Contents

Statements, Expressions, and Operators (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

The C# code that comprises an application consists of statements made up of keywords, expressions and operators. This section contains information regarding these fundamental elements of a C# program.

For more information, see:

- Statements
- Expressions
  - Expression-bodied members
- Operators
- Anonymous Functions
- Overloadable Operators
- Conversion Operators
  - Using Conversion Operators
  - How to: Implement User-Defined Conversions Between Structs
- Equality Comparisons

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# Programming Guide
- Casting and Type Conversions

Statements (C# Programming Guide)

1/23/2019 • 6 minutes to read • Edit Online

The actions that a program takes are expressed in statements. Common actions include declaring variables, assigning values, calling methods, looping through collections, and branching to one or another block of code, depending on a given condition. The order in which statements are executed in a program is called the flow of control or flow of execution. The flow of control may vary every time that a program is run, depending on how the program reacts to input that it receives at run time.

A statement can consist of a single line of code that ends in a semicolon, or a series of single-line statements in a block. A statement block is enclosed in {} brackets and can contain nested blocks. The following code shows two examples of single-line statements, and a multi-line statement block:

```
    static void Main()
    {
        // Declaration statement.
        int counter;

        // Assignment statement.
        counter = 1;

        // Error! This is an expression, not an expression statement.
        // counter + 1;

        // Declaration statements with initializers are functionally
        // equivalent to  declaration statement followed by assignment statement:
        int[] radii = { 15, 32, 108, 74, 9 }; // Declare and initialize an array.
        const double pi = 3.14159; // Declare and initialize  constant.

        // foreach statement block that contains multiple statements.
        foreach (int radius in radii)
        {
            // Declaration statement with initializer.
            double circumference = pi * (2 * radius);

            // Expression statement (method invocation). A single-line
            // statement can span multiple text lines because line breaks
            // are treated as white space, which is ignored by the compiler.
            System.Console.WriteLine("Radius of circle #{0} is {1}. Circumference = {2:N2}",
                                    counter, radius, circumference);

            // Expression statement (postfix increment).
            counter++;

        } // End of foreach statement block
    } // End of Main method body.
} // End of SimpleStatements class.
/*
   Output:
    Radius of circle #1 = 15. Circumference = 94.25
    Radius of circle #2 = 32. Circumference = 201.06
    Radius of circle #3 = 108. Circumference = 678.58
    Radius of circle #4 = 74. Circumference = 464.96
    Radius of circle #5 = 9. Circumference = 56.55
*/
```

Types of Statements

The following table lists the various types of statements in C# and their associated keywords, with links to topics that include more information:

CATEGORY	C# KEYWORDS / NOTES
Declaration statements	A declaration statement introduces a new variable or constant. A variable declaration can optionally assign a value to the variable. In a constant declaration, the assignment is required.
Expression statements	Expression statements that calculate a value must store the value in a variable. For more information, see Expression Statements.
Selection statements	Selection statements enable you to branch to different sections of code, depending on one or more specified conditions. For more information, see the following topics: if, else, switch, case
Iteration statements	Iteration statements enable you to loop through collections like arrays, or perform the same set of statements repeatedly until a specified condition is met. For more information, see the following topics: do, for, foreach, in, while
Jump statements	Jump statements transfer control to another section of code. For more information, see the following topics: break, continue, default, goto, return, yield
Exception handling statements	Exception handling statements enable you to gracefully recover from exceptional conditions that occur at run time. For more information, see the following topics: throw, try-catch, try-finally, try-catch-finally
Checked and unchecked	Checked and unchecked statements enable you to specify whether numerical operations are allowed to cause an overflow when the result is stored in a variable that is too small to hold the resulting value. For more information, see checked and unchecked.
The `await` statement	If you mark a method with the async modifier, you can use the await operator in the method. When control reaches an `await` expression in the async method, control returns to the caller, and progress in the method is suspended until the awaited task completes. When the task is complete, execution can resume in the method. For a simple example, see the "Async Methods" section of Methods. For more information, see Asynchronous Programming with async and await.

CATEGORY	C# KEYWORDS / NOTES
The `yield return` statement	An iterator performs a custom iteration over a collection, such as a list or an array. An iterator uses the yield return statement to return each element one at a time. When a `yield return` statement is reached, the current location in code is remembered. Execution is restarted from that location when the iterator is called the next time. For more information, see Iterators.
The `fixed` statement	The fixed statement prevents the garbage collector from relocating a movable variable. For more information, see fixed.
The `lock` statement	The lock statement enables you to limit access to blocks of code to only one thread at a time. For more information, see lock.
Labeled statements	You can give a statement a label and then use the goto keyword to jump to the labeled statement. (See the example in the following row.)
The empty statement	The empty statement consists of a single semicolon. It does nothing and can be used in places where a statement is required but no action needs to be performed.

Declaration statements

The following code shows examples of variable declarations with and without an initial assignment, and a constant declaration with the necessary initialization.

```
// Variable declaration statements.
double area;
double radius = 2;

// Constant declaration statement.
const double pi = 3.14159;
```

Expression statements

The following code shows examples of expression statements, including assignment, object creation with assignment, and method invocation.

```
// Expression statement (assignment).
area = 3.14 * (radius * radius);

// Error. Not  statement because no assignment:
//circ * 2;

// Expression statement (method invocation).
System.Console.WriteLine();

// Expression statement (new object creation).
System.Collections.Generic.List<string> strings =
    new System.Collections.Generic.List<string>();
```

The empty statement

The following examples show two uses for an empty statement:

```
void ProcessMessages()
{
    while (ProcessMessage())
        ; // Statement needed here.
}

void F()
{
    //...
    if (done) goto exit;
//...
exit:
    ; // Statement needed here.
}
```

Embedded Statements

Some statements, including do, while, for, and foreach, always have an embedded statement that follows them. This embedded statement may be either a single statement or multiple statements enclosed by {} brackets in a statement block. Even single-line embedded statements can be enclosed in {} brackets, as shown in the following example:

```
// Recommended style. Embedded statement in  block.
foreach (string s in System.IO.Directory.GetDirectories(
                        System.Environment.CurrentDirectory))
{
    System.Console.WriteLine(s);
}

// Not recommended.
foreach (string s in System.IO.Directory.GetDirectories(
                        System.Environment.CurrentDirectory))
    System.Console.WriteLine(s);
```

An embedded statement that is not enclosed in {} brackets cannot be a declaration statement or a labeled statement. This is shown in the following example:

```
if(pointB == true)
    //Error CS1023:
    int radius = 5;
```

Put the embedded statement in a block to fix the error:

```
if (b == true)
{
    // OK:
    System.DateTime d = System.DateTime.Now;
    System.Console.WriteLine(d.ToLongDateString());
}
```

Nested Statement Blocks

Statement blocks can be nested, as shown in the following code:

```
foreach (string s in System.IO.Directory.GetDirectories(
    System.Environment.CurrentDirectory))
{
    if (s.StartsWith("CSharp"))
    {
        if (s.EndsWith("TempFolder"))
        {
            return s;
        }
    }

}
return "Not found.";
```

Unreachable Statements

If the compiler determines that the flow of control can never reach a particular statement under any circumstances, it will produce warning CS0162, as shown in the following example:

```
// An over-simplified example of unreachable code.
const int val = 5;
if (val < 4)
{
    System.Console.WriteLine("I'll never write anything."); //CS0162
}
```

Related Sections

- Statement Keywords
- Expressions
- 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# Programming Guide

Expressions (C# Programming Guide)

1/23/2019 • 4 minutes to read • Edit Online

An *expression* is a sequence of one or more operands and zero or more operators that can be evaluated to a single value, object, method, or namespace. Expressions can consist of a literal value, a method invocation, an operator and its operands, or a *simple name*. Simple names can be the name of a variable, type member, method parameter, namespace or type.

Expressions can use operators that in turn use other expressions as parameters, or method calls whose parameters are in turn other method calls, so expressions can range from simple to very complex. Following are two examples of expressions:

```
((x < 10) && ( x > 5)) || ((x > 20) && (x < 25));

System.Convert.ToInt32("35");
```

Expression values

In most of the contexts in which expressions are used, for example in statements or method parameters, the expression is expected to evaluate to some value. If x and y are integers, the expression `x + y` evaluates to a numeric value. The expression `new MyClass()` evaluates to a reference to a new instance of a `MyClass` class. The expression `myClass.ToString()` evaluates to a string because that is the return type of the method. However, although a namespace name is classified as an expression, it does not evaluate to a value and therefore can never be the final result of any expression. You cannot pass a namespace name to a method parameter, or use it in a new expression, or assign it to a variable. You can only use it as a sub-expression in a larger expression. The same is true for types (as distinct from System.Type objects), method group names (as distinct from specific methods), and event add and remove accessors.

Every value has an associated type. For example, if x and y are both variables of type `int`, the value of the expression `x + y` is also typed as `int`. If the value is assigned to a variable of a different type, or if x and y are different types, the rules of type conversion are applied. For more information about how such conversions work, see Casting and Type Conversions.

Overflows

Numeric expressions may cause overflows if the value is larger than the maximum value of the value's type. For more information, see Checked and Unchecked and Explicit Numeric Conversions Table.

Operator precedence and associativity

The manner in which an expression is evaluated is governed by the rules of associativity and operator precedence. For more information, see Operators.

Most expressions, except assignment expressions and method invocation expressions, must be embedded in a statement. For more information, see Statements.

Literals and simple names

The two simplest types of expressions are literals and simple names. A literal is a constant value that has no name. For example, in the following code example, both `5` and `"Hello World"` are literal values:

```
// Expression statements.
int i = 5;
string s = "Hello World";
```

For more information on literals, see Types.

In the preceding example, both `i` and `s` are simple names that identify local variables. When those variables are used in an expression, the variable name evaluates to the value that is currently stored in the variable's location in memory. This is shown in the following example:

```
int num = 5;
System.Console.WriteLine(num); // Output: 5
num = 6;
System.Console.WriteLine(num); // Output: 6
```

Invocation expressions

In the following code example, the call to `DoWork` is an invocation expression.

```
DoWork();
```

A method invocation requires the name of the method, either as a name as in the previous example, or as the result of another expression, followed by parenthesis and any method parameters. For more information, see Methods. A delegate invocation uses the name of a delegate and method parameters in parenthesis. For more information, see Delegates. Method invocations and delegate invocations evaluate to the return value of the method, if the method returns a value. Methods that return void cannot be used in place of a value in an expression.

Query expressions

The same rules for expressions in general apply to query expressions. For more information, see LINQ Query Expressions.

Lambda expressions

Lambda expressions represent "inline methods" that have no name but can have input parameters and multiple statements. They are used extensively in LINQ to pass arguments to methods. Lambda expressions are compiled to either delegates or expression trees depending on the context in which they are used. For more information, see Lambda Expressions.

Expression trees

Expression trees enable expressions to be represented as data structures. They are used extensively by LINQ providers to translate query expressions into code that is meaningful in some other context, such as a SQL database. For more information, see Expression Trees (C#).

Expression body definitions

C# supports *expression-bodied members*, which allow you to supply a concise expression body definition for methods, constructors, finalizers, properties, and indexers. For more information, see Expression-bodied members.

Remarks

Whenever a variable, object property, or object indexer access is identified from an expression, the value of that item is used as the value of the expression. An expression can be placed anywhere in C# where a value or object is required, as long as the expression ultimately evaluates to the required type.

See also

- C# Programming Guide
- Methods
- Delegates
- Operators
- Types
- LINQ Query Expressions

Expression-bodied members (C# programming guide)

2/7/2019 • 3 minutes to read • Edit Online

Expression body definitions let you provide a member's implementation in a very concise, readable form. You can use an expression body definition whenever the logic for any supported member, such as a method or property, consists of a single expression. An expression body definition has the following general syntax:

```
member => expression;
```

where *expression* is a valid expression.

Support for expression body definitions was introduced for methods and read-only properties in C# 6 and was expanded in C# 7.0. Expression body definitions can be used with the type members listed in the following table:

MEMBER	SUPPORTED AS OF...
Method	C# 6
Read-only property	C# 6
Property	C# 7.0
Constructor	C# 7.0
Finalizer	C# 7.0
Indexer	C# 7.0

Methods

An expression-bodied method consists of a single expression that returns a value whose type matches the method's return type, or, for methods that return `void`, that performs some operation. For example, types that override the ToString method typically include a single expression that returns the string representation of the current object.

The following example defines a `Person` class that overrides the ToString method with an expression body definition. It also defines a `DisplayName` method that displays a name to the console. Note that the `return` keyword is not used in the `ToString` expression body definition.

```
using System;

public class Person
{
   public Person(string firstName, string lastName)
   {
      fname = firstName;
      lname = lastName;
   }

   private string fname;
   private string lname;

   public override string ToString() => $"{fname} {lname}".Trim();
   public void DisplayName() => Console.WriteLine(ToString());
}

class Example
{
   static void Main()
   {
      Person p = new Person("Mandy", "Dejesus");
      Console.WriteLine(p);
      p.DisplayName();
   }
}
```

For more information, see Methods (C# Programming Guide).

Read-only properties

Starting with C# 6, you can use expression body definition to implement a read-only property. To do that, use the following syntax:

```
PropertyType PropertyName => expression;
```

The following example defines a `Location` class whose read-only `Name` property is implemented as an expression body definition that returns the value of the private `locationName` field:

```
public class Location
{
   private string locationName;

   public Location(string name)
   {
      locationName = name;
   }

   public string Name => locationName;
}
```

For more information about properties, see Properties (C# Programming Guide).

Properties

Starting with C# 7.0, you can use expression body definitions to implement property `get` and `set` accessors. The following example demonstrates how to do that:

```
public class Location
{
   private string locationName;

   public Location(string name) => Name = name;

   public string Name
   {
      get => locationName;
      set => locationName = value;
   }
}
```

For more information about properties, see Properties (C# Programming Guide).

Constructors

An expression body definition for a constructor typically consists of a single assignment expression or a method call that handles the constructor's arguments or initializes instance state.

The following example defines a `Location` class whose constructor has a single string parameter named *name*. The expression body definition assigns the argument to the `Name` property.

```
public class Location
{
   private string locationName;

   public Location(string name) => Name = name;

   public string Name
   {
      get => locationName;
      set => locationName = value;
   }
}
```

For more information, see Constructors (C# Programming Guide).

Finalizers

An expression body definition for a finalizer typically contains cleanup statements, such as statements that release unmanaged resources.

The following example defines a finalizer that uses an expression body definition to indicate that the finalizer has been called.

```
using System;

public class Destroyer
{
   public override string ToString() => GetType().Name;

   ~Destroyer() => Console.WriteLine($"The {ToString()} destructor is executing.");
}
```

For more information, see Finalizers (C# Programming Guide).

Indexers

Like properties, an indexer's get and set accessors consist of expression body definitions if the get accessor consists of a single statement that returns a value or the set accessor performs a simple assignment.

The following example defines a class named `Sports` that includes an internal String array that contains the names of a number of sports. Both the indexer's get and set accessors are implemented as expression body definitions.

```
using System;
using System.Collections.Generic;

public class Sports
{
   private string[] types = { "Baseball", "Basketball", "Football",
                              "Hockey", "Soccer", "Tennis",
                              "Volleyball" };

   public string this[int i]
   {
      get => types[i];
      set => types[i] = value;
   }
}
```

For more information, see Indexers (C# Programming Guide).

Operators (C# Programming Guide)

1/23/2019 • 5 minutes to read • Edit Online

In C#, an *operator* is a program element that is applied to one or more *operands* in an expression or statement. Operators that take one operand, such as the increment operator (`++`) or `new`, are referred to as *unary* operators. Operators that take two operands, such as arithmetic operators (`+`, `-`, `*`, `/`), are referred to as *binary* operators. One operator, the conditional operator (`?:`), takes three operands and is the sole ternary operator in C#.

The following C# statement contains a single unary operator and a single operand. The increment operator, `++`, modifies the value of the operand `y`.

```
y++;
```

The following C# statement contains two binary operators, each with two operands. The assignment operator, `=`, has the integer variable `y` and the expression `2 + 3` as operands. The expression `2 + 3` itself consists of the addition operator and two operands, `2` and `3`.

```
y = 2 + 3;
```

Operators, evaluation, and operator precedence

An operand can be a valid expression that is composed of any length of code, and it can comprise any number of sub expressions. In an expression that contains multiple operators, the order in which the operators are applied is determined by *operator precedence*, *associativity*, and parentheses.

Each operator has a defined precedence. In an expression that contains multiple operators that have different precedence levels, the precedence of the operators determines the order in which the operators are evaluated. For example, the following statement assigns 3 to `n1`.

`n1 = 11 - 2 * 4;`

The multiplication is executed first because multiplication takes precedence over subtraction.

The following table separates the operators into categories based on the type of operation they perform. The categories are listed in order of precedence.

Primary Operators

EXPRESSION	DESCRIPTION
x.y x?.y	Member access Conditional member access
f(x)	Method and delegate invocation
a[x] a?[x]	Array and indexer access Conditional array and indexer access

EXPRESSION	DESCRIPTION
x++	Post-increment
x--	Post-decrement
new T(...)	Object and delegate creation
`new` T(...){...}	Object creation with initializer. See Object and Collection Initializers.
`new` {...}	Anonymous object initializer. See Anonymous Types.
`new` T[...]	Array creation. See Arrays.
typeof(T)	Obtain System.Type object for T
checked(x)	Evaluate expression in checked context
unchecked(x)	Evaluate expression in unchecked context
default (T)	Obtain default value of type T
delegate {}	Anonymous function (anonymous method)

Unary Operators

EXPRESSION	DESCRIPTION
+x	Identity
-x	Negation
!x	Logical negation
~x	Bitwise negation
++x	Pre-increment
--x	Pre-decrement
(T)x	Explicitly convert x to type T

Multiplicative Operators

EXPRESSION	DESCRIPTION
*	Multiplication
/	Division
%	Remainder

Additive Operators

EXPRESSION	DESCRIPTION
x + y	Addition, string concatenation, delegate combination
x - y	Subtraction, delegate removal

Shift Operators

EXPRESSION	DESCRIPTION
x << y	Shift left
x >> y	Shift right

Relational and Type Operators

EXPRESSION	DESCRIPTION
x < y	Less than
x > y	Greater than
x <= y	Less than or equal
x >= y	Greater than or equal
x is T	Return true if x is a T, false otherwise
x as T	Return x typed as T, or null if x is not a T

Equality Operators

EXPRESSION	DESCRIPTION
x == y	Equal
x != y	Not equal

Logical, Conditional, and Null Operators

CATEGORY	EXPRESSION	DESCRIPTION
Logical AND	x & y	Integer bitwise AND, Boolean logical AND
Logical XOR	x ^ y	Integer bitwise XOR, Boolean logical XOR
Logical OR	x | y	Integer bitwise OR, Boolean logical OR
Conditional AND	x && y	Evaluates y only if x is true

CATEGORY	EXPRESSION	DESCRIPTION
Conditional OR	x || y	Evaluates y only if x is false
Null coalescing	x ?? y	Evaluates to y if x is null, to x otherwise
Conditional	x ? y : z	Evaluates to y if x is true, z if x is false

Assignment and Anonymous Operators

EXPRESSION	DESCRIPTION
=	Assignment
x op= y	Compound assignment. Supports these operators: +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=
(T x) => y	Anonymous function (lambda expression)

Associativity

When two or more operators that have the same precedence are present in an expression, they are evaluated based on associativity. Left-associative operators are evaluated in order from left to right. For example, `x * y / z` is evaluated as `(x * y) / z`. Right-associative operators are evaluated in order from right to left. For example, the assignment operator is right associative. If it were not, the following code would result in an error.

```
int a, b, c;
c = 1;
// The following two lines are equivalent.
a = b = c;
a = (b = c);

// The following line, which forces left associativity, causes an error.
//(a = b) = c;
```

As another example the ternary operator (?:) is right associative. Most binary operators are left associative.

Whether the operators in an expression are left associative or right associative, the operands of each expression are evaluated first, from left to right. The following examples illustrate the order of evaluation of operators and operands.

STATEMENT	ORDER OF EVALUATION
`a = b`	a, b, =
`a = b + c`	a, b, c, +, =
`a = b + c * d`	a, b, c, d, *, +, =
`a = b * c + d`	a, b, c, *, d, +, =
`a = b - c + d`	a, b, c, -, d, +, =

STATEMENT	ORDER OF EVALUATION
`a += b -= c`	a, b, c, -=, +=

Adding parentheses

You can change the order imposed by operator precedence and associativity by using parentheses. For example, `2 + 3 * 2` ordinarily evaluates to 8, because multiplicative operators take precedence over additive operators. However, if you write the expression as `(2 + 3) * 2`, the addition is evaluated before the multiplication, and the result is 10. The following examples illustrate the order of evaluation in parenthesized expressions. As in previous examples, the operands are evaluated before the operator is applied.

STATEMENT	ORDER OF EVALUATION
`a = (b + c) * d`	a, b, c, +, d, *, =
`a = b - (c + d)`	a, b, c, d, +, -, =
`a = (b + c) * (d - e)`	a, b, c, +, d, e, -, *, =

Operator overloading

You can change the behavior of operators for custom classes and structs. This process is referred to as *operator overloading*. For more information, see Overloadable Operators and the operator keyword article.

Related sections

For more information, see Operator Keywords and C# Operators.

See also

- C# Programming Guide
- Statements, Expressions, and Operators

Anonymous Functions (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

An anonymous function is an "inline" statement or expression that can be used wherever a delegate type is expected. You can use it to initialize a named delegate or pass it instead of a named delegate type as a method parameter.

There are two kinds of anonymous functions, which are discussed individually in the following topics:

- Lambda Expressions.
- Anonymous Methods

> **NOTE**
>
> Lambda expressions can be bound to expression trees and also to delegates.

The Evolution of Delegates in C#

In C# 1.0, you created an instance of a delegate by explicitly initializing it with a method that was defined elsewhere in the code. C# 2.0 introduced the concept of anonymous methods as a way to write unnamed inline statement blocks that can be executed in a delegate invocation. C# 3.0 introduced lambda expressions, which are similar in concept to anonymous methods but more expressive and concise. These two features are known collectively as *anonymous functions*. In general, applications that target version 3.5 and later of the .NET Framework should use lambda expressions.

The following example demonstrates the evolution of delegate creation from C# 1.0 to C# 3.0:

```
class Test
{
    delegate void TestDelegate(string s);
    static void M(string s)
    {
        Console.WriteLine(s);
    }

    static void Main(string[] args)
    {
        // Original delegate syntax required
        // initialization with a named method.
        TestDelegate testDelA = new TestDelegate(M);

        // C# 2.0: A delegate can be initialized with
        // inline code, called an "anonymous method." This
        // method takes a string as an input parameter.
        TestDelegate testDelB = delegate(string s) { Console.WriteLine(s); };

        // C# 3.0. A delegate can be initialized with
        // a lambda expression. The lambda also takes a string
        // as an input parameter (x). The type of x is inferred by the compiler.
        TestDelegate testDelC = (x) => { Console.WriteLine(x); };

        // Invoke the delegates.
        testDelA("Hello. My name is M and I write lines.");
        testDelB("That's nothing. I'm anonymous and ");
        testDelC("I'm a famous author.");

        // Keep console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
    Hello. My name is M and I write lines.
    That's nothing. I'm anonymous and
    I'm a famous author.
    Press any key to exit.
 */
```

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

- Statements, Expressions, and Operators
- Lambda Expressions
- Delegates
- Expression Trees (C#)

Lambda expressions (C# Programming Guide)

1/23/2019 • 9 minutes to read • Edit Online

A lambda expression is an anonymous function that you can use to create delegates or expression tree types. By using lambda expressions, you can write local functions that can be passed as arguments or returned as the value of function calls. Lambda expressions are particularly helpful for writing LINQ query expressions.

To create a lambda expression, you specify input parameters (if any) on the left side of the lambda operator =>, and you put the expression or statement block on the other side. For example, the lambda expression `x => x * x` specifies a parameter that's named `x` and returns the value of `x` squared. You can assign this expression to a delegate type, as the following example shows:

```
delegate int del(int i);
static void Main(string[] args)
{
    del myDelegate = x => x * x;
    int j = myDelegate(5); //j = 25
}
```

To create an expression tree type:

```
using System.Linq.Expressions;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            Expression<del> myET = x => x * x;
        }
    }
}
```

The `=>` operator has the same precedence as assignment (`=`) and is right associative (see "Associativity" section of the Operators article).

Lambdas are used in method-based LINQ queries as arguments to standard query operator methods such as Where.

When you use method-based syntax to call the Where method in the Enumerable class (as you do in LINQ to Objects and LINQ to XML) the parameter is a delegate type System.Func<T,TResult>. A lambda expression is the most convenient way to create that delegate. When you call the same method in, for example, the System.Linq.Queryable class (as you do in LINQ to SQL) then the parameter type is an System.Linq.Expressions.Expression<Func> where Func is any of the Func delegates with up to sixteen input parameters. Again, a lambda expression is just a very concise way to construct that expression tree. The lambdas allow the `Where` calls to look similar although in fact the type of object created from the lambda is different.

In the previous example, notice that the delegate signature has one implicitly-typed input parameter of type `int`, and returns an `int`. The lambda expression can be converted to a delegate of that type because it also has one input parameter (`x`) and a return value that the compiler can implicitly convert to type `int`. (Type inference is discussed in more detail in the following sections.) When the delegate is invoked by using an input parameter of 5, it returns a result of 25.

Lambdas are not allowed on the left side of the is or as operator.

All restrictions that apply to anonymous methods also apply to lambda expressions. For more information, see Anonymous Methods.

Expression lambdas

A lambda expression with an expression on the right side of the => operator is called an *expression lambda*. Expression lambdas are used extensively in the construction of Expression Trees. An expression lambda returns the result of the expression and takes the following basic form:

```
(input-parameters) => expression
```

The parentheses are optional only if the lambda has one input parameter; otherwise they are required. Two or more input parameters are separated by commas enclosed in parentheses:

```
(x, y) => x == y
```

Sometimes it is difficult or impossible for the compiler to infer the input types. When this occurs, you can specify the types explicitly as shown in the following example:

```
(int x, string s) => s.Length > x
```

Input parameter types must be all explicit or all implicit; otherwise, C# generates a CS0748 compiler error.

Specify zero input parameters with empty parentheses:

```
() => SomeMethod()
```

Note in the previous example that the body of an expression lambda can consist of a method call. However, if you are creating expression trees that are evaluated outside of the .NET Framework, such as in SQL Server, you should not use method calls in lambda expressions. The methods will have no meaning outside the context of the .NET common language runtime.

Statement lambdas

A statement lambda resembles an expression lambda except that the statement(s) is enclosed in braces:

(input-parameters) => { statement; }

The body of a statement lambda can consist of any number of statements; however, in practice there are typically no more than two or three.

```
delegate void TestDelegate(string s);
```

```
TestDelegate del = n => { string s = n + " World";
                          Console.WriteLine(s); };
```

Statement lambdas, like anonymous methods, cannot be used to create expression trees.

Async lambdas

You can easily create lambda expressions and statements that incorporate asynchronous processing by using the async and await keywords. For example, the following Windows Forms example contains an event handler that calls and awaits an async method, `ExampleMethodAsync`.

```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
    }

    private async void button1_Click(object sender, EventArgs e)
    {
        // ExampleMethodAsync returns a Task.
        await ExampleMethodAsync();
        textBox1.Text += "\r\nControl returned to Click event handler.\n";
    }

    async Task ExampleMethodAsync()
    {
        // The following line simulates a task-returning asynchronous process.
        await Task.Delay(1000);
    }
}
```

You can add the same event handler by using an async lambda. To add this handler, add an `async` modifier before the lambda parameter list, as the following example shows.

```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
        button1.Click += async (sender, e) =>
        {
            // ExampleMethodAsync returns a Task.
            await ExampleMethodAsync();
            textBox1.Text += "\nControl returned to Click event handler.\n";
        };
    }

    async Task ExampleMethodAsync()
    {
        // The following line simulates a task-returning asynchronous process.
        await Task.Delay(1000);
    }
}
```

For more information about how to create and use async methods, see Asynchronous Programming with async and await.

Lambdas with the standard query operators

Many standard query operators have an input parameter whose type is one of the Func<T,TResult> family of generic delegates. These delegates use type parameters to define the number and types of input parameters, and the return type of the delegate. `Func` delegates are very useful for encapsulating user-defined expressions that are applied to each element in a set of source data. For example, consider the following delegate type:

```
public delegate TResult Func<TArg0, TResult>(TArg0 arg0)
```

The delegate can be instantiated as `Func<int,bool> myFunc` where `int` is an input parameter and `bool` is the return value. The return value is always specified in the last type parameter. `Func<int, string, bool>` defines a delegate with two input parameters, `int` and `string`, and a return type of `bool`. The following `Func` delegate, when it is invoked, will return true or false to indicate whether the input parameter is equal to 5:

```
Func<int, bool> myFunc = x => x == 5;
bool result = myFunc(4); // returns false of course
```

You can also supply a lambda expression when the argument type is an `Expression<Func>`, for example in the standard query operators that are defined in System.Linq.Queryable. When you specify an `Expression<Func>` argument, the lambda will be compiled to an expression tree.

A standard query operator, the Count method, is shown here:

```
int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
int oddNumbers = numbers.Count(n => n % 2 == 1);
```

The compiler can infer the type of the input parameter, or you can also specify it explicitly. This particular lambda expression counts those integers (`n`) which when divided by two have a remainder of 1.

The following line of code produces a sequence that contains all elements in the `numbers` array that are to the left side of the 9 because that's the first number in the sequence that doesn't meet the condition:

```
var firstNumbersLessThan6 = numbers.TakeWhile(n => n < 6);
```

This example shows how to specify multiple input parameters by enclosing them in parentheses. The method returns all the elements in the numbers array until a number is encountered whose value is less than its position. Do not confuse the lambda operator (`=>`) with the greater than or equal operator (`>=`).

```
var firstSmallNumbers = numbers.TakeWhile((n, index) => n >= index);
```

Type inference in lambdas

When writing lambdas, you often do not have to specify a type for the input parameters because the compiler can infer the type based on the lambda body, the parameter's delegate type, and other factors as described in the C# Language Specification. For most of the standard query operators, the first input is the type of the elements in the source sequence. So if you are querying an `IEnumerable<Customer>`, then the input variable is inferred to be a `Customer` object, which means you have access to its methods and properties:

```
customers.Where(c => c.City == "London");
```

The general rules for lambdas are as follows:

- The lambda must contain the same number of parameters as the delegate type.
- Each input parameter in the lambda must be implicitly convertible to its corresponding delegate parameter.
- The return value of the lambda (if any) must be implicitly convertible to the delegate's return type.

Note that lambda expressions in themselves do not have a type because the common type system has no intrinsic concept of "lambda expression." However, it is sometimes convenient to speak informally of the "type" of a lambda expression. In these cases the type refers to the delegate type or Expression type to which the lambda expression is converted.

Variable scope in lambda expressions

Lambdas can refer to *outer variables* (see Anonymous Methods) that are in scope in the method that defines the lambda function, or in scope in the type that contains the lambda expression. Variables that are captured in this manner are stored for use in the lambda expression even if the variables would otherwise go out of scope and be garbage collected. An outer variable must be definitely assigned before it can be consumed in a lambda expression. The following example demonstrates these rules:

```
delegate bool D();
delegate bool D2(int i);

class Test
{
    D del;
    D2 del2;
    public void TestMethod(int input)
    {
        int j = 0;
        // Initialize the delegates with lambda expressions.
        // Note access to 2 outer variables.
        // del will be invoked within this method.
        del = () => { j = 10;  return j > input; };

        // del2 will be invoked after TestMethod goes out of scope.
        del2 = (x) => {return x == j; };

        // Demonstrate value of j:
        // Output: j = 0
        // The delegate has not been invoked yet.
        Console.WriteLine("j = {0}", j);        // Invoke the delegate.
        bool boolResult = del();

        // Output: j = 10 b = True
        Console.WriteLine("j = {0}. b = {1}", j, boolResult);
    }

    static void Main()
    {
        Test test = new Test();
        test.TestMethod(5);

        // Prove that del2 still has a copy of
        // local variable j from TestMethod.
        bool result = test.del2(10);

        // Output: True
        Console.WriteLine(result);

        Console.ReadKey();
    }
}
```

The following rules apply to variable scope in lambda expressions:

- A variable that is captured will not be garbage-collected until the delegate that references it becomes eligible for garbage collection.
- Variables introduced within a lambda expression are not visible in the outer method.
- A lambda expression cannot directly capture an `in`, `ref`, or `out` parameter from an enclosing method.
- A return statement in a lambda expression does not cause the enclosing method to return.

- A lambda expression cannot contain a `goto` statement, `break` statement, or `continue` statement that is inside the lambda function if the jump statement’s target is outside the block. It is also an error to have a jump statement outside the lambda function block if the target is inside the block.

C# language specification

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

Featured book chapter

Delegates, Events, and Lambda Expressions in C# 3.0 Cookbook, Third Edition: More than 250 solutions for C# 3.0 programmers

See also

- C# Programming Guide
- LINQ (Language-Integrated Query)
- Anonymous Methods
- is
- Expression Trees
- Visual Studio 2008 C# Samples (see LINQ Sample Queries files and XQuery program)
- Recursive lambda expressions

How to: Use Lambda Expressions in a Query (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

You do not use lambda expressions directly in query syntax, but you do use them in method calls, and query expressions can contain method calls. In fact, some query operations can only be expressed in method syntax. For more information about the difference between query syntax and method syntax, see Query Syntax and Method Syntax in LINQ.

Example

The following example demonstrates how to use a lambda expression in a method-based query by using the Enumerable.Where standard query operator. Note that the Where method in this example has an input parameter of the delegate type Func<TResult> and that delegate takes an integer as input and returns a Boolean. The lambda expression can be converted to that delegate. If this were a LINQ to SQL query that used the Queryable.Where method, the parameter type would be an `Expression<Func<int,bool>>` but the lambda expression would look exactly the same. For more information on the Expression type, see System.Linq.Expressions.Expression.

```
class SimpleLambda
{
    static void Main()
    {

        // Data source.
        int[] scores = { 90, 71, 82, 93, 75, 82 };

        // The call to Count forces iteration of the source
        int highScoreCount = scores.Where(n => n > 80).Count();

        Console.WriteLine("{0} scores are greater than 80", highScoreCount);

        // Outputs: 4 scores are greater than 80
    }
}
```

Example

The following example demonstrates how to use a lambda expression in a method call of a query expression. The lambda is necessary because the Sum standard query operator cannot be invoked by using query syntax.

The query first groups the students according to their grade level, as defined in the `GradeLevel` enum. Then for each group it adds the total scores for each student. This requires two `Sum` operations. The inner `Sum` calculates the total score for each student, and the outer `Sum` keeps a running, combined total for all students in the group.

```
private static void TotalsByGradeLevel()
{
    // This query retrieves the total scores for First Year students, Second Years, and so on.
    // The outer Sum method uses a lambda in order to specify which numbers to add together.
    var categories =
    from student in students
    group student by student.Year into studentGroup
    select new { GradeLevel = studentGroup.Key, TotalScore = studentGroup.Sum(s => s.ExamScores.Sum()) };

    // Execute the query.
    foreach (var cat in categories)
    {
        Console.WriteLine("Key = {0} Sum = {1}", cat.GradeLevel, cat.TotalScore);
    }
}
/*
     Outputs:
     Key = SecondYear Sum = 1014
     Key = ThirdYear Sum = 964
     Key = FirstYear Sum = 1058
     Key = FourthYear Sum = 974
*/
```

Compiling the Code

To run this code, copy and paste the method into the `StudentClass` that is provided in How to: Query a Collection of Objects and call it from the `Main` method.

See also

- Lambda Expressions
- Expression Trees (C#)

How to: Use Lambda Expressions Outside LINQ (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

Lambda expressions are not limited to LINQ queries. You can use them anywhere a delegate value is expected, that is, wherever an anonymous method can be used. The following example shows how to use a lambda expression in a Windows Forms event handler. Notice that the types of the inputs (Object and MouseEventArgs) are inferred by the compiler and do not have to be explicitly given in the lambda input parameters.

Example

```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
        // Use a lambda expression to define an event handler.
       this.Click += (s, e) => { MessageBox.Show(((MouseEventArgs)e).Location.ToString());};
    }
}
```

See also

- Lambda Expressions
- Anonymous Methods
- Language Integrated Query (LINQ))

Anonymous Methods (C# Programming Guide)

1/23/2019 • 3 minutes to read • Edit Online

In versions of C# before 2.0, the only way to declare a delegate was to use named methods. C# 2.0 introduced anonymous methods and in C# 3.0 and later, lambda expressions supersede anonymous methods as the preferred way to write inline code. However, the information about anonymous methods in this topic also applies to lambda expressions. There is one case in which an anonymous method provides functionality not found in lambda expressions. Anonymous methods enable you to omit the parameter list. This means that an anonymous method can be converted to delegates with a variety of signatures. This is not possible with lambda expressions. For more information specifically about lambda expressions, see Lambda Expressions.

Creating anonymous methods is essentially a way to pass a code block as a delegate parameter. Here are two examples:

```
// Create a handler for a click event.
button1.Click += delegate(System.Object o, System.EventArgs e)
                   { System.Windows.Forms.MessageBox.Show("Click!"); };
```

```
// Create a delegate.
delegate void Del(int x);

// Instantiate the delegate using an anonymous method.
Del d = delegate(int k) { /* ... */ };
```

By using anonymous methods, you reduce the coding overhead in instantiating delegates because you do not have to create a separate method.

For example, specifying a code block instead of a delegate can be useful in a situation when having to create a method might seem an unnecessary overhead. A good example would be when you start a new thread. This class creates a thread and also contains the code that the thread executes without creating an additional method for the delegate.

```
void StartThread()
{
    System.Threading.Thread t1 = new System.Threading.Thread
      (delegate()
            {
                System.Console.Write("Hello, ");
                System.Console.WriteLine("World!");
            });
    t1.Start();
}
```

Remarks

The scope of the parameters of an anonymous method is the *anonymous-method-block*.

It is an error to have a jump statement, such as goto, break, or continue, inside the anonymous method block if the target is outside the block. It is also an error to have a jump statement, such as `goto`, `break`, or `continue`, outside the anonymous method block if the target is inside the block.

The local variables and parameters whose scope contains an anonymous method declaration are called *outer* variables of the anonymous method. For example, in the following code segment, `n` is an outer variable:

```
int n = 0;
Del d = delegate() { System.Console.WriteLine("Copy #:{0}", ++n); };
```

A reference to the outer variable `n` is said to be *captured* when the delegate is created. Unlike local variables, the lifetime of a captured variable extends until the delegates that reference the anonymous methods are eligible for garbage collection.

An anonymous method cannot access the in, ref or out parameters of an outer scope.

No unsafe code can be accessed within the *anonymous-method-block*.

Anonymous methods are not allowed on the left side of the is operator.

Example

The following example demonstrates two ways of instantiating a delegate:

- Associating the delegate with an anonymous method.
- Associating the delegate with a named method (`DoWork`).

In each case, a message is displayed when the delegate is invoked.

```
// Declare a delegate.
delegate void Printer(string s);

class TestClass
{
    static void Main()
    {
        // Instantiate the delegate type using an anonymous method.
        Printer p = delegate(string j)
        {
            System.Console.WriteLine(j);
        };

        // Results from the anonymous delegate call.
        p("The delegate using the anonymous method is called.");

        // The delegate instantiation using a named method "DoWork".
        p = DoWork;

        // Results from the old style delegate call.
        p("The delegate using the named method is called.");
    }

    // The method associated with the named delegate.
    static void DoWork(string k)
    {
        System.Console.WriteLine(k);
    }
}
/* Output:
    The delegate using the anonymous method is called.
    The delegate using the named method is called.
*/
```

See also

- C# Reference
- C# Programming Guide
- Delegates
- Lambda Expressions
- Unsafe Code and Pointers
- Methods
- Delegates with Named vs. Anonymous Methods

Overloadable operators (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

C# allows user-defined types to overload operators by defining static member functions using the operator keyword. Not all operators can be overloaded, however, and others have restrictions, as listed in this table:

OPERATORS	OVERLOADABILITY
+, -, !, ~, ++, --, true, false	These unary operators can be overloaded.
+, -, *, /, %, &, |, ^, <<, >>	These binary operators can be overloaded.
==, !=, <, >, <=, >=	The comparison operators can be overloaded (but see the note that follows this table).
&&, ||	The conditional logical operators cannot be overloaded, but they are evaluated using `&` and `|`, which can be overloaded.
[]	The array indexing operator cannot be overloaded, but you can define indexers.
(T)x	The cast operator cannot be overloaded, but you can define new conversion operators (see explicit and implicit).
+=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=	Assignment operators cannot be explicitly overloaded. However, when you overload a binary operator, the corresponding assignment operator, if any, is also implicitly overloaded. For example, `+=` is evaluated using `+`, which can be overloaded.
=, ., ?:, ??, ->, =>, f(x), as, checked, unchecked, default, delegate, is, new, sizeof, typeof	These operators cannot be overloaded.

NOTE

The comparison operators, if overloaded, must be overloaded in pairs; that is, if `==` is overloaded, `!=` must also be overloaded. The reverse is also true, where overloading `!=` requires an overload for `==`. The same is true for comparison operators `<` and `>` and for `<=` and `>=`.

For information about how to overload an operator, see the operator keyword article.

See also

- C# Programming Guide
- Statements, Expressions, and Operators
- Operators
- C# Operators
- Why are overloaded operators always static in C#?

Conversion operators (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

C# enables programmers to declare conversions on classes or structs so that classes or structs can be converted to and/or from other classes or structs, or basic types. Conversions are defined like operators and are named for the type to which they convert. Either the type of the argument to be converted, or the type of the result of the conversion, but not both, must be the containing type.

```
class SampleClass
{
    public static explicit operator SampleClass(int i)
    {
        SampleClass temp = new SampleClass();
        // code to convert from int to SampleClass...

        return temp;
    }
}
```

Conversion operators overview

Conversion operators have the following properties:

- Conversions declared as `implicit` occur automatically when it is required.
- Conversions declared as `explicit` require a cast to be called.
- All conversions must be declared as `static`.

Related sections

For more information:

- Using Conversion Operators
- Casting and Type Conversions
- How to: Implement User-Defined Conversions Between Structs
- explicit
- implicit
- static

See also

- Convert
- C# Programming Guide
- Chained user-defined explicit conversions in C#

Using Conversion Operators (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

You can use `implicit` conversion operators, which are easier to use, or `explicit` conversion operators, which clearly indicate to anyone reading the code that you're converting a type. This topic demonstrates both types of conversion operator.

NOTE

For information about simple type conversions, see How to: Convert a String to a Number, How to: Convert a byte Array to an int, How to: Convert Between Hexadecimal Strings and Numeric Types, or Convert.

Example

This is an example of an explicit conversion operator. This operator converts from the type Byte to a value type called `Digit`. Because not all bytes can be converted to a digit, the conversion is explicit, meaning that a cast must be used, as shown in the `Main` method.

```
struct Digit
{
    byte value;

    public Digit(byte value)  //constructor
    {
        if (value > 9)
        {
            throw new System.ArgumentException();
        }
        this.value = value;
    }

    public static explicit operator Digit(byte b)  // explicit byte to digit conversion operator
    {
        Digit d = new Digit(b);  // explicit conversion

        System.Console.WriteLine("Conversion occurred.");
        return d;
    }
}

class TestExplicitConversion
{
    static void Main()
    {
        try
        {
            byte b = 3;
            Digit d = (Digit)b;  // explicit conversion
        }
        catch (System.Exception e)
        {
            System.Console.WriteLine("{0} Exception caught.", e);
        }
    }
}
// Output: Conversion occurred.
```

Example

This example demonstrates an implicit conversion operator by defining a conversion operator that undoes what the previous example did: it converts from a value class called `Digit` to the integral Byte type. Because any digit can be converted to a Byte, there's no need to force users to be explicit about the conversion.

```
struct Digit
{
    byte value;

    public Digit(byte value)  //constructor
    {
        if (value > 9)
        {
            throw new System.ArgumentException();
        }
        this.value = value;
    }

    public static implicit operator byte(Digit d)  // implicit digit to byte conversion operator
    {
        System.Console.WriteLine("conversion occurred");
        return d.value;  // implicit conversion
    }
}

class TestImplicitConversion
{
    static void Main()
    {
        Digit d = new Digit(3);
        byte b = d;  // implicit conversion -- no cast needed
    }
}
// Output: Conversion occurred.
```

See also

- C# Reference
- C# Programming Guide
- Conversion Operators
- is

How to: Implement User-Defined Conversions Between Structs (C# Programming Guide)

1/23/2019 • 2 minutes to read • Edit Online

This example defines two structs, `RomanNumeral` and `BinaryNumeral`, and demonstrates conversions between them.

Example

```
struct RomanNumeral
{
    private int value;

    public RomanNumeral(int value)  //constructor
    {
        this.value = value;
    }

    static public implicit operator RomanNumeral(int value)
    {
        return new RomanNumeral(value);
    }

    static public implicit operator RomanNumeral(BinaryNumeral binary)
    {
        return new RomanNumeral((int)binary);
    }

    static public explicit operator int(RomanNumeral roman)
    {
        return roman.value;
    }

    static public implicit operator string(RomanNumeral roman)
    {
        return ("Conversion to string is not implemented");
    }
}

struct BinaryNumeral
{
    private int value;

    public BinaryNumeral(int value)  //constructor
    {
        this.value = value;
    }

    static public implicit operator BinaryNumeral(int value)
    {
        return new BinaryNumeral(value);
    }

    static public explicit operator int(BinaryNumeral binary)
    {
        return (binary.value);
    }

    static public implicit operator string(BinaryNumeral binary)
    {
        return ("Conversion to string is not implemented");
    }
}

class TestConversions
{
    static void Main()
    {
        RomanNumeral roman;
        BinaryNumeral binary;

        roman = 10;

        // Perform a conversion from a RomanNumeral to a BinaryNumeral:
        binary = (BinaryNumeral)(int)roman;

        // Perform a conversion from a BinaryNumeral to a RomanNumeral:
        // No cast is required:
        roman = binary;

        System.Console.WriteLine((int)binary);
        System.Console.WriteLine(binary);

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
/* Output:
    10
    Conversion to string is not implemented
*/
```

Robust Programming

- In the previous example, the statement:

  ```
  binary = (BinaryNumeral)(int)roman;
  ```

  performs a conversion from a `RomanNumeral` to a `BinaryNumeral`. Because there is no direct conversion from `RomanNumeral` to `BinaryNumeral`, a cast is used to convert from a `RomanNumeral` to an `int`, and another cast to convert from an `int` to a `BinaryNumeral`.

- Also the statement

  ```
  roman = binary;
  ```

  performs a conversion from a `BinaryNumeral` to a `RomanNumeral`. Because `RomanNumeral` defines an implicit conversion from `BinaryNumeral`, no cast is required.

See also

- C# Reference
- C# Programming Guide
- Conversion Operators

default value expressions (C# programming guide)

1/23/2019 • 4 minutes to read • Edit Online

A default value expression `default(T)` produces the default value of a type `T`. The following table shows which values are produced for various types:

TYPE	DEFAULT VALUE
Any reference type	`null`
Numeric value type	Zero
bool	`false`
char	`'\0'`
enum	The value produced by the expression `(E)0`, where `E` is the enum identifier.
struct	The value produced by setting all value type fields to their default value and all reference type fields to `null`.
Nullable type	An instance for which the HasValue property is `false` and the Value property is undefined.

Default value expressions are particularly useful in generic classes and methods. One issue that arises using generics is how to assign a default value of a parameterized type `T` when you don't know the following in advance:

- Whether `T` is a reference type or a value type.
- If `T` is a value type, whether it's a numeric value or a struct.

Given a variable `t` of a parameterized type `T`, the statement `t = null` is only valid if `T` is a reference type. The assignment `t = 0` only works for numeric value types but not for structs. To solve that, use a default value expression:

```
T t = default(T);
```

The `default(T)` expression is not limited to generic classes and methods. Default value expressions can be used with any managed type. Any of these expressions are valid:

```
var s = default(string);
var d = default(dynamic);
var i = default(int);
var n = default(int?); // n is a Nullable int where HasValue is false.
```

The following example from the `GenericList<T>` class shows how to use the `default(T)` operator in a generic class. For more information, see Introduction to Generics.

```
namespace ConsoleApplication1
namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            // Test with a non-empty list of integers.
            GenericList<int> gll = new GenericList<int>();
            gll.AddNode(5);
            gll.AddNode(4);
            gll.AddNode(3);
            int intVal = gll.GetLast();
            // The following line displays 5.
            System.Console.WriteLine(intVal);

            // Test with an empty list of integers.
            GenericList<int> gll2 = new GenericList<int>();
            intVal = gll2.GetLast();
            // The following line displays 0.
            System.Console.WriteLine(intVal);

            // Test with a non-empty list of strings.
            GenericList<string> gll3 = new GenericList<string>();
            gll3.AddNode("five");
            gll3.AddNode("four");
            string sVal = gll3.GetLast();
            // The following line displays five.
            System.Console.WriteLine(sVal);

            // Test with an empty list of strings.
            GenericList<string> gll4 = new GenericList<string>();
            sVal = gll4.GetLast();
            // The following line displays a blank line.
            System.Console.WriteLine(sVal);
        }
    }

    // T is the type of data stored in a particular instance of GenericList.
    public class GenericList<T>
    {
        private class Node
        {
            // Each node has a reference to the next node in the list.
            public Node Next;
            // Each node holds a value of type T.
            public T Data;
        }

        // The list is initially empty.
        private Node head = null;

        // Add a node at the beginning of the list with t as its data value.
        public void AddNode(T t)
        {
            Node newNode = new Node();
            newNode.Next = head;
            newNode.Data = t;
            head = newNode;
        }

        // The following method returns the data value stored in the last node in
        // the list. If the list is empty, the default value for type T is
        // returned.
        public T GetLast()
        {
            // The value of temp is returned as the value of the method.
            // The following declaration initializes temp to the appropriate
            // default value for type T. The default value is returned if the
            // list is empty.
            T temp = default(T);

            Node current = head;
            while (current != null)
            {
                temp = current.Data;
                current = current.Next;
            }
            return temp;
        }
    }
}
```

default literal and type inference

Beginning with C# 7.1, the `default` literal can be used for default value expressions when the compiler can infer the type of the expression. The `default` literal produces the same value as the equivalent `default(T)` where `T` is the inferred type. This can make code more concise by reducing the redundancy of declaring a type more than once. The `default` literal can be used in any of the following locations:

- variable initializer
- variable assignment
- declaring the default value for an optional parameter
- providing the value for a method call argument
- return statement (or expression in an expression bodied member)

The following example shows many usages of the `default` literal in a default value expression:

```csharp
public struct Point
{
    public double X { get; }
    public double Y { get; }

    public Point(double x, double y)
    {
        X = x;
        Y = y;
    }
}

public class LabeledPoint
{
    public double X { get; private set; }
    public double Y { get; private set; }
    public string Label { get; set; }

    // Providing the value for a default argument:
    public LabeledPoint(double x, double y, string label = default)
    {
        X = x;
        Y = y;
        Label = label;
    }

    public static LabeledPoint MovePoint(LabeledPoint source,
        double xDistance, double yDistance)
    {
        // return a default value:
        if (source == null)
            return default;

        return new LabeledPoint(source.X + xDistance, source.Y + yDistance,
        source.Label);
    }

    public static LabeledPoint FindClosestLocation(IEnumerable<LabeledPoint> sequence,
        Point location)
    {
        // initialize variable:
        LabeledPoint rVal = default;
        double distance = double.MaxValue;

        foreach (var pt in sequence)
        {
            var thisDistance = Math.Sqrt((pt.X - location.X) * (pt.X - location.X) +
                (pt.Y - location.Y) * (pt.Y - location.Y));
            if (thisDistance < distance)
            {
                distance = thisDistance;
                rVal = pt;
            }
        }

        return rVal;
    }

    public static LabeledPoint ClosestToOrigin(IEnumerable<LabeledPoint> sequence)
        // Pass default value of an argument.
        => FindClosestLocation(sequence, default);
}
```

See also

- System.Collections.Generic
- C# Programming Guide
- Generics (C# Programming Guide)
- Generic Methods
- Generics in .NET
- Default values table

Equality Comparisons (C# Programming Guide)

1/23/2019 • 3 minutes to read • Edit Online

It is sometimes necessary to compare two values for equality. In some cases, you are testing for *value equality*, also known as *equivalence*, which means that the values that are contained by the two variables are equal. In other cases, you have to determine whether two variables refer to the same underlying object in memory. This type of equality is called *reference equality*, or *identity*. This topic describes these two kinds of equality and provides links to other topics for more information.

Reference Equality

Reference equality means that two object references refer to the same underlying object. This can occur through simple assignment, as shown in the following example.

```
using System;
class Test
{
    public int Num { get; set; }
    public string Str { get; set; }

    static void Main()
    {
        Test a = new Test() { Num = 1, Str = "Hi" };
        Test b = new Test() { Num = 1, Str = "Hi" };

        bool areEqual = System.Object.ReferenceEquals(a, b);
        // False:
        System.Console.WriteLine("ReferenceEquals(a, b) = {0}", areEqual);

        // Assign b to a.
        b = a;

        // Repeat calls with different results.
        areEqual = System.Object.ReferenceEquals(a, b);
        // True:
        System.Console.WriteLine("ReferenceEquals(a, b) = {0}", areEqual);

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

In this code, two objects are created, but after the assignment statement, both references refer to the same object. Therefore they have reference equality. Use the ReferenceEquals method to determine whether two references refer to the same object.

The concept of reference equality applies only to reference types. Value type objects cannot have reference equality because when an instance of a value type is assigned to a variable, a copy of the value is made. Therefore you can never have two unboxed structs that refer to the same location in memory. Furthermore, if you use ReferenceEquals to compare two value types, the result will always be `false`, even if the values that are contained in the objects are all identical. This is because each variable is boxed into a separate object instance. For more information, see How to: Test for Reference Equality (Identity).

Value Equality

Value equality means that two objects contain the same value or values. For primitive value types such as int or bool, tests for value equality are straightforward. You can use the == operator, as shown in the following example.

```
int a = GetOriginalValue();
int b = GetCurrentValue();

// Test for value equality.
if( b == a)
{
    // The two integers are equal.
}
```

For most other types, testing for value equality is more complex because it requires that you understand how the type defines it. For classes and structs that have multiple fields or properties, value equality is often defined to mean that all fields or properties have the same value. For example, two `Point` objects might be defined to be equivalent if pointA.X is equal to pointB.X and pointA.Y is equal to pointB.Y.

However, there is no requirement that equivalence be based on all the fields in a type. It can be based on a subset. When you compare types that you do not own, you should make sure to understand specifically how equivalence is defined for that type. For more information about how to define value equality in your own classes and structs, see How to: Define Value Equality for a Type.

Value Equality for Floating Point Values

Equality comparisons of floating point values (double and float) are problematic because of the imprecision of floating point arithmetic on binary computers. For more information, see the remarks in the topic System.Double.

Related Topics

TITLE	DESCRIPTION
How to: Test for Reference Equality (Identity)	Describes how to determine whether two variables have reference equality.
How to: Define Value Equality for a Type	Describes how to provide a custom definition of value equality for a type.
C# Programming Guide	Provides links to detailed information about important C# language features and features that are available to C# through the .NET Framework.
Types	Provides information about the C# type system and links to additional information.

See also

- C# Programming Guide

How to: Define Value Equality for a Type (C# Programming Guide)

1/23/2019 • 8 minutes to read • Edit Online

When you define a class or struct, you decide whether it makes sense to create a custom definition of value equality (or equivalence) for the type. Typically, you implement value equality when objects of the type are expected to be added to a collection of some sort, or when their primary purpose is to store a set of fields or properties. You can base your definition of value equality on a comparison of all the fields and properties in the type, or you can base the definition on a subset. But in either case, and in both classes and structs, your implementation should follow the five guarantees of equivalence:

1. `x.Equals(x)` returns `true`. This is called the reflexive property.
2. `x.Equals(y)` returns the same value as `y.Equals(x)`. This is called the symmetric property.
3. if `(x.Equals(y) && y.Equals(z))` returns `true`, then `x.Equals(z)` returns `true`. This is called the transitive property.
4. Successive invocations of `x.Equals(y)` return the same value as long as the objects referenced by x and y are not modified.
5. `x.Equals(null)` returns `false`. However, `null.Equals(null)` throws an exception; it does not obey rule number two above.

Any struct that you define already has a default implementation of value equality that it inherits from the System.ValueType override of the Object.Equals(Object) method. This implementation uses reflection to examine all the fields and properties in the type. Although this implementation produces correct results, it is relatively slow compared to a custom implementation that you write specifically for the type.

The implementation details for value equality are different for classes and structs. However, both classes and structs require the same basic steps for implementing equality:

1. Override the virtual Object.Equals(Object) method. In most cases, your implementation of `bool Equals( object obj )` should just call into the type-specific `Equals` method that is the implementation of the System.IEquatable<T> interface. (See step 2.)
2. Implement the System.IEquatable<T> interface by providing a type-specific `Equals` method. This is where the actual equivalence comparison is performed. For example, you might decide to define equality by comparing only one or two fields in your type. Do not throw exceptions from `Equals`. For classes only: This method should examine only fields that are declared in the class. It should call `base.Equals` to examine fields that are in the base class. (Do not do this if the type inherits directly from Object, because the Object implementation of Object.Equals(Object) performs a reference equality check.)
3. Optional but recommended: Overload the == and != operators.
4. Override Object.GetHashCode so that two objects that have value equality produce the same hash code.
5. Optional: To support definitions for "greater than" or "less than," implement the IComparable<T> interface for your type, and also overload the <= and >= operators.

The first example that follows shows a class implementation. The second example shows a struct implementation.

Example

The following example shows how to implement value equality in a class (reference type).

```
namespace ValueEquality
{
    using System;
    class TwoDPoint : IEquatable<TwoDPoint>
    {
        // Readonly auto-implemented properties.
        public int X { get; private set; }
        public int Y { get; private set; }

        // Set the properties in the constructor.
        public TwoDPoint(int x, int y)
        {
            if ((x < 1) || (x > 2000) || (y < 1) || (y > 2000))
            {
                throw new System.ArgumentException("Point must be in range 1 - 2000");
            }
            this.X = x;
            this.Y = y;
        }

        public override bool Equals(object obj)
        {
            return this.Equals(obj as TwoDPoint);
        }

        public bool Equals(TwoDPoint p)
        {
            // If parameter is null, return false.
            if (Object.ReferenceEquals(p, null))
            {
                return false;
            }

            // Optimization for a common success case.
            if (Object.ReferenceEquals(this, p))
            {
                return true;
            }

            // If run-time types are not exactly the same, return false.
            if (this.GetType() != p.GetType())
            {
                return false;
            }

            // Return true if the fields match.
            // Note that the base class is not invoked because it is
            // System.Object, which defines Equals as reference equality.
            return (X == p.X) && (Y == p.Y);
        }

        public override int GetHashCode()
        {
            return X * 0x00010000 + Y;
        }

        public static bool operator ==(TwoDPoint lhs, TwoDPoint rhs)
        {
            // Check for null on left side.
            if (Object.ReferenceEquals(lhs, null))
            {
```

```
            if (Object.ReferenceEquals(rhs, null))
            {
                // null == null = true.
                return true;
            }

            // Only the left side is null.
            return false;
        }
        // Equals handles case of null on right side.
        return lhs.Equals(rhs);
    }

    public static bool operator !=(TwoDPoint lhs, TwoDPoint rhs)
    {
        return !(lhs == rhs);
    }
}

// For the sake of simplicity, assume a ThreeDPoint IS a TwoDPoint.
class ThreeDPoint : TwoDPoint, IEquatable<ThreeDPoint>
{
    public int Z { get; private set; }

    public ThreeDPoint(int x, int y, int z)
        : base(x, y)
    {
        if ((z < 1) || (z > 2000))
        {
            throw new System.ArgumentException("Point must be in range 1 - 2000");
        }
        this.Z = z;
    }

    public override bool Equals(object obj)
    {
        return this.Equals(obj as ThreeDPoint);
    }

    public bool Equals(ThreeDPoint p)
    {
        // If parameter is null, return false.
        if (Object.ReferenceEquals(p, null))
        {
            return false;
        }

        // Optimization for a common success case.
        if (Object.ReferenceEquals(this, p))
        {
            return true;
        }

        // Check properties that this class declares.
        if (Z == p.Z)
        {
            // Let base class check its own fields
            // and do the run-time type comparison.
            return base.Equals((TwoDPoint)p);
        }
        else
        {
            return false;
        }
    }

    public override int GetHashCode()
    {
        return (X * 0x100000) + (Y * 0x1000) + Z;
    }

    public static bool operator ==(ThreeDPoint lhs, ThreeDPoint rhs)
    {
        // Check for null.
        if (Object.ReferenceEquals(lhs, null))
        {
            if (Object.ReferenceEquals(rhs, null))
            {
                // null == null = true.
                return true;
            }

            // Only the left side is null.
            return false;
        }
        // Equals handles the case of null on right side.
        return lhs.Equals(rhs);
    }

    public static bool operator !=(ThreeDPoint lhs, ThreeDPoint rhs)
    {
        return !(lhs == rhs);
    }
}

class Program
{
    static void Main(string[] args)
    {
        ThreeDPoint pointA = new ThreeDPoint(3, 4, 5);
        ThreeDPoint pointB = new ThreeDPoint(3, 4, 5);
        ThreeDPoint pointC = null;
        int i = 5;

        Console.WriteLine("pointA.Equals(pointB) = {0}", pointA.Equals(pointB));
        Console.WriteLine("pointA == pointB = {0}", pointA == pointB);
        Console.WriteLine("null comparison = {0}", pointA.Equals(pointC));
        Console.WriteLine("Compare to some other type = {0}", pointA.Equals(i));

        TwoDPoint pointD = null;
        TwoDPoint pointE = null;



        Console.WriteLine("Two null TwoDPoints are equal: {0}", pointD == pointE);

        pointE = new TwoDPoint(3, 4);
        Console.WriteLine("(pointE == pointA) = {0}", pointE == pointA);
        Console.WriteLine("(pointA == pointE) = {0}", pointA == pointE);
        Console.WriteLine("(pointA != pointE) = {0}", pointA != pointE);

        System.Collections.ArrayList list = new System.Collections.ArrayList();
        list.Add(new ThreeDPoint(3, 4, 5));
        Console.WriteLine("pointE.Equals(list[0]): {0}", pointE.Equals(list[0]));

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

/* Output:
    pointA.Equals(pointB) = True
    pointA == pointB = True
    null comparison = False
    Compare to some other type = False
    Two null TwoDPoints are equal: True
    (pointE == pointA) = False
```

```
            (pointA == pointE) = False
            (pointA != pointE) = True
            pointE.Equals(list[0]): False
        */
    }
```

On classes (reference types), the default implementation of both Object.Equals(Object) methods performs a reference equality comparison, not a value equality check. When an implementer overrides the virtual method, the purpose is to give it value equality semantics.

The `==` and `!=` operators can be used with classes even if the class does not overload them. However, the default behavior is to perform a reference equality check. In a class, if you overload the `Equals` method, you should overload the `==` and `!=` operators, but it is not required.

Example

The following example shows how to implement value equality in a struct (value type):

```
    struct TwoDPoint : IEquatable<TwoDPoint>
    {
        // Read/write auto-implemented properties.
        public int X { get; private set; }
        public int Y { get; private set; }

        public TwoDPoint(int x, int y)
            : this()
        {
            X = x;
            Y = x;
        }

        public override bool Equals(object obj)
        {
            if (obj is TwoDPoint)
            {
                return this.Equals((TwoDPoint)obj);
            }
            return false;
        }

        public bool Equals(TwoDPoint p)
        {
            return (X == p.X) && (Y == p.Y);
        }

        public override int GetHashCode()
        {
            return X ^ Y;
        }

        public static bool operator ==(TwoDPoint lhs, TwoDPoint rhs)
        {
            return lhs.Equals(rhs);
        }

        public static bool operator !=(TwoDPoint lhs, TwoDPoint rhs)
        {
            return !(lhs.Equals(rhs));
        }
    }


    class Program
    {
        static void Main(string[] args)
        static void Main(string[] args)
        {
            TwoDPoint pointA = new TwoDPoint(3, 4);
            TwoDPoint pointB = new TwoDPoint(3, 4);
            int i = 5;

            // Compare using virtual Equals, static Equals, and == and != operators.
            // True:
            Console.WriteLine("pointA.Equals(pointB) = {0}", pointA.Equals(pointB));
            // True:
            Console.WriteLine("pointA == pointB = {0}", pointA == pointB);
            // True:
            Console.WriteLine("Object.Equals(pointA, pointB) = {0}", Object.Equals(pointA, pointB));
            // False:
            Console.WriteLine("pointA.Equals(null) = {0}", pointA.Equals(null));
            // False:
            Console.WriteLine("(pointA == null) = {0}", pointA == null);
            // True:
            Console.WriteLine("(pointA != null) = {0}", pointA != null);
            // False:
            Console.WriteLine("pointA.Equals(i) = {0}", pointA.Equals(i));
            // CS0019:
            // Console.WriteLine("pointA == i = {0}", pointA == i);

            // Compare unboxed to boxed.
            System.Collections.ArrayList list = new System.Collections.ArrayList();
            list.Add(new TwoDPoint(3, 4));
            // True:
            Console.WriteLine("pointE.Equals(list[0]): {0}", pointA.Equals(list[0]));


            // Compare nullable to nullable and to non-nullable.
            TwoDPoint? pointC = null;
            TwoDPoint? pointD = null;
            // False:
            Console.WriteLine("pointA == (pointC = null) = {0}", pointA == pointC);
            // True:
            Console.WriteLine("pointC == pointD = {0}", pointC == pointD);

            TwoDPoint temp = new TwoDPoint(3, 4);
            pointC = temp;
            // True:
            Console.WriteLine("pointA == (pointC = 3,4) = {0}", pointA == pointC);

            pointD = temp;
            // True:
            Console.WriteLine("pointD == (pointC = 3,4) = {0}", pointD == pointC);

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

    /* Output:
        pointA.Equals(pointB) = True
        pointA == pointB = True
        Object.Equals(pointA, pointB) = True
        pointA.Equals(null) = False
        (pointA == null) = False
        (pointA != null) = True
        pointA.Equals(i) = False
        pointE.Equals(list[0]): True
        pointA == (pointC = null) = False
        pointC == pointD = True
        pointA == (pointC = 3,4) = True
        pointD == (pointC = 3,4) = True
    */
}
```

For structs, the default implementation of Object.Equals(Object) (which is the overridden version in System.ValueType) performs a value equality check by using reflection to compare the values of every field in the type. When an implementer overrides the virtual `Equals` method in a struct, the purpose is to provide a more efficient means of performing the value equality check and optionally to base the comparison on some subset of the struct's field or properties.

The == and != operators cannot operate on a struct unless the struct explicitly overloads them.

See also

- Equality Comparisons
- C# Programming Guide

How to: Test for Reference Equality (Identity) (C# Programming Guide)

1/23/2019 • 3 minutes to read • Edit Online

You do not have to implement any custom logic to support reference equality comparisons in your types. This functionality is provided for all types by the static Object.ReferenceEquals method.

The following example shows how to determine whether two variables have *reference equality*, which means that they refer to the same object in memory.

The example also shows why Object.ReferenceEquals always returns `false` for value types and why you should not use ReferenceEquals to determine string equality.

Example

```
namespace TestReferenceEquality
{
    struct TestStruct
    {
        public int Num { get; private set; }
        public string Name { get; private set; }

        public TestStruct(int i, string s) : this()
        {
            Num = i;
            Name = s;
        }
    }

    class TestClass
    {
        public int Num { get; set; }
        public string Name { get; set; }
    }

    class Program
    {
        static void Main()
        {
            // Demonstrate reference equality with reference types.
            #region ReferenceTypes

            // Create two reference type instances that have identical values.
            TestClass tcA = new TestClass() { Num = 1, Name = "New TestClass" };
            TestClass tcB = new TestClass() { Num = 1, Name = "New TestClass" };

            Console.WriteLine("ReferenceEquals(tcA, tcB) = {0}",
                                Object.ReferenceEquals(tcA, tcB)); // false

            // After assignment, tcB and tcA refer to the same object.
            // They now have reference equality.
            tcB = tcA;
            Console.WriteLine("After asignment: ReferenceEquals(tcA, tcB) = {0}",
                                Object.ReferenceEquals(tcA, tcB)); // true

            // Changes made to tcA are reflected in tcB. Therefore, objects
            // that have reference equality also have value equality.
            tcA.Num = 42;
            tcA.Name = "TestClass 42";
            Console.WriteLine("tcB.Name = {0} tcB.Num: {1}", tcB.Name, tcB.Num);
            #endregion

            // Demonstrate that two value type instances never have reference equality.
            #region ValueTypes

            TestStruct tsC = new TestStruct( 1, "TestStruct 1");

            // Value types are copied on assignment. tsD and tsC have
            // the same values but are not the same object.
            TestStruct tsD = tsC;
            Console.WriteLine("After asignment: ReferenceEquals(tsC, tsD) = {0}",
                                Object.ReferenceEquals(tsC, tsD)); // false
            #endregion

            #region stringRefEquality
            // Constant strings within the same assembly are always interned by the runtime.
            // This means they are stored in the same location in memory. Therefore,
            // the two strings have reference equality although no assignment takes place.
            string strA = "Hello world!";
            string strB = "Hello world!";
            Console.WriteLine("ReferenceEquals(strA, strB) = {0}",
                             Object.ReferenceEquals(strA, strB)); // true

            // After a new string is assigned to strA, strA and strB
            // are no longer interned and no longer have reference equality.
            strA = "Goodbye world!";
            Console.WriteLine("strA = \"{0}\" strB = \"{1}\"", strA, strB);

            Console.WriteLine("After strA changes, ReferenceEquals(strA, strB) = {0}",
                            Object.ReferenceEquals(strA, strB)); // false

            // A string that is created at runtime cannot be interned.
            StringBuilder sb = new StringBuilder("Hello world!");
            string stringC = sb.ToString();
            // False:
            Console.WriteLine("ReferenceEquals(stringC, strB) = {0}",
                            Object.ReferenceEquals(stringC, strB));

            // The string class overloads the == operator to perform an equality comparison.
            Console.WriteLine("stringC == strB = {0}", stringC == strB); // true

            #endregion

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

/* Output:
    ReferenceEquals(tcA, tcB) = False
    After asignment: ReferenceEquals(tcA, tcB) = True
    tcB.Name = TestClass 42 tcB.Num: 42
    After asignment: ReferenceEquals(tsC, tsD) = False
    ReferenceEquals(strA, strB) = True
    strA = "Goodbye world!" strB = "Hello world!"
    After strA changes, ReferenceEquals(strA, strB) = False
*/
```

The implementation of `Equals` in the System.Object universal base class also performs a reference equality check, but it is best not to use this because, if a class happens to override the method, the results might not be what you expect. The same is true for the `==` and `!=` operators. When they are operating on reference types, the default behavior of `==` and `!=` is to perform a reference equality check. However, derived classes can overload the operator to perform a value equality check. To minimize the potential for error, it is best to always use

ReferenceEquals when you have to determine whether two objects have reference equality.

Constant strings within the same assembly are always interned by the runtime. That is, only one instance of each unique literal string is maintained. However, the runtime does not guarantee that strings created at runtime are interned, nor does it guarantee that two equal constant strings in different assemblies are interned.

See also

- Equality Comparisons