C# Quick Reference Guide

Variables

```
class MainClass {

  // Value type
  enum myEnum { Zero, One };

  static void Main () {

    // Value types
    bool myBool = true; // True or false
    byte myByte = 255; // 0 to 255
    char myChar = 'a'; // U +0000 to U +ffff
    decimal myDecimal = 1m; // 128-bit decimal values
    double myDouble = 1d; // 64-bit double-precision
    float myFloat = 1f; // 32-bit single-precision
    int myInt = 1; // -2,147,483,648 to 2,147,483,647
    long myLong = 1L; // 64-bit signed integer type
    sbyte mySbyte = 1; // -128 to 127
    short myShort = 1; // -32,768 to 32,767
    uint myUint = 1; // 0 to 4,294,967,295
    ulong myUlong = 1; // 0 to 18,446,744,073,709,551,615
    ushort myUshort = 1; // 0 to 65,535

    // Reference types
    dynamic myDynamic = 1; // Bypass compile-time type checking
    object myObject = new myClass();
    string myString = "test";

    // Pointer types
    /*
    unsafe {
      int* myIntVariable; // Int variable address
    }
    */
  }

  // Reference type
  class myClass { };
  interface myInterface { };
  delegate void myDelegate();
}
```

Type Conversion

```
Convert.ToBoolean(x);    // Converts a type to a Boolean value
Convert.ToByte(x);       // Converts a type to a byte
Convert.ToChar(x);       // Converts a type to a single Unicode
character
Convert.ToDateTime(x);   // Converts a type (integer or string type)
to date-time structures
Convert.ToDecimal(x);    // Converts a floating point or integer type
to a decimal type
Convert.ToDouble(x);     // Converts a type to a double type
Convert.ToInt16(x);      // Converts a type to a 16-bit integer
Convert.ToInt32(x);      // Converts a type to a 32-bit integer
Convert.ToInt64(x);      // Converts a type to a 64-bit integer
Convert.ToSbyte(x);      // Converts a type to a signed byte type
Convert.ToSingle(x);     // Converts a type to a small floating point
number
Convert.ToString(x);     // Converts a type to a string
Convert.ToType(x);       // Converts a type to a specified type
Convert.ToUInt16(x);     // Converts a type to an unsigned int type
Convert.ToUInt32(x);     // Converts a type to an unsigned long type
Convert.ToUInt64(x);     // Converts a type to an unsigned big
integer
```

- As

```
SomeType x = y as SomeType;
if (x != null)
{
  // Do something
}
```

Sizeof

```
// Constant value 4:
int intSize = sizeof(int);
```

Operators

- Arithmetic Operators

```
x + y   // Adds two operands
x - y   // Subtracts second operand from the first
x * y   // Multiplies both operands
x / y   // Divides numerator by de-numerator
x % y   // Modulus Operator and remainder of after an integer
division
x++     // Increment operator increases integer value by one
x--     // Decrement operator decreases integer value by one
```

- Relational Operators

```
(x == y)   // Checks if the values of two operands are equal
(x != y)   // Checks if the values of two operands are equal or not
(x > y)    // Checks if the value of left operand is greater than the
value of right operand
(x < y)    // Checks if the value of left operand is less than the
value of right operand
(x >= y)   // Checks if the value of left operand is greater than or
equal to the value of right operand
(x <= y)   // Checks if the value of left operand is less than or
equal to the value of right operand
```

- Logical Operators

```
(x && y)   // Logical AND operator
(x || y)   // Logical OR Operator
!(x || y)  // Logical NOT Operator
```

- Overload a built-in operator

[Run example] [Oficial docs]

```
using System;

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);
    }

    // user-defined conversion from Fraction to double
    public static implicit operator double(Fraction f)
    {
        return (double)f.num / f.den;
    }

    static void Main () {
        Fraction x = new Fraction(1, 2);
        Fraction y = new Fraction(3, 4);

        Console.WriteLine ((double)x + y);
    }
}
```

Decision Making

- If statement
  [Run example]

```
if(boolean_expression)
{
   /* boolean expression is true */
}
```

- If else statements
  [Run example]

```
if(boolean_expression)
{
   /* boolean expression is true */
}
else
{
   /* expression is false */
}
```

- If, else if, else statements
  [Run example]

```
if(boolean_expression1)
{
   /* boolean expression 1 is true */
}
else if (boolean_expression2)
{
   /* boolean expression 2 is true */
}
else
{
   /* expression 1 and 2 are false */
}
```

- Nested if statements
  [Run example]

```
if( boolean_expression1)
{
   /* boolean expression 1 is true */
   if(boolean_expression2)
   {
      /* expression 2 is true */
   }
}
```

- Switch statement
  [Run example]

```
switch(place)
{
   case 1  :
      Console.WriteLine("First!");
      break;
   case 2  :
      Console.WriteLine("Second!");
      break;
   default : /* Optional */
      Console.WriteLine("Invalid place!");
      break;
}
```

Loops

- While loop

```
while(condition)
{
   Console.WriteLine("Hello!");
}
```

- For loop

```
for (int x = 0; x < 10; x++)
{
   Console.WriteLine($"value of x: {x}");
}
```

- Do...while loop

```
int x = 0;

do
{
   Console.WriteLine($"value of x: {x}");
   x++;
}
while (x < 10);
```

- Foreach, in

```
ArrayList numbers = new ArrayList();
numbers.Add(1);
numbers.Add(2);
numbers.Add(3);

Console.WriteLine($"Count: {numbers.Count}");

foreach (int number in numbers)
{
   Console.Write(number + " ");
}
```

- Break Statement
  [Run example]

```
int x = 0;

while (x < 10)
{
   Console.WriteLine($"value of x: {x}");
   x++;
   if (x > 5)
   {
      /* terminate the loop using break statement */
      break;
   }
}
```

- Continue Statement

```
int x = 0;

do
{
   if (x == 5)
   {
      x++;
      /* skips printing 6 */
      continue;
   }
   x++;
   Console.WriteLine($"value of x: {x}");
}
while (x < 10);
```

Methods

```
using System;
namespace CalculatorApplication
{
   class Calculator
   {
      public int Sum(int x, int y)
      {
         return x + y;
      }
      static void Main(string[] args)
      {
         var result = Sum(2, 2);
         Console.WriteLine("result: {0}", result);
      }
   }
}
```

Nullables

```
int? x = null;
int? y = 2;

int? variableName = null;
double? variableName = null;
bool? variableName = null;
int?[] arr = new int?[10];

var z = x ?? 10; // Null Coalescing Operator
```

Arrays

```
double[] balance = new double[10]; // Initializing an Array
double[] marks = { 1, 2, 3 }; // Assigning Values to an Array

balance[0] = 10;

var first = balance[0];
```

Strings

```
string name = "John doe";
Console.WriteLine("Name: {0}", name);
```

Structures

```
struct Books
{
   public string title;
   public string author;
   public string subject;
   public int book_id;
};

Books book1;   /* Declare Book1 of type Book */
book1.title = "Csharp Programming";
Console.WriteLine( "Book 1 title : {0}", Book1.title);

Books book2 = new Books() {title = "Hamlet", author = "William
Shakespeare", subject = "tragedy", book_id = 1};
Console.WriteLine( "Book 1 title : {0}", Book2.title);
```

Enums

```
enum Days { Sun, Mon, tue, Wed, thu, Fri, Sat };

Console.WriteLine("Monday: {0}", (int)Days.Mon);
```

Classes

```
class Person
{
    public string Name { get; set; }
    public int Age { get; set; }

    public Person(int age, string name)
    {
        Age = age;
        Name = name;
    }

    public int Talk()
    {
        return "Hello!";
    }
}

public class Application
{
    static void Main()
    {
      Person person = new Person("Bill", 42);
      Console.WriteLine("person Name = {0} Age = {1}",
                        person.Name, person.Age);
    }
}
```

Polymorphism

```
public class Shape

{
    // A few example members
    public int X { get; private set; }
    public int Y { get; private set; }
    public int Height { get; set; }
    public int Width { get; set; }

    // Virtual method
    public virtual void Draw()
    {
        Console.WriteLine("Performing base class drawing tasks");
    }
}

class Circle : Shape
{
    public override void Draw()
    {
        // Code to draw a circle...
        Console.WriteLine("Drawing a circle");
        base.Draw();
    }
}

class Rectangle : Shape
{
    public override void Draw()
    {
        // Code to draw a rectangle...
        Console.WriteLine("Drawing a rectangle");
        base.Draw();
    }
}
```

Inheritance

```
class Shape
{
   public void setWidth(int w)
   {
      width = w;
   }
   public void setHeight(int h)
   {
      height = h;
   }
   protected int width;
   protected int height;
}

// Derived class
class Rectangle: Shape
{
   public int getArea()
   {
      return (width * height);
   }
}
```

Abstract

```
abstract class BaseClass
{
    protected int _x = 100;
    protected int _y = 150;
    public abstract void AbstractMethod();
}

class DerivedClass : BaseClass
{
    public override void AbstractMethod()
    {
        _x++;
        _y++;
    }
}
```

Interface

```
public interface IPerson
{
    // interface members
    public int Talk();
}

class Person : IPerson
{
    public string Name { get; set; }
    public int Age { get; set; }

    public Person(int age, string name)
    {
        Age = age;
        Name = name;
    }

    public int Talk()
    {
        return "Hello!";
    }
}
```

Exception Handling

```
try
{
   // statements causing exception
}
catch( ExceptionName e1 )
{
   // error handling code
}
catch( ExceptionName e2 )
{
   // error handling code
}
catch( ExceptionName eN )
{
   // error handling code
}
finally
{
   // statements to be executed
}
```

- Exception filters [C# 6.0]

```
try
  {
    throw new Exception("Exception 1");
  }
  catch(Exception ex) when(ex.Message == "Exception 2")
  {
    Console.WriteLine("caught Exception 2");
  }
  catch(Exception ex) when(ex.Message == "Exception 1")
  {
    Console.WriteLine("caught Exception 1");
  }
```

Checked and Unchecked

- Checked

```
// The following statements are checked by default at compile time.
They do not
// compile.
int1 = 2147483647 + 10;
int1 = ConstantMax + 10;

// If the previous sum is attempted in a checked environment, an
// OverflowException error is raised.

// Checked expression.
Console.WriteLine(checked(2147483647 + ten));
```

- Unchecked

```
// The following statements compile and run.
unchecked
{
   int1 = 2147483647 + 10;
}
```

Delegate

```
// 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 multByTwo = Double;

        // Invoke delegate multByTwo:
        Console.WriteLine(multByTwo(4.5)); // 9

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

        Console.WriteLine(square(5)); // 25

        // Instantiate delegate with lambda expression
        MathAction cube = s => s * s * s;

        Console.WriteLine(cube(4.375)); // 83.740234375
    }
}
```

Event

```
public class SampleEventArgs
{
    public SampleEventArgs(string s) { Text = s; }
    public String Text {get; private set;} // 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"));
    }
}
```

Explicit

```
// Must be defined inside a class called Fahrenheit:
public static explicit operator Celsius(Fahrenheit fahr)
{
    return new Celsius((5.0f / 9.0f) * (fahr.degrees - 32));
}

Fahrenheit fahr = new Fahrenheit(100.0f);
Console.Write("{0} Fahrenheit", fahr.Degrees);
Celsius c = (Celsius)fahr;
```

Extern

```
// Used to declare a method that is implemented externally
[DllImport("avifil32.dll")]
private static extern void AVIFileInit();
```

Fixed

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

// Fixed prevents the garbage collector from relocating a movable
variable
// The fixed statement is only permitted in an unsafe context
unsafe static void TestMethod()
{
    // 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;
    }
}
```

Goto

```
// Transfers the program control directly to a labeled statement
switch (option)
{
   case 1:
       Console.WriteLine("Case 1.");
       break;
   case 2:
       Console.WriteLine("Case 2.");
       goto case 1;
   case 3:
       Console.WriteLine("Case 3.");
       goto case 1;
   default:
       Console.WriteLine("Invalid selection.");
       break;
}

for (int i = 0; i < 10; i++)
{
    if (i = 5)
    {
        goto Found;
    }
}

Found:
   Console.WriteLine("Found 5!");
```

Implicit

```
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);
    }
}

// Use
// Implicit "double" operator
double num = dig;

// Implicit "Digit" operator
Digit dig2 = 12;
```

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
```

Is

```
if (obj is Person) { // Checks if an object is compatible with a
given type
   // Do something if obj is a Person.
}
```

Lock

```
class Account
{
    decimal balance;
    private Object thisLock = new Object();

    public void Withdraw(decimal amount)
    {
        lock (thisLock) // Ensures that one thread does not enter a
critical section of code while another thread is in the critical
section.
        {
            if (amount > balance)
            {
                throw new Exception("Insufficient funds");
            }
            balance -= amount;
        }
    }
}
```

Override

```
abstract class ShapesClass
{
    abstract public int Area(); // Abstract method to override
}
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() // Overridden implementation
    {
        return side * side;
    }
}
```

Readonly

```
class Age
{
    readonly int _year;
    Age(int year)
    {
        _year = year;
    }
    void ChangeYear()
    {
        //_year = 1967; // Compile error if uncommented.
    }
}
```

Method Parameters

- Params

```
public static void UseParams(params object[] list) // Variable number
of arguments.
{
  for (int i = 0; i < list.Length; i++)
  {
      Console.Write(list[i] + " ");
  }
}

UseParams(1, 'a', "test");
```

- Ref

```
class RefExample
{
    static void Method(ref int i)
    {
        i = i + 44;
    }

    static void Main()
    {
        int val = 1;
        Method(ref val);
        Console.WriteLine(val); // 45
    }
}
```

- Out [C# 7.0]
    - Parameter modifier

```
class OutExample
{
   static void Method(out int i)
   {
      i = 44;
   }

   static void Main()
   {
      int value;
      Method(out value);
      Console.WriteLine(value);     // value is now 44
   }
}
```

    - Generic type parameter declarations

```
// 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;
    }
}
```

Sealed

```
class A {}
sealed class B : A {} // No class can inherit from class B

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("C.F"); }

    // Overriding F2 is allowed.
    protected override void F2() { Console.WriteLine("Z.F2"); }
}
```

Stackalloc

```
class Fibonacci
{
    static unsafe void Main() // Unsafe code context
    {
        const int arraySize = 20;
        int* fib = stackalloc int[arraySize]; // Allocate a block of memory on the stack
        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]);
        }

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

Static

```
// Declare a static member, which belongs to the type itself rather than to a specific object.
static class CompanyEmployee
{
    public static void DoSomething() { /*...*/ }
    public static void DoSomethingElse() { /*...*/  }
}

CompanyEmployee.DoSomething();
CompanyEmployee.DoSomethingElse();

class Employee
{
   public static string name;
}

Employee.name
```

This

```
// Use to qualify members hidden by similar names
public Employee(string name)
{
    this.name = name;
}

// Use to pass an object as a parameter to other methods
CalcTax(this);

// Use to declare indexers
public int this[int param]
{
    get { return array[param]; }
    set { array[param] = value; }
}
```

Typeof

```
System.Type type = typeof(int); // System.Int32
```

Unsafe

```
unsafe static void FastCopy(byte[] src, byte[] dst, int count)
{
    // Unsafe context: can use pointers here.
}
```

Using static

```
using static System.Console; // Designates a type whose static
members you can
                             // access without specifying a type
name.

class Program
{
    static void Main()
    {
        WriteLine("Hello world!"); // Without specifying Console
    }
}
```

Virtual

```
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";
            }
        }
    }

}
```

Volatile

```
class VolatileTest
{
    public volatile int i; // Indicates that a field might be modified by multiple
                           // threads that are executing at the same time

    public void Test(int _i)
    {
        i = _i;
    }
}
```

Generics

[C# 2.0]

```
// Declare the generic class.
public class GenericList<T>
{
    void Add(T input) { }
}
class TestGenericList
{
    private class ExampleClass { }
    static void Main()
    {
        // Declare a list of type int.
        GenericList<int> list1 = new GenericList<int>();

        // Declare a list of type string.
        GenericList<string> list2 = new GenericList<string>();

        // Declare a list of type ExampleClass.
        GenericList<ExampleClass> list3 = new
GenericList<ExampleClass>();
    }
}
```

Partial Types

[C# 2.0]

```
// Declare first partial class
public partial class MyClass
{
    int x;
}

// Declare second partial class
public partial class MyClass
{
    int y;
}

// Declare third partial class
public partial class MyClass
{
    public MyClass()
    {
          this.x = 10;
          this.y = 20;
    }
}

// The three partials will generate just one class after compiled
```

Anonymous methods

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

// 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.");

// Output: The delegate using the anonymous method is called.
```

Iterators

[C# 2.0]

```
// Iterator can be used to step through collections such as lists and
arrays
class Department
{
   private List<Employees> _employees;

   public IEnumerator<Employees> GetEnumerator()
   {
      foreach (Employees emp in _employees)
      yield return emp;
   }
}

static void Main(string[] args)
{
   Department dept = new Department("MyDepartment");
   foreach (Employees emp in dept)
   {
      //...
   }
}
```

Getter and setter separate accessibility

[C# 2.0]

```
class Customer
{ // Different accessibility on get and set accessors using accessor-
modifier
   public string Name { get; protected set; }
}
```

Method group conversions

[C# 2.0]

```
// suppose we have a method called RemoveSpaces(string s) and a
delegate called Del
// to assign a method to the delegate:
Del d = RemoveSpaces;
```

Covariance and Contravariance for delegates

[C# 2.0]

```
static object GetObject() { return null; }
static void SetObject(object obj) { }

static string GetString() { return “”; }
static void SetString(string str) { }

// Covariance. A delegate specifies a return type as object,
// but I can assign a method that returns a string.
Func<object> del = GetString;

// Contravariance. A delegate specifies a parameter type as string,
// but I can assign a method that takes an object.
Action<string> del2 = SetObject;
```

Delegate inference

[C# 2.0]

```
//create a delegate instance without the new keyword part
delegate void SomeAction();
SomeAction newStyle = SayHello;
```

Implicitly typed local variables

[C# 3.0]

```
// compiled as an int
var foo = 5;

// compiled as a string
var foo = "Hello";

// compiled as int[]
var foo = new[] { 0, 1, 2 };

// expr is compiled as IEnumerable<Customer> or perhaps
IQueryable<Customer>
var foo =
    from c in customers
    where c.City == "London"
    select c;

// compiled as an anonymous type
var foo = new { Name = "Terry", Age = 34 };

// compiled as List<int>
var foo = new List<int>();
```

Object and collection initializers

[C# 3.0]

```
// Object initializer
class Customer
{
   public string Name { get; set; }
   public int Age { get; set; }
}

Customer foo = new Customer { Name = "Spock", Age = 21 };

// Anonymous object initializer
var bar = new  { Name = "Spock", Age = 21 };

// Collection initializer
List<Customer> foos = new List<Customer>
{
    new Customer { Name = "John", Age = 21 };
    new Customer { Name = "Ringo", Age = 32 };
    new Customer { Name = "Paul", Age = 43 };
};
```

Auto-Implemented properties

[C# 3.0]

```
class Customer
{
    // Auto-Implemented properties for trivial get and set
   public int CustomerID { get; set; }
   public string Name { get; set; }
}
```

Anonymous Types

[C# 3.0]

```
// Anonymous types provide a convenient way to encapsulate a set of
read-only
// properties into a single object without having to explicitly
define a type first

var v = new { Amount = 108, Message = "Hello" };
Console.WriteLine(v.Amount + v.Message);

// Anonymous types typically are used in the select clause of a query
expression
// to return a subset of the properties from each object in the
source sequence

var productQuery =
    from prod in products
    select new { prod.Color, prod.Price };
```

Extension Methods

[C# 3.0]

```
// Extension methods enable you to "add" methods to existing types
without
// creating a new derived type, recompiling, or otherwise modifying
the original type

namespace ExtensionMethods
{
    public static class MyExtensions
    {
        public static int WordCount(this String str)
        {
            return str.Split(new char[] { ' ', '.', '?' },

StringSplitOptions.RemoveEmptyEntries).Length;
        }
    }
}

string s = "Hello Extension Methods";
// Extension methods are defined as static methods but are called by
using instance method syntax
int i = s.WordCount();
```

Lambda expressions

[C# 3.0]

```
// A lambda expression is an anonymous function that you
// can use to create delegates or expression tree types.
delegate int del(int i);
static void Main(string[] args)
{
    del myDelegate = x => x * x;
    int j = myDelegate(5); //j = 25

    Expression<del> myET = x => x * x;
}
```

Expression trees

[C# 3.0]

```
// Create an expression using expression lambda
Expression<Func<int, int, int>> expression = (num1, num2) => num1 +
num2;

// Compile the expression
Func<int, int, int> compiledExpression = expression.Compile();

// Execute the expression.
int result = compiledExpression(3, 4); //return 7
```

Partial methods

[C# 3.0]

```
partial class MyClass
{
    partial void OnSomethingHappened(string s);
}

// This part can be in a separate file.
partial class MyClass
{
    // Comment out this method and the program
    // will still compile.
    partial void OnSomethingHappened(String s)
    {
        Console.WriteLine("Something happened: {0}", s);
    }
}
```

Query expressions

[C# 3.0]

```
// A query is a set of instructions that describes what data to
retrieve from a given
// data source (or sources) and what shape and organization the
returned data should have.

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

// Query Expression.
IEnumerable<int> scoreQuery = //query variable
  from score in scores //required
  where score > 80 // optional
  orderby score descending // optional
  select score; //must end with select or group

// Execute the query to produce the results
foreach (int testScore in scoreQuery)
{
  Console.WriteLine(testScore);
}
```

Dynamic binding

[C# 4.0]

```
// Dynamic binding refers to delaying the process of type resolution
from compile time to runtime.

// Static binding
Person obj = new Person();
obj.Run(); // Compiler will try to find a method named Run
           // If not found the compiler will generate an error

// Dynamic binding
dynamic obj = new Person();
obj.Run(); // Resolves binding on runtime instead of compile time.
```

Named and optional arguments

[C# 4.0]

```
 // Example method
 public static int Sum(int firstNumber, int secondNumber = 1)
{
    return firstNumber+ secondNumber;
}

// Passing parameters using the normal way
Sum(10, 20);

// Passing parameters using named parameter
Sum(firstNumber: 10, secondNumber: 20);

// Passing parameters using default value
Sum(10);

// Example method using optional parameters
public int Sum(int firstNumber, [Optional] int secondNumber)
{
   return firstNumber + secondNumber;
}

// Example method using params keyword
public int Sum(int firstNumber, params int[] numbers)
{
   int total = 0;
   foreach (int number in numbers)
   {
       number += number;
   }
   return total + firstNumber;
}
```

Generic co and contravariance

- Covariance

```
// Enables you to use a more derived type than originally specified
IEnumerable<Derived> d = new List<Derived>();
IEnumerable<Base> b = d;
```

- Contravariance

```
// Enables you to use a more generic (less derived) type than
originally specified
Action<Base> b = (target) => {
Console.WriteLine(target.GetType().Name); };
Action<Derived> d = b;
d(new Derived());
```

Caller info attributes

[C# 5.0]

```
public void DoProcessing()
{
    TraceMessage("Something happened.");
}

public void TraceMessage(string message,
        [CallerMemberName] string memberName = "",
        [CallerFilePath] string sourceFilePath = "",
        [CallerLineNumber] int sourceLineNumber = 0)
{
    // message: Something happened
    Trace.WriteLine("message: " + message);
    // member name: DoProcessing
    Trace.WriteLine("member name: " + memberName);
    // file path: c:\Users\username\Documents\Form1.cs
    Trace.WriteLine("file path: " + sourceFilePath);
    // source line number: 31
    Trace.WriteLine("source line number: " + sourceLineNumber);
}
```

Asynchronous methods

[C# 5.0]

```
// For I/O-bound code, you await an operation which returns a Task or Task<T> inside of an async method.
private readonly HttpClient _httpClient = new HttpClient();

downloadButton.Clicked += async (o, e) =>
{
    // This line will yield control to the UI as the request
    // from the web service is happening.
    //
    // The UI thread is now free to perform other work.
    var stringData = await _httpClient.GetStringAsync(URL);
    DoSomethingWithData(stringData);
};

// For CPU-bound code, you await an operation which is started on a background thread with the Task.Run method.
private DamageResult CalculateDamageDone()
{
    // Code omitted:
    //
    // Does an expensive calculation and returns
    // the result of that calculation.
}


calculateButton.Clicked += async (o, e) =>
{
    // This line will yield control to the UI while CalculateDamageDone()
    // performs its work.  The UI thread is free to perform other work.
    var damageResult = await Task.Run(() => CalculateDamageDone());
    DisplayDamage(damageResult);
};
```

Compiler as a service Roslyn

```
// Roslyn provides open-source C# and Visual Basic compilers with rich code analysis APIs.

const string programText =
@"using System;
using System.Collections;
using System.Linq;
using System.Text;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine(""Hello, World!"");
        }
    }
}";

// Syntax analysis traversing trees
// Build the syntax tree
SyntaxTree tree = CSharpSyntaxTree.ParseText(programText);
CompilationUnitSyntax root = tree.GetCompilationUnitRoot(); // Retrieve the root node of that tree

// Examine the nodes in the tree.
WriteLine($"The tree is a {root.Kind()} node.");
WriteLine($"The tree has {root.Members.Count} elements in it.");
WriteLine($"The tree has {root.Usings.Count} using statements. They are:");
foreach (UsingDirectiveSyntax element in root.Usings)
    WriteLine($"\t{element.Name}");

// Semantic analysis Querying symbols
var compilation = CSharpCompilation.Create("HelloWorld")
    .AddReferences(MetadataReference.CreateFromFile(
        typeof(string).Assembly.Location))
    .AddSyntaxTrees(tree);

// Querying the semantic model
SemanticModel model = compilation.GetSemanticModel(tree);

// Use the syntax tree to find "using System;"
UsingDirectiveSyntax usingSystem = root.Usings[0];
NameSyntax systemName = usingSystem.Name;

// Use the semantic model for symbol information:
SymbolInfo nameInfo = model.GetSymbolInfo(systemName);
```

Import of static type members into namespace

[C# 6.0]

```
// Without using static
using System;
Math.PI

// Using static directive designates a type whose static members you
can access without specifying a type name.
using static System.Math;
Math.PI
```

Await in catch finally blocks

[C# 6.0]

```
try
{
  await ThatMayThrowAsync();
}
catch (ExpectedException ex)
{
  await Logger.LogAsync(ex);
}
```

Auto property initializers

[C# 6.0]

```
public decimal Price { get; set; } = 0.50m;
public string Name { get; set; } = "John";
```

Nameof operator

[C# 6.0]

```
class Person {
   public string Name { get; set; }
}

var person = new Person();

int number = 0;
string text = "lorem ipsum";
Console.WriteLine(nameof(number)); // number
Console.WriteLine(nameof(text)); // text
Console.WriteLine(nameof(person.Name)); // Name
```

String interpolation

[C# 6.0]

```
Console.WriteLine($"Hello, {name}! Today is {date.DayOfWeek}, it's
{date:HH:mm} now.");
```

Expression-bodied members

[C# 6.0]

```
class Person {
  public string FirstName { get; set; }
  public string LastName { get; set; }
  public string GetFullName() => FirstName + " " + LastName;
}

var person = new Person();
person.FirstName = "John";
person.LastName = "Doe";
Console.WriteLine(person.GetFullName());
```

Dictionary initializer

```
var dictionary = new Dictionary<string, int>
{
    ["one"] = 1,
    ["two"] = 2,
    ["three"] = 3
};
```

Null propagator (null-conditional operator, succinct null checking)

[C# 6.0]

```
int? length = customers?.Length; // null if customers is null
Customer first = customers?[0];  // null if customers is null
int? count = customers?[0]?.Orders?.Count();  // null if customers,
the first customer, or Orders is null
```

Default values for getter only properties

```
public class Dog
{
    public string Name { get; set; }

    // DogCreationTime is immutable
    public DateTime DogCreationTime { get; } = DateTime.Now;

    public Dog(string name)
    {
        Name = name;
    }
}
```

Pattern Matching

[C# 7.0] [Oficial docs]

Patterns test that a value has a certain shape, and can extract information from the value when it has the matching shape.

```
public static void SwitchPattern(object o)
{
    switch (o)
    {
        case null:
            Console.WriteLine("it's a constant pattern");
            break;
        case int i:
            Console.WriteLine("it's an int");
            break;
        case Person p when p.FirstName.StartsWith("A"):
            Console.WriteLine($"a A person {p.FirstName}");
            break;
        case Person p:
            Console.WriteLine($"any other person {p.FirstName}");
            break;
        case var x:
            Console.WriteLine($"it's a var pattern with the type
{x?.GetType().Name} ");
            break;
        default:
            break;
    }
}
```

Tuples

[C# 7.0] [Oficial docs]

Tuples are lightweight data structures that contain multiple fields to represent the data members.

```
// You can create a tuple by assigning a value to each member
(string Alpha, string Beta) namedLetters = ("a", "b");
Console.WriteLine($"{namedLetters.Alpha}, {namedLetters.Beta}");

// You can also specify the names of the fields on the right-hand
side of the assignment
var alphabetStart = (Alpha: "a", Beta: "b");
Console.WriteLine($"{alphabetStart.Alpha}, {alphabetStart.Beta}");
```

Deconstruction

```
// There may be times when you want to unpackage the members of a
tuple that were returned from a method
(int max, int min) = Range(numbers);
Console.WriteLine(max);
Console.WriteLine(min);
```

Local functions

[C# 7.0] [Oficial docs]

Local functions enable you to declare methods inside the context of another method.

```
public static void Main()
{
    Console.WriteLine(Sum(1,1));
}

public static string Sum(int x, int y) {
    return DisplayResult(x + y);

    string DisplayResult(int result) {
        return result.ToString();
    }
}
```

Keywords

```
abstract        // Indicates that the thing being modified has a missing or incomplete implementation
as              // Performs certain types of conversions between compatible reference types or nullable type
base            // Access members of the base class from within a derived class
bool            // Used to declare variables to store the Boolean values, true and false
break           // Terminates the closest enclosing loop or switch statement in which it appears
byte            // Denotes an integral type
case            // Chooses a single switch section to execute from a list of candidates based on a pattern match
catch           // Specify handlers for different exceptions
char            // Represent a Unicode character
checked         // Used to explicitly enable overflow checking for integral-type arithmetic
                // operations and conversions
class           // Create your own custom types by grouping together variables of other types, methods and events
const           // Declare a constant field or a constant local
continue        // Passes control to the next iteration
decimal         // Indicates a 128-bit data type
default         // Can be used in the switch statement or in a default value expression
delegate        // Type that can be used to encapsulate a named or an anonymous method
do              // Executes a statement or a block of statements repeatedly until a specified expression evaluates to false
double          // Simple type that stores 64-bit floating-point values
else            // Identifies which statement to run based on the value of a Boolean expression
enum            // Distinct type that consists of a set of named constants called the enumerator list
event           // Used to declare an event in a publisher class
explicit        // User-defined type conversion operator that must be invoked with a cast
extern          // Modifier is used to declare a method that is implemented externally
false           // Represents boolean false
finally         // Can clean up any resources that are allocated in a try block
fixed           // Prevents the garbage collector from relocating a movable variable
float           // Signifies a simple type that stores 32-bit floating-point values
for             // Run a statement or a block of statements repeatedly until a specified expression evaluates to false
foreach, in     // Repeats a group of embedded statements for each element in an array or an object collection
goto            // Transfers the program control directly to a labeled statement
if              // Identifies which statement to run based on the value of a Boolean expression
implicit        // Used to declare an implicit user-defined type conversion operator
in              // (generic modifier) specifies that the type parameter is contravariant
int             // Denotes an integral type
interface       // Contains only the signatures of methods, properties, events or indexers
internal        // Access modifier fortypes or members are accessible only within files in the same assembly
is              // Checks if an object is compatible with a given type
lock            // Marks a statement block as a critical section by obtaining the mutual-exclusion lock
                // for a given object, executing a statement, and then releasing the lock
long            // Denotes an integral type
namespace       // Keyword is used to declare a scope that contains a set of related objects
new             // Keyword can be used as an operator, a modifier, or a constraint
                // Operator - create objects and invoke constructors
                // Modifier - hide an inherited member from a base class member
                // Constraint - restrict types that might be used as arguments for a type parameter in a generic declaration
```

```
null            // Is a literal that represents a null reference, one that does not refer to any object
object          // All types, predefined and user-defined, reference types and value types, inherit directly or indirectly from Object
operator        // To overload a built-in operator or to provide a user-defined conversion in a class or struct declaration.
out             // As a parameter modifier, which lets you pass an argument to a method by reference
                // rather than by value.
                // Generic type parameter declarations for interfaces and delegates, which specifies that a type
                // parameter is covariant
out             // (generic modifier) Enables you to use a more derived type than that specified
                // by the generic parameter
override        // Modifier is required to extend or modify the abstract or virtual implementation of
                // an inherited method, property, indexer, or event
params          // You can specify a method parameter that takes a variable number of arguments
private         // Is a member access modifier the least permissive access level
protected       // Is a member access modifier accessible within its class and by derived class instances
public          // Is an access modifier for types and type members, the most permissive access level
readonly        // Assignments can only occur as part of the declaration or in a constructor in the same class
ref             // Indicates a value that is passed by reference
return          // Terminates execution of the method in which it appears and returns control to the calling method
sbyte           // An integral type, signed 8-bit integer
sealed          // Prevents other classes from inheriting from it
short           // An integral type, signed 16-bit integer
sizeof          // Obtain the size in bytes for an unmanaged type
stackalloc      // Is used in an unsafe code context to allocate a block of memory on the stack
static          // Modifier to declare a static member, which belongs to the type itself rather than
                // to a specific object
string          // Represents a sequence of zero or more Unicode characters
struct          // Is a value type that is typically used to encapsulate small groups of related variables
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
this            // Refers to the current instance of the class and is also used as a modifier of
                // the first parameter of an extension method
throw           // Signals the occurrence of an exception during program execution
true            // Represents the boolean value true
try             // Is followed by one or more catch clauses, which specify handlers for different exceptions
typeof          // Used to obtain the System.Type object for a type
uint            // An integral type, unsigned 32-bit integer
ulong           // Denotes an integral type, unsigned 64-bit integer
unchecked       // Is used to suppress overflow-checking for integral-type arithmetic operations and conversions
unsafe          // Denotes an unsafe context, which is required for any operation involving pointers
ushort          // An integral type, unsigned 16-bit integer
using           // As a directive, when it is used to create an alias for a namespace or to import types
                // defined in other namespace. As a statement, when it defines a scope at the end of which
                // an object will be disposed
using static    // Designates a type whose static members you can access without specifying a type name
virtual         // Is used to modify a method, property, indexer, or event declaration and allow for it to
                // be overridden in a derived class
void            // Specifies that the method doesn't return a value.
volatile        // Indicates that a field might be modified by multiple threads that are executing at the same time
while           // Executes a statement or a block of statements until a specified expression evaluates to false
```

Contextual Keywords

```
add             // Define a custom event accessor that is invoked when client code subscribes to your event
alias           // Reference two versions of assemblies that have the same fully-qualified type names
ascending       // Used in the orderby clause in query expressions to specify that the sort order is from smallest to largest
async           // Specify that a method, lambda expression, or anonymous method is asynchronous
await           // Applied to a task in an asynchronous method to insert a suspension point in the execution of the method until the
                // awaited task completes
descending      // Used in the orderby clause in query expressions to specify that the sort order is from largest to smallest
dynamic         // Enables the operations in which it occurs to bypass compile-time type checking
from            // A query expression must begin with a from clause
get             // Defines an accessor method in a property or indexer that returns the property value or the indexer element
global          // Refers to the global namespace
group           // Sequence of IGrouping<TKey,TElement> objects that contain zero or more items that match the key value for the group
into            // Used to create a temporary identifier to store the results of a group, join or select clause into a new identifier
join            // Useful for associating elements from different source sequences that have no direct relationship in the object model
let             // Useful to store the result of a sub-expression in order to use it in subsequent clauses
nameof          // Used to obtain the simple (unqualified) string name of a variable, type, or member
orderby         // Causes the returned sequence or subsequence (group) to be sorted in either ascending or descending order
partial         // (type) Allow for the definition of a class, struct, or interface to be split into multiple files
partial         // (method) A partial method has its signature defined in one part of a partial type, and its implementation defined in
                // another part of the type
remove          // Used to define a custom event accessor that is invoked when client code unsubscribes from your event
select          // Specifies the type of values that will be produced when the query is executed
set             // Accessor method in a property or indexer that assigns a value to the property or the indexer element
value           // Used in the set accessor in ordinary property declarations.
var             // Variables that are declared at method scope can have an implicit "type" var
when            // Used as catch statement of a try/catch or try/catch/finally block or label of a switch statement
where           // (generic type constraint) Specify constraints on the types that can be used as arguments for a type parameter defined
                // in a generic declaration
where           // Specify which elements from the data source will be returned in the query expression
yield           // You indicate that the method, operator, or get accessor in which it appears is an iterator
```