
1. Introduction

C# (pronounced “see sharp”) is a simple, modern, object-oriented, and type-safe programming language. C# has its roots in the C family of languages and will be immediately familiar to C, C++, and Java programmers. C# is standardized by ECMA International as the *ECMA-334* standard and by ISO/IEC as the *ISO/IEC 23270* standard. Microsoft’s C# compiler for the .NET Framework is a conforming implementation of both of these standards.

C# is an object-oriented language, but C# further includes support for *component-oriented* programming. Contemporary software design increasingly relies on software components in the form of self-contained and self-describing packages of functionality. Key to such components is that they present a programming model with properties, methods, and events; they have attributes that provide declarative information about the component; and they incorporate their own documentation. C# provides language constructs to directly support these concepts, making C# a very natural language in which to create and use software components.

Several C# features aid in the construction of robust and durable applications: *Garbage collection* automatically reclaims memory occupied by unused objects; *exception handling* provides a structured and extensible approach to error detection and recovery; and the *type-safe* design of the language makes it impossible to read from uninitialized variables, to index arrays beyond their bounds, or to perform unchecked type casts.

C# has a *unified type system*. All C# types, including primitive types such as `int` and `double`, inherit from a single root object type. Thus, all types share a set of common operations, and 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, allowing dynamic allocation of objects as well as in-line storage of lightweight structures.

To ensure that C# programs and libraries can evolve over time in a compatible manner, much emphasis has been placed on *versioning* in C#’s design. Many programming languages pay little attention to this issue and, as a result, programs written in those languages break more often than necessary when newer versions of dependent libraries are introduced. 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.

■ **ERIC LIPPERT** This is unlike the Java language. Also, the fact that the declaration order is insignificant in C# is unlike the C++ language.

■ **CHRIS SELLS** Notice in the previous example the `using Acme.Collections` statement, which looks like a C-style `#include` directive, but isn't. Instead, it's merely a naming convenience so that when the compiler encounters the `Stack`, it has a set of namespaces in which to look for the class. It would have been just the same to the compiler if this example used the fully qualified name:

```
Acme.Collections.Stack s = new Acme.Collections.Stack();
```

■ **DON BOX** What's interesting is that by version 2.0, C# had the partial class feature, which in my experience has been much more useful than `#include`. It certainly is better integrated into the language and tool ecosystem.

1.3 Types and Variables

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

■ **DON BOX** Putting the value/reference distinction on the definition of the type versus the use of the type took some adjustment for those of us coming from the C++ world. While intellectually I prefer the C++ model, in which the user of the type makes the decision, pragmatically it's very nice to make the decision once at type-def time and then largely forget about it the thousands of times you use the type as a field or parameter type.

C#'s value types are further divided into *simple types*, *enum types*, *struct types*, and *nullable types*; its reference types are further divided into *class types*, *interface types*, *array types*, and *delegate types*.

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

Category		Description
Value types	Simple types	Signed integral: <code>sbyte</code> , <code>short</code> , <code>int</code> , <code>long</code>
		Unsigned integral: <code>byte</code> , <code>ushort</code> , <code>uint</code> , <code>ulong</code>
		Unicode characters: <code>char</code>
		IEEE floating point: <code>float</code> , <code>double</code>
		High-precision decimal: <code>decimal</code>
		Boolean: <code>bool</code>
	Enum types	User-defined types of the form <code>enum E { ... }</code>
	Struct types	User-defined types of the form <code>struct S { ... }</code>
Nullable types	Extensions of all other value types with a <code>null</code> value	
Reference types	Class types	Ultimate base class of all other types: <code>object</code>
		Unicode strings: <code>string</code>
		User-defined types of the form <code>class C { ... }</code>
	Interface types	User-defined types of the form <code>interface I { ... }</code>
	Array types	Single- and multi-dimensional, for example, <code>int[]</code> and <code>int[,]</code>
	Delegate types	User-defined types of the form e.g. <code>delegate int D(...)</code>

The eight integral types provide support for 8-bit, 16-bit, 32-bit, and 64-bit values in signed or unsigned form.

The two floating point types, `float` and `double`, are represented using the 32-bit single-precision and 64-bit double-precision IEEE 754 formats.

The `decimal` type is a 128-bit data type suitable for financial and monetary calculations.

C#'s `bool` type is used to represent boolean values—values that are either `true` or `false`.

Character and string processing in C# uses Unicode encoding. The `char` type represents a UTF-16 code unit, and the `string` type represents a sequence of UTF-16 code units.