 items in 1 group
Left Outer Join 2:
Cola      1
Tea       1
Mustard   2
Pickles   2
Carrots   3
Bok Choy  3
Nothing!  4
Peaches   5
Melons    5
LeftOuterJoin2: 9 items in 1 group
Press any key to exit.
*/
```

Remarks

A `join` clause that is not followed by `into` is translated into a Join method call. A `join` clause that is followed by `into` is translated to a GroupJoin method call.

See also

- Query Keywords (LINQ)
- Language Integrated Query (LINQ)
- Join Operations
- group clause
- Perform left outer joins
- Perform inner joins

- Perform grouped joins
- Order the results of a join clause
- Join by using composite keys
- Compatible database systems for Visual Studio

let clause (C# Reference)

8/19/2018 • 2 minutes to read • Edit Online

In a query expression, it is sometimes useful to store the result of a sub-expression in order to use it in subsequent clauses. You can do this with the `let` keyword, which creates a new range variable and initializes it with the result of the expression you supply. Once initialized with a value, the range variable cannot be used to store another value. However, if the range variable holds a queryable type, it can be queried.

Example

In the following example `let` is used in two ways:

1. To create an enumerable type that can itself be queried.
2. To enable the query to call `ToLower` only one time on the range variable `word`. Without using `let`, you would have to call `ToLower` in each predicate in the `where` clause.

```
class LetSample1
{
    static void Main()
    {
        string[] strings =
        {
            "A penny saved is a penny earned.",
            "The early bird catches the worm.",
            "The pen is mightier than the sword."
        };

        // Split the sentence into an array of words
        // and select those whose first letter is a vowel.
        var earlyBirdQuery =
            from sentence in strings
            let words = sentence.Split(' ')
            from word in words
            let w = word.ToLower()
            where w[0] == 'a' || w[0] == 'e'
                || w[0] == 'i' || w[0] == 'o'
                || w[0] == 'u'
            select word;

        // Execute the query.
        foreach (var v in earlyBirdQuery)
        {
            Console.WriteLine("\"{0}\" starts with a vowel", v);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
    "A" starts with a vowel
    "is" starts with a vowel
    "a" starts with a vowel
    "earned." starts with a vowel
    "early" starts with a vowel
    "is" starts with a vowel
*/
```

See also

- C# Reference
- Query Keywords (LINQ)
- Language Integrated Query (LINQ)
- Getting Started with LINQ in C#
- Handle exceptions in query expressions

ascending (C# Reference)

8/25/2018 • 2 minutes to read • Edit Online

The `ascending` contextual keyword is used in the orderby clause in query expressions to specify that the sort order is from smallest to largest. Because `ascending` is the default sort order, you do not have to specify it.

Example

The following example shows the use of `ascending` in an orderby clause.

```
IEnumerable<string> sortAscendingQuery =
    from vegetable in vegetables
    orderby vegetable ascending
    select vegetable;
```

See Also

- C# Reference
- LINQ Query Expressions
- descending

descending (C# Reference)

8/25/2018 • 2 minutes to read • Edit Online

The `descending` contextual keyword is used in the orderby clause in query expressions to specify that the sort order is from largest to smallest.

Example

The following example shows the use of `descending` in an orderby clause.

```
IEnumerable<string> sortDescendingQuery =
    from vegetable in vegetables
    orderby vegetable descending
    select vegetable;
```

See also

- C# Reference
- LINQ Query Expressions
- ascending

on (C# Reference)

8/24/2018 • 2 minutes to read • Edit Online

The `on` contextual keyword is used in the join clause of a query expression to specify the join condition.

Example

The following example shows the use of `on` in a `join` clause.

```
var innerJoinQuery =
    from category in categories
    join prod in products on category.ID equals prod.CategoryID
    select new { ProductName = prod.Name, Category = category.Name };
```

See also

- C# Reference
- Language Integrated Query (LINQ)

equals (C# Reference)

8/25/2018 • 2 minutes to read • Edit Online

The `equals` contextual keyword is used in a `join` clause in a query expression to compare the elements of two sequences. For more information, see join clause.

Example

The following example shows the use of the `equals` keyword in a `join` clause.

```
var innerJoinQuery =
    from category in categories
    join prod in products on category.ID equals prod.CategoryID
    select new { ProductName = prod.Name, Category = category.Name };
```

See also

- Language Integrated Query (LINQ)

by (C# Reference)

8/25/2018 • 2 minutes to read • Edit Online

The `by` contextual keyword is used in the `group` clause in a query expression to specify how the returned items should be grouped. For more information, see group clause.

Example

The following example shows the use of the `by` contextual keyword in a `group` clause to specify that the students should be grouped according to the first letter of the last name of each student.

```
var query = from student in students
            group student by student.LastName[0];
```

See also

- LINQ Query Expressions

in (C# Reference)

9/5/2018 • 2 minutes to read • Edit Online

The `in` keyword is used in four contexts:

- generic type parameters in generic interfaces and delegates.
- As a parameter modifier, which lets you pass an argument to a method by reference rather than by value.
- foreach statements.
- from clauses in LINQ query expressions.
- join clauses in LINQ query expressions.

See Also

- C# Keywords
- C# Reference