A tour of the C# language

3/6/2021 • 13 minutes to read • Edit Online

C# (pronounced "See Sharp") is a modern, object-oriented, and type-safe programming language. C# enables developers to build many types of secure and robust applications that run in the .NET ecosystem. C# has its roots in the C family of languages and will be immediately familiar to C, C++, Java, and JavaScript programmers. This tour provides an overview of the major components of the language in C# 8 and earlier. If you want to explore the language through interactive examples, try the introduction to C# tutorials.

C# is an object-oriented, ***component-oriented*** programming language. C# provides language constructs to directly support these concepts, making C# a natural language in which to create and use software components. Since its origin, C# has added features to support new workloads and emerging software design practices.

Several C# features help create robust and durable applications. ***Garbage collection*** automatically reclaims memory occupied by unreachable unused objects. ***Nullable types*** guard against variables that don't refer to allocated objects. ***Exception handling*** provides a structured and extensible approach to error detection and recovery. ***Lambda expressions*** support functional programming techniques. ***Language Integrated Query (LINQ)*** syntax creates a common pattern for working with data from any source. Language support for ***asynchronous operations*** provides syntax for building distributed systems. C# has a ***unified type system***. All C# types, including primitive types such as `int` and `double`, inherit from a single root `object` type. All types share a set of common operations. Values of any type can be stored, transported, and operated upon in a consistent manner. Furthermore, C# supports both user-defined reference types and value types. C# allows dynamic allocation of objects and in-line storage of lightweight structures. C# supports generic methods and types, which provide increased type safety and performance. C# provides iterators, which enable implementers of collection classes to define custom behaviors for client code.

C# emphasizes ***versioning*** to ensure programs and libraries can evolve over time in a compatible manner. Aspects of C#'s design that were directly influenced by versioning considerations include the separate `virtual` and `override` modifiers, the rules for method overload resolution, and support for explicit interface member declarations.

.NET architecture

C# programs run on .NET, a virtual execution system called the common language runtime (CLR) and a set of class libraries. The CLR is the implementation by Microsoft of the common language infrastructure (CLI), an international standard. The CLI is the basis for creating execution and development environments in which languages and libraries work together seamlessly.

Source code written in C# is compiled into an intermediate language (IL) that conforms to the CLI specification. The IL code and resources, such as bitmaps and strings, are stored in an assembly, typically with an extension of *.dll*. An assembly contains a manifest that provides information about the assembly's types, version, and culture.

When the C# program is executed, the assembly is loaded into the CLR. The CLR performs Just-In-Time (JIT) compilation to convert the IL code to native machine instructions. The CLR provides other services related to automatic garbage collection, exception handling, and resource management. Code that's executed by the CLR is sometimes referred to as "managed code," in contrast to "unmanaged code," which is compiled into native machine language that targets a specific platform.

Language interoperability is a key feature of .NET. IL code produced by the C# compiler conforms to the Common Type Specification (CTS). IL code generated from C# can interact with code that was generated from the .NET versions of F#, Visual Basic, C++, or any of more than 20 other CTS-compliant languages. A single

assembly may contain multiple modules written in different .NET languages, and the types can reference each other as if they were written in the same language.

In addition to the run time services, .NET also includes extensive libraries. These libraries support many different workloads. They're organized into namespaces that provide a wide variety of useful functionality for everything from file input and output to string manipulation to XML parsing, to web application frameworks to Windows Forms controls. The typical C# application uses the .NET class library extensively to handle common "plumbing" chores.

For more information about .NET, see Overview of .NET.

Hello world

The "Hello, World" program is traditionally used to introduce a programming language. Here it is in C#:

```
using System;

class Hello
{
    static void Main()
    {
        Console.WriteLine("Hello, World");
    }
}
```

The "Hello, World" program starts with a `using` directive that references the `System` namespace. Namespaces provide a hierarchical means of organizing C# programs and libraries. Namespaces contain types and other namespaces—for example, the `System` namespace contains a number of types, such as the `Console` class referenced in the program, and a number of other namespaces, such as `IO` and `Collections`. A `using` directive that references a given namespace enables unqualified use of the types that are members of that namespace. Because of the `using` directive, the program can use `Console.WriteLine` as shorthand for `System.Console.WriteLine`.

The `Hello` class declared by the "Hello, World" program has a single member, the method named `Main`. The `Main` method is declared with the `static` modifier. While instance methods can reference a particular enclosing object instance using the keyword `this`, static methods operate without reference to a particular object. By convention, a static method named `Main` serves as the entry point of a C# program.

The output of the program is produced by the `WriteLine` method of the `Console` class in the `System` namespace. This class is provided by the standard class libraries, which, by default, are automatically referenced by the compiler.

Types and variables

There are two kinds of types in C#: *value types* and *reference types*. Variables of value types directly contain their data. Variables of reference types store references to their data, the latter being known as objects. With reference types, it's possible for two variables to reference the same object and possible for operations on one variable to affect the object referenced by the other variable. With value types, the variables each have their own copy of the data, and it isn't possible for operations on one to affect the other (except for `ref` and `out` parameter variables).

An ***identifier*** is a variable name. An identifier is a sequence of unicode characters without any whitespace. An identifier may be a C# reserved word, if it's prefixed by `@`. Using a reserved word as an identifier can be useful when interacting with other languages.

C#'s value types are further divided into *simple types*, *enum types*, *struct types*, *nullable value types*, and *tuple*

value types. C#'s reference types are further divided into *class types*, *interface types*, *array types*, and *delegate types*.

The following outline provides an overview of C#'s type system.

- Value types
  - Simple types
    - Signed integral: `sbyte`, `short`, `int`, `long`
    - Unsigned integral: `byte`, `ushort`, `uint`, `ulong`
    - Unicode characters: `char`, which represents a UTF-16 code unit
    - IEEE binary floating-point: `float`, `double`
    - High-precision decimal floating-point: `decimal`
    - Boolean: `bool`, which represents Boolean values—values that are either `true` or `false`
  - Enum types
    - User-defined types of the form `enum E {...}`. An `enum` type is a distinct type with named constants. Every `enum` type has an underlying type, which must be one of the eight integral types. The set of values of an `enum` type is the same as the set of values of the underlying type.
  - Struct types
    - User-defined types of the form `struct S {...}`
  - Nullable value types
    - Extensions of all other value types with a `null` value
  - Tuple value types
    - User-defined types of the form `(T1, T2, ...)`
- Reference types
  - Class types
    - Ultimate base class of all other types: `object`
    - Unicode strings: `string`, which represents a sequence of UTF-16 code units
    - User-defined types of the form `class C {...}`
  - Interface types
    - User-defined types of the form `interface I {...}`
  - Array types
    - Single-dimensional, multi-dimensional, and jagged. For example: `int[]`, `int[,]`, and `int[][]`
  - Delegate types
    - User-defined types of the form `delegate int D(...)`

C# programs use *type declarations* to create new types. A type declaration specifies the name and the members of the new type. Six of C#'s categories of types are user-definable: class types, struct types, interface types, enum types, delegate types, and tuple value types.

- A `class` type defines a data structure that contains data members (fields) and function members (methods, properties, and others). Class types support single inheritance and polymorphism, mechanisms whereby derived classes can extend and specialize base classes.
- A `struct` type is similar to a class type in that it represents a structure with data members and function members. However, unlike classes, structs are value types and don't typically require heap allocation. Struct types don't support user-specified inheritance, and all struct types implicitly inherit from type `object`.
- An `interface` type defines a contract as a named set of public members. A `class` or `struct` that implements an `interface` must provide implementations of the interface's members. An `interface` may inherit from multiple base interfaces, and a `class` or `struct` may implement multiple interfaces.
- A `delegate` type represents references to methods with a particular parameter list and return type.

Delegates make it possible to treat methods as entities that can be assigned to variables and passed as parameters. Delegates are analogous to function types provided by functional languages. They're also similar to the concept of function pointers found in some other languages. Unlike function pointers, delegates are object-oriented and type-safe.

The `class`, `struct`, `interface`, and `delegate` types all support generics, whereby they can be parameterized with other types.

C# supports single-dimensional and multi-dimensional arrays of any type. Unlike the types listed above, array types don't have to be declared before they can be used. Instead, array types are constructed by following a type name with square brackets. For example, `int[]` is a single-dimensional array of `int`, `int[,]` is a two-dimensional array of `int`, and `int[][]` is a single-dimensional array of single-dimensional arrays, or a "jagged" array, of `int`.

Nullable types don't require a separate definition. For each non-nullable type `T`, there's a corresponding nullable type `T?`, which can hold an additional value, `null`. For instance, `int?` is a type that can hold any 32-bit integer or the value `null`, and `string?` is a type that can hold any `string` or the value `null`.

C#'s type system is unified such that a value of any type can be treated as an `object`. Every type in C# directly or indirectly derives from the `object` class type, and `object` is the ultimate base class of all types. Values of reference types are treated as objects simply by viewing the values as type `object`. Values of value types are treated as objects by performing *boxing* and *unboxing operations*. In the following example, an `int` value is converted to `object` and back again to `int`.

```
int i = 123;
object o = i;    // Boxing
int j = (int)o;  // Unboxing
```

When a value of a value type is assigned to an `object` reference, a "box" is allocated to hold the value. That box is an instance of a reference type, and the value is copied into that box. Conversely, when an `object` reference is cast to a value type, a check is made that the referenced `object` is a box of the correct value type. If the check succeeds, the value in the box is copied to the value type.

C#'s unified type system effectively means that value types are treated as `object` references "on demand." Because of the unification, general-purpose libraries that use type `object` can be used with all types that derive from `object`, including both reference types and value types.

There are several kinds of *variables* in C#, including fields, array elements, local variables, and parameters. Variables represent storage locations. Every variable has a type that determines what values can be stored in the variable, as shown below.

- Non-nullable value type
  - A value of that exact type
- Nullable value type
  - A `null` value or a value of that exact type
- object
  - A `null` reference, a reference to an object of any reference type, or a reference to a boxed value of any value type
- Class type
  - A `null` reference, a reference to an instance of that class type, or a reference to an instance of a class derived from that class type
- Interface type
  - A `null` reference, a reference to an instance of a class type that implements that interface type, or a

reference to a boxed value of a value type that implements that interface type

- Array type
  - A `null` reference, a reference to an instance of that array type, or a reference to an instance of a compatible array type
- Delegate type
  - A `null` reference or a reference to an instance of a compatible delegate type

Program structure

The key organizational concepts in C# are ***programs***, ***namespaces***, ***types***, ***members***, and ***assemblies***. Programs declare types, which contain members and can be organized into namespaces. Classes, structs, and interfaces are examples of types. Fields, methods, properties, and events are examples of members. When C# programs are compiled, they're physically packaged into assemblies. Assemblies typically have the file extension `.exe` or `.dll`, depending on whether they implement ***applications*** or ***libraries***, respectively.

As a small example, consider an assembly that contains the following code:

```
using System;

namespace Acme.Collections
{
    public class Stack<T>
    {
        Entry _top;

        public void Push(T data)
        {
            _top = new Entry(_top, data);
        }

        public T Pop()
        {
            if (_top == null)
            {
                throw new InvalidOperationException();
            }
            T result = _top.Data;
            _top = _top.Next;

            return result;
        }

        class Entry
        {
            public Entry Next { get; set; }
            public T Data { get; set; }

            public Entry(Entry next, T data)
            {
                Next = next;
                Data = data;
            }
        }
    }
}
```

The fully qualified name of this class is `Acme.Collections.Stack`. The class contains several members: a field named `top`, two methods named `Push` and `Pop`, and a nested class named `Entry`. The `Entry` class further contains three members: a field named `next`, a field named `data`, and a constructor. The `Stack` is a *generic* class. It has one type parameter, `T` that is replaced with a concrete type when it's used.

NOTE

A *stack* is a "first in - last out" (FILO) collection. New elements are added to the top of the stack. When an element is removed, it is removed from the top of the stack.

Assemblies contain executable code in the form of Intermediate Language (IL) instructions, and symbolic information in the form of metadata. Before it's executed, the Just-In-Time (JIT) compiler of .NET Common Language Runtime converts the IL code in an assembly to processor-specific code.

Because an assembly is a self-describing unit of functionality containing both code and metadata, there's no need for `#include` directives and header files in C#. The public types and members contained in a particular assembly are made available in a C# program simply by referencing that assembly when compiling the program. For example, this program uses the `Acme.Collections.Stack` class from the `acme.dll` assembly:

```
using System;
using Acme.Collections;

class Example
{
    public static void Main()
    {
        var s = new Stack<int>();
        s.Push(1); // stack contains 1
        s.Push(10); // stack contains 1, 10
        s.Push(100); // stack contains 1, 10, 100
        Console.WriteLine(s.Pop()); // stack contains 1, 10
        Console.WriteLine(s.Pop()); // stack contains 1
        Console.WriteLine(s.Pop()); // stack is empty
    }
}
```

To compile this program, you would need to *reference* the assembly containing the stack class defined in the earlier example.

C# programs can be stored in several source files. When a C# program is compiled, all of the source files are processed together, and the source files can freely reference each other. Conceptually, it's as if all the source files were concatenated into one large file before being processed. Forward declarations are never needed in C# because, with few exceptions, declaration order is insignificant. C# doesn't limit a source file to declaring only one public type nor does it require the name of the source file to match a type declared in the source file.

Further articles in this tour explain these organizational blocks.

NEXT