Contents

C# programming guide

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

This section provides detailed information on key C# language features and features accessible to C# through the .NET Framework.

Most of this section assumes that you already know something about C# and general programming concepts. If you are a complete beginner with programming or with C#, you might want to visit the Introduction to C# Tutorials or Getting Started with C# interactive tutorial, where no prior programming knowledge is required.

For information about specific keywords, operators and preprocessor directives, see C# Reference. For information about the C# Language Specification, see C# Language Specification.

Program sections

Inside a C# Program

Main() and Command-Line Arguments

Language Sections

Statements, Expressions, and Operators

Types

Classes and Structs

Interfaces

Enumeration Types

Delegates

Arrays

Strings

Properties

Indexers

Events

Generics

Iterators

LINQ Query Expressions

Lambda Expressions

Namespaces

Nullable Types

Unsafe Code and Pointers

XML Documentation Comments

Platform Sections

Application Domains

Assemblies and the Global Assembly Cache

Attributes

Collections

Exceptions and Exception Handling

File System and the Registry (C# Programming Guide)

Interoperability

Reflection

See also

- C# Reference
- C#

Inside a C# program

1/11/2019 • 2 minutes to read • Edit Online

The section discusses the general structure of a C# program, and includes the standard "Hello, World!" example.

In this section

- Hello World -- Your First Program
- General Structure of a C# Program
- Identifier names
- C# Coding Conventions

Related sections

- Getting Started with C#
- C# Programming Guide
- C# Reference
- Samples and tutorials

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

Main() and command-line arguments (C# Programming Guide)

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

The `Main` method is the entry point of a C# application. (Libraries and services do not require a `Main` method as an entry point.) When the application is started, the `Main` method is the first method that is invoked.

There can only be one entry point in a C# program. If you have more than one class that has a `Main` method, you must compile your program with the **/main** compiler option to specify which `Main` method to use as the entry point. For more information, see /main (C# Compiler Options).

```
class TestClass
{
    static void Main(string[] args)
    {
        // Display the number of command line arguments:
        System.Console.WriteLine(args.Length);
    }
}
```

Overview

- The `Main` method is the entry point of an executable program; it is where the program control starts and ends.
- `Main` is declared inside a class or struct. `Main` must be static and it need not be public. (In the earlier example, it receives the default access of private.) The enclosing class or struct is not required to be static.
- `Main` can either have a `void`, `int`, or, starting with C# 7.1, `Task`, or `Task<int>` return type.
- If and only if `Main` returns a `Task` or `Task<int>`, the declaration of `Main` may include the `async` modifier. Note that this specifically excludes an `async void Main` method.
- The `Main` method can be declared with or without a `string[]` parameter that contains command-line arguments. When using Visual Studio to create Windows applications, you can add the parameter manually or else use the Environment class to obtain the command-line arguments. Parameters are read as zero-indexed command-line arguments. Unlike C and C++, the name of the program is not treated as the first command-line argument.

The addition of `async` and `Task`, `Task<int>` return types simplifies program code when console applications need to start and `await` asynchronous operations in `Main`.

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

- Command-line Building With csc.exe
- C# Programming Guide
- Methods
- Inside a C# Program

Programming Concepts (C#)

11/9/2018 • 2 minutes to read • Edit Online

This section explains programming concepts in the C# language.

In This Section

TITLE	DESCRIPTION
Assemblies and the Global Assembly Cache (C#)	Describes how to create and use assemblies.
Asynchronous Programming with async and await (C#)	Describes how to write asynchronous solutions by using the async and await keywords in C#. Includes a walkthrough.
Attributes (C#)	Discusses how to provide additional information about programming elements such as types, fields, methods, and properties by using attributes.
Caller Information (C#)	Describes how to obtain information about the caller of a method. This information includes the file path and the line number of the source code and the member name of the caller.
Collections (C#)	Describes some of the types of collections provided by the .NET Framework. Demonstrates how to use simple collections and collections of key/value pairs.
Covariance and Contravariance (C#)	Shows how to enable implicit conversion of generic type parameters in interfaces and delegates.
Expression Trees (C#)	Explains how you can use expression trees to enable dynamic modification of executable code.
Iterators (C#)	Describes iterators, which are used to step through collections and return elements one at a time.
Language-Integrated Query (LINQ) (C#)	Discusses the powerful query capabilities in the language syntax of C#, and the model for querying relational databases, XML documents, datasets, and in-memory collections.
Object-Oriented Programming (C#)	Describes common object-oriented concepts, including encapsulation, inheritance, and polymorphism.
Reflection (C#)	Explains how to use reflection to dynamically create an instance of a type, bind the type to an existing object, or get the type from an existing object and invoke its methods or access its fields and properties.
Serialization (C#)	Describes key concepts in binary, XML, and SOAP serialization.

Related Sections

Performance Tips	Discusses several basic rules that may help you increase the performance of your application.

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

Types (C# Programming Guide)

2/5/2019 • 11 minutes to read • Edit Online

Types, Variables, and Values

C# is a strongly-typed language. Every variable and constant has a type, as does every expression that evaluates to a value. Every method signature specifies a type for each input parameter and for the return value. The .NET class library defines a set of built-in numeric types as well as more complex types that represent a wide variety of logical constructs, such as the file system, network connections, collections and arrays of objects, and dates. A typical C# program uses types from the class library as well as user-defined types that model the concepts that are specific to the program's problem domain.

The information stored in a type can include the following:

- The storage space that a variable of the type requires.
- The maximum and minimum values that it can represent.
- The members (methods, fields, events, and so on) that it contains.
- The base type it inherits from.
- The location where the memory for variables will be allocated at run time.
- The kinds of operations that are permitted.

The compiler uses type information to make sure that all operations that are performed in your code are *type safe*. For example, if you declare a variable of type int, the compiler allows you to use the variable in addition and subtraction operations. If you try to perform those same operations on a variable of type bool, the compiler generates an error, as shown in the following example:

```
int a = 5;
int b = a + 2; //OK

bool test = true;

// Error. Operator '+' cannot be applied to operands of type 'int' and 'bool'.
int c = a + test;
```

> **NOTE**
> C and C++ developers, notice that in C#, bool is not convertible to int.

The compiler embeds the type information into the executable file as metadata. The common language runtime (CLR) uses that metadata at run time to further guarantee type safety when it allocates and reclaims memory.

Specifying Types in Variable Declarations

When you declare a variable or constant in a program, you must either specify its type or use the var keyword to let the compiler infer the type. The following example shows some variable declarations that use both built-in numeric types and complex user-defined types:

```
// Declaration only:
float temperature;
string name;
MyClass myClass;

// Declaration with initializers (four examples):
char firstLetter = 'C';
var limit = 3;
int[] source = { 0, 1, 2, 3, 4, 5 };
var query = from item in source
            where item <= limit
            select item;
```

The types of method parameters and return values are specified in the method signature. The following signature shows a method that requires an int as an input argument and returns a string:

```
public string GetName(int ID)
{
    if (ID < names.Length)
        return names[ID];
    else
        return String.Empty;
}
private string[] names = { "Spencer", "Sally", "Doug" };
```

After a variable is declared, it cannot be re-declared with a new type, and it cannot be assigned a value that is not compatible with its declared type. For example, you cannot declare an int and then assign it a Boolean value of true. However, values can be converted to other types, for example when they are assigned to new variables or passed as method arguments. A *type conversion* that does not cause data loss is performed automatically by the compiler. A conversion that might cause data loss requires a *cast* in the source code.

For more information, see Casting and Type Conversions.

Built-in Types

C# provides a standard set of built-in numeric types to represent integers, floating point values, Boolean expressions, text characters, decimal values, and other types of data. There are also built-in `string` and `object` types. These are available for you to use in any C# program. For more information about the built-in types, see Reference Tables for Types.

Custom Types

You use the struct, class, interface, and enum constructs to create your own custom types. The .NET class library itself is a collection of custom types provided by Microsoft that you can use in your own applications. By default, the most frequently used types in the class library are available in any C# program. Others become available only when you explicitly add a project reference to the assembly in which they are defined. After the compiler has a reference to the assembly, you can declare variables (and constants) of the types declared in that assembly in source code. For more information, see .NET Class Library.

The Common Type System

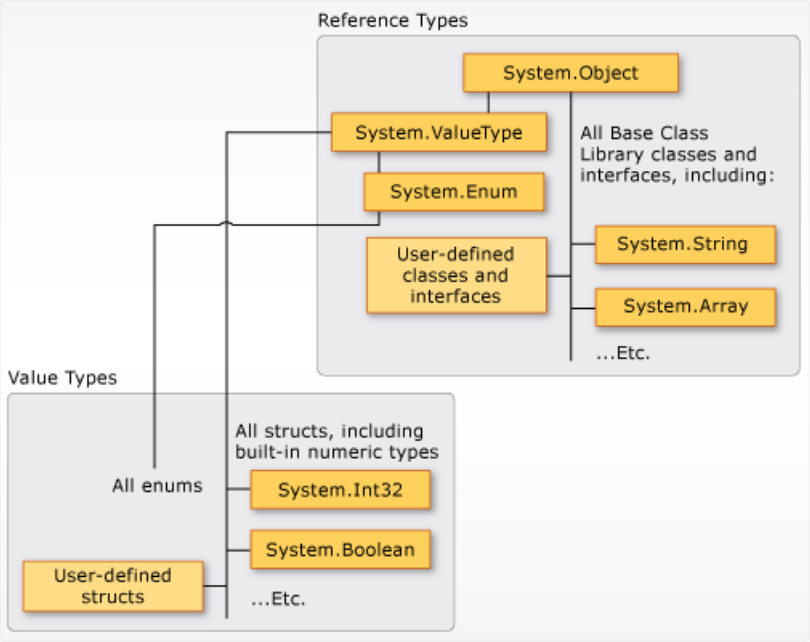
It is important to understand two fundamental points about the type system in .NET:

- It supports the principle of inheritance. Types can derive from other types, called *base types*. The derived type inherits (with some restrictions) the methods, properties, and other members of the base type. The base type can in turn derive from some other type, in which case the derived type inherits the members of

both base types in its inheritance hierarchy. All types, including built-in numeric types such as System.Int32 (C# keyword: int), derive ultimately from a single base type, which is System.Object (C# keyword: object). This unified type hierarchy is called the Common Type System (CTS). For more information about inheritance in C#, see Inheritance.

- Each type in the CTS is defined as either a *value type* or a *reference type*. This includes all custom types in the .NET class library and also your own user-defined types. Types that you define by using the struct keyword are value types; all the built-in numeric types are `structs`. Types that you define by using the class keyword are reference types. Reference types and value types have different compile-time rules, and different run-time behavior.

The following illustration shows the relationship between value types and reference types in the CTS.



Value types and reference types in the CTS

> **NOTE**
>
> You can see that the most commonly used types are all organized in the System namespace. However, the namespace in which a type is contained has no relation to whether it is a value type or reference type.

Value Types

Value types derive from System.ValueType, which derives from System.Object. Types that derive from System.ValueType have special behavior in the CLR. Value type variables directly contain their values, which means that the memory is allocated inline in whatever context the variable is declared. There is no separate heap allocation or garbage collection overhead for value-type variables.

There are two categories of value types: struct and enum.

The built-in numeric types are structs, and they have properties and methods that you can access:

```
// Static method on type byte.
byte b = byte.MaxValue;
```

But you declare and assign values to them as if they were simple non-aggregate types:

```
byte num = 0xA;
int i = 5;
char c = 'Z';
```

Value types are *sealed*, which means, for example, that you cannot derive a type from System.Int32, and you cannot define a struct to inherit from any user-defined class or struct because a struct can only inherit from System.ValueType. However, a struct can implement one or more interfaces. You can cast a struct type to any interface type that it implements; this causes a *boxing* operation to wrap the struct inside a reference type object on the managed heap. Boxing operations occur when you pass a value type to a method that takes a System.Object or any interface type as an input parameter. For more information, see Boxing and Unboxing.

You use the struct keyword to create your own custom value types. Typically, a struct is used as a container for a small set of related variables, as shown in the following example:

```
public struct Coords
{
    public int x, y;

    public Coords(int p1, int p2)
    {
        x = p1;
        y = p2;
    }
}
```

For more information about structs, see Structs. For more information about value types in .NET, see Value Types.

The other category of value types is enum. An enum defines a set of named integral constants. For example, the System.IO.FileMode enumeration in the .NET class library contains a set of named constant integers that specify how a file should be opened. It is defined as shown in the following example:

```
public enum FileMode
{
    CreateNew = 1,
    Create = 2,
    Open = 3,
    OpenOrCreate = 4,
    Truncate = 5,
    Append = 6,
}
```

The `System.IO.FileMode.Create` constant has a value of 2. However, the name is much more meaningful for humans reading the source code, and for that reason it is better to use enumerations instead of constant literal numbers. For more information, see System.IO.FileMode.

All enums inherit from System.Enum, which inherits from System.ValueType. All the rules that apply to structs also apply to enums. For more information about enums, see Enumeration Types.

Reference Types

A type that is defined as a class, delegate, array, or interface is a *reference type*. At run time, when you declare a variable of a reference type, the variable contains the value null until you explicitly create an object by using the new operator, or assign it an object that has been created elsewhere by using `new`, as shown in the following example:

```
MyClass mc = new MyClass();
MyClass mc2 = mc;
```

An interface must be initialized together with a class object that implements it. If `MyClass` implements `IMyInterface`, you create an instance of `IMyInterface` as shown in the following example:

```
IMyInterface iface = new MyClass();
```

When the object is created, the memory is allocated on the managed heap, and the variable holds only a reference to the location of the object. Types on the managed heap require overhead both when they are allocated and when they are reclaimed by the automatic memory management functionality of the CLR, which is known as *garbage collection*. However, garbage collection is also highly optimized, and in most scenarios it does not create a performance issue. For more information about garbage collection, see Automatic Memory Management.

All arrays are reference types, even if their elements are value types. Arrays implicitly derive from the System.Array class, but you declare and use them with the simplified syntax that is provided by C#, as shown in the following example:

```
// Declare and initialize an array of integers.
int[] nums = { 1, 2, 3, 4, 5 };

// Access an instance property of System.Array.
int len = nums.Length;
```

Reference types fully support inheritance. When you create a class, you can inherit from any other interface or class that is not defined as sealed, and other classes can inherit from your class and override your virtual methods. For more information about how to create your own classes, see Classes and Structs. For more information about inheritance and virtual methods, see Inheritance.

Types of Literal Values

In C#, literal values receive a type from the compiler. You can specify how a numeric literal should be typed by appending a letter to the end of the number. For example, to specify that the value 4.56 should be treated as a float, append an "f" or "F" after the number: `4.56f`. If no letter is appended, the compiler will infer a type for the literal. For more information about which types can be specified with letter suffixes, see the reference pages for individual types in Value Types.

Because literals are typed, and all types derive ultimately from System.Object, you can write and compile code such as the following:

```
string s = "The answer is " + 5.ToString();
// Outputs: "The answer is 5"
Console.WriteLine(s);

Type type = 12345.GetType();
// Outputs: "System.Int32"
Console.WriteLine(type);
```

Generic Types

A type can be declared with one or more *type parameters* that serve as a placeholder for the actual type (the *concrete type*) that client code will provide when it creates an instance of the type. Such types are called *generic types*. For example, the .NET type System.Collections.Generic.List<T> has one type parameter that by convention is given the name *T*. When you create an instance of the type, you specify the type of the objects that the list will contain, for example, string:

```
List<string> stringList = new List<string>();
stringList.Add("String example");
// compile time error adding a type other than a string:
stringList.Add(4);
```

The use of the type parameter makes it possible to reuse the same class to hold any type of element, without having to convert each element to object. Generic collection classes are called *strongly-typed collections* because the compiler knows the specific type of the collection's elements and can raise an error at compile-time if, for example, you try to add an integer to the `stringList` object in the previous example. For more information, see Generics.

Implicit Types, Anonymous Types, and Nullable Types

As stated previously, you can implicitly type a local variable (but not class members) by using the var keyword. The variable still receives a type at compile time, but the type is provided by the compiler. For more information, see Implicitly Typed Local Variables.

In some cases, it is inconvenient to create a named type for simple sets of related values that you do not intend to store or pass outside method boundaries. You can create *anonymous types* for this purpose. For more information, see Anonymous Types.

Ordinary value types cannot have a value of null. However, you can create nullable value types by affixing a `?` after the type. For example, `int?` is an `int` type that can also have the value null. In the CTS, nullable types are instances of the generic struct type System.Nullable<T>. Nullable types are especially useful when you are passing data to and from databases in which numeric values might be null. For more information, see Nullable Types.

Related Sections

For more information, see the following topics:

- Casting and Type Conversions
- Boxing and Unboxing
- Using Type dynamic
- Value Types
- Reference Types
- Classes and Structs
- Anonymous Types
- Generics

C# Language Specification

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

See also

- C# Reference
- C# Programming Guide
- Conversion of XML Data Types

- Integral Types Table

Classes and Structs (C# Programming Guide)

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

Classes and structs are two of the basic constructs of the common type system in the .NET Framework. Each is essentially a data structure that encapsulates a set of data and behaviors that belong together as a logical unit. The data and behaviors are the *members* of the class or struct, and they include its methods, properties, and events, and so on, as listed later in this topic.

A class or struct declaration is like a blueprint that is used to create instances or objects at run time. If you define a class or struct called `Person`, `Person` is the name of the type. If you declare and initialize a variable `p` of type `Person`, `p` is said to be an object or instance of `Person`. Multiple instances of the same `Person` type can be created, and each instance can have different values in its properties and fields.

A class is a reference type. When an object of the class is created, the variable to which the object is assigned holds only a reference to that memory. When the object reference is assigned to a new variable, the new variable refers to the original object. Changes made through one variable are reflected in the other variable because they both refer to the same data.

A struct is a value type. When a struct is created, the variable to which the struct is assigned holds the struct's actual data. When the struct is assigned to a new variable, it is copied. The new variable and the original variable therefore contain two separate copies of the same data. Changes made to one copy do not affect the other copy.

In general, classes are used to model more complex behavior, or data that is intended to be modified after a class object is created. Structs are best suited for small data structures that contain primarily data that is not intended to be modified after the struct is created.

For more information, see Classes, Objects, and Structs.

Example

In the following example, `CustomClass` in the `ProgrammingGuide` namespace has three members: an instance constructor, a property named `Number`, and a method named `Multiply`. The `Main` method in the `Program` class creates an instance (object) of `CustomClass`, and the object's method and property are accessed by using dot notation.

```
using System;

namespace ProgrammingGuide
{
   // Class definition.
   public class CustomClass
   {
      // Class members.
      //
      // Property.
      public int Number { get; set; }

      // Method.
      public int Multiply(int num)
      {
          return num * Number;
      }

      // Instance Constructor.
      public CustomClass()
      {
          Number = 0;
      }
   }

   // Another class definition that contains Main, the program entry point.
   class Program
   {
      static void Main(string[] args)
      {
         // Create an object of type CustomClass.
         CustomClass custClass = new CustomClass();

         // Set the value of the public property.
         custClass.Number = 27;

         // Call the public method.
         int result = custClass.Multiply(4);
         Console.WriteLine($"The result is {result}.");
      }
   }
}
// The example displays the following output:
//      The result is 108.
```

Encapsulation

Encapsulation is sometimes referred to as the first pillar or principle of object-oriented programming. According to the principle of encapsulation, a class or struct can specify how accessible each of its members is to code outside of the class or struct. Methods and variables that are not intended to be used from outside of the class or assembly can be hidden to limit the potential for coding errors or malicious exploits.

For more information about classes, see Classes and Objects.

Members

All methods, fields, constants, properties, and events must be declared within a type; these are called the *members* of the type. In C#, there are no global variables or methods as there are in some other languages. Even a program's entry point, the `Main` method, must be declared within a class or struct. The following list includes all the various kinds of members that may be declared in a class or struct.

- Fields
- Constants

- Properties
- Methods
- Constructors
- Events
- Finalizers
- Indexers
- Operators
- Nested Types

Accessibility

Some methods and properties are meant to be called or accessed from code outside your class or struct, known as *client code*. Other methods and properties might be only for use in the class or struct itself. It is important to limit the accessibility of your code so that only the intended client code can reach it. You specify how accessible your types and their members are to client code by using the access modifiers public, protected, internal, protected internal, private and private protected. The default accessibility is `private`. For more information, see Access Modifiers.

Inheritance

Classes (but not structs) support the concept of inheritance. A class that derives from another class (the *base class*) automatically contains all the public, protected, and internal members of the base class except its constructors and finalizers. For more information, see Inheritance and Polymorphism.

Classes may be declared as abstract, which means that one or more of their methods have no implementation. Although abstract classes cannot be instantiated directly, they can serve as base classes for other classes that provide the missing implementation. Classes can also be declared as sealed to prevent other classes from inheriting from them. For more information, see Abstract and Sealed Classes and Class Members.

Interfaces

Classes and structs can inherit multiple interfaces. To inherit from an interface means that the type implements all the methods defined in the interface. For more information, see Interfaces.

Generic Types

Classes and structs can be defined with one or more type parameters. Client code supplies the type when it creates an instance of the type. For example The List<T> class in the System.Collections.Generic namespace is defined with one type parameter. Client code creates an instance of a `List<string>` or `List<int>` to specify the type that the list will hold. For more information, see Generics.

Static Types

Classes (but not structs) can be declared as static. A static class can contain only static members and cannot be instantiated with the new keyword. One copy of the class is loaded into memory when the program loads, and its members are accessed through the class name. Both classes and structs can contain static members. For more information, see Static Classes and Static Class Members.

Nested Types

A class or struct can be nested within another class or struct. For more information, see Nested Types.

Partial Types

You can define part of a class, struct or method in one code file and another part in a separate code file. For more information, see Partial Classes and Methods.

Object Initializers

You can instantiate and initialize class or struct objects, and collections of objects, without explicitly calling their constructor. For more information, see Object and Collection Initializers.

Anonymous Types

In situations where it is not convenient or necessary to create a named class, for example when you are populating a list with data structures that you do not have to persist or pass to another method, you use anonymous types. For more information, see Anonymous Types.

Extension Methods

You can "extend" a class without creating a derived class by creating a separate type whose methods can be called as if they belonged to the original type. For more information, see Extension Methods.

Implicitly Typed Local Variables

Within a class or struct method, you can use implicit typing to instruct the compiler to determine the correct type at compile time. For more information, see Implicitly Typed Local Variables.

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

Interfaces (C# Programming Guide)

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

An interface contains definitions for a group of related functionalities that a class or a struct can implement.

By using interfaces, you can, for example, include behavior from multiple sources in a class. That capability is important in C# because the language doesn't support multiple inheritance of classes. In addition, you must use an interface if you want to simulate inheritance for structs, because they can't actually inherit from another struct or class.

You define an interface by using the interface keyword. as the following example shows.

```
interface IEquatable<T>
{
    bool Equals(T obj);
}
```

The name of the struct must be a valid C# identifier name. By convention, interface names begin with a capital `I`.

Any class or struct that implements the IEquatable<T> interface must contain a definition for an Equals method that matches the signature that the interface specifies. As a result, you can count on a class that implements `IEquatable<T>` to contain an `Equals` method with which an instance of the class can determine whether it's equal to another instance of the same class.

The definition of `IEquatable<T>` doesn't provide an implementation for `Equals`. The interface defines only the signature. In that way, an interface in C# is similar to an abstract class in which all the methods are abstract. However, a class or struct can implement multiple interfaces, but a class can inherit only a single class, abstract or not. Therefore, by using interfaces, you can include behavior from multiple sources in a class.

For more information about abstract classes, see Abstract and Sealed Classes and Class Members.

Interfaces can contain methods, properties, events, indexers, or any combination of those four member types. For links to examples, see Related Sections. An interface can't contain constants, fields, operators, instance constructors, finalizers, or types. Interface members are automatically public, and they can't include any access modifiers. Members also can't be static.

To implement an interface member, the corresponding member of the implementing class must be public, non-static, and have the same name and signature as the interface member.

When a class or struct implements an interface, the class or struct must provide an implementation for all of the members that the interface defines. The interface itself provides no functionality that a class or struct can inherit in the way that it can inherit base class functionality. However, if a base class implements an interface, any class that's derived from the base class inherits that implementation.

The following example shows an implementation of the IEquatable<T> interface. The implementing class, `Car`, must provide an implementation of the Equals method.

```
public class Car : IEquatable<Car>
{
    public string Make {get; set;}
    public string Model { get; set; }
    public string Year { get; set; }

    // Implementation of IEquatable<T> interface
    public bool Equals(Car car)
    {
        return this.Make == car.Make &&
               this.Model == car.Model &&
               this.Year == car.Year;
    }
}
```

Properties and indexers of a class can define extra accessors for a property or indexer that's defined in an interface. For example, an interface might declare a property that has a get accessor. The class that implements the interface can declare the same property with both a `get` and set accessor. However, if the property or indexer uses explicit implementation, the accessors must match. For more information about explicit implementation, see Explicit Interface Implementation and Interface Properties.

Interfaces can inherit from other interfaces. A class might include an interface multiple times through base classes that it inherits or through interfaces that other interfaces inherit. However, the class can provide an implementation of an interface only one time and only if the class declares the interface as part of the definition of the class (`class ClassName : InterfaceName`). If the interface is inherited because you inherited a base class that implements the interface, the base class provides the implementation of the members of the interface. However, the derived class can reimplement any virtual interface members instead of using the inherited implementation.

A base class can also implement interface members by using virtual members. In that case, a derived class can change the interface behavior by overriding the virtual members. For more information about virtual members, see Polymorphism.

Interfaces summary

An interface has the following properties:

- An interface is like an abstract base class. Any class or struct that implements the interface must implement all its members.
- An interface can't be instantiated directly. Its members are implemented by any class or struct that implements the interface.
- Interfaces can contain events, indexers, methods, and properties.
- Interfaces contain no implementation of methods.
- A class or struct can implement multiple interfaces. A class can inherit a base class and also implement one or more interfaces.

In this section

Explicit Interface Implementation
Explains how to create a class member that's specific to an interface.

How to: Explicitly Implement Interface Members
Provides an example of how to explicitly implement members of interfaces.

How to: Explicitly Implement Members of Two Interfaces
Provides an example of how to explicitly implement members of interfaces with inheritance.

Related Sections

- Interface Properties
- Indexers in Interfaces
- How to: Implement Interface Events
- Classes and Structs
- Inheritance
- Methods
- Polymorphism
- Abstract and Sealed Classes and Class Members
- Properties
- Events
- Indexers

featured book chapter

Interfaces in Learning C# 3.0: Master the Fundamentals of C# 3.0

See also

- C# Programming Guide
- Inheritance
- Identifier names

Enumeration types (C# Programming Guide)

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

An enumeration type (also named an enumeration or an enum) provides an efficient way to define a set of named integral constants that may be assigned to a variable. For example, assume that you have to define a variable whose value will represent a day of the week. There are only seven meaningful values which that variable will ever store. To define those values, you can use an enumeration type, which is declared by using the enum keyword.

```
enum Day { Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday };
enum Month : byte { Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec };
```

By default the underlying type of each element in the enum is int. You can specify another integral numeric type by using a colon, as shown in the previous example. For a full list of possible types, see enum (C# Reference).

You can verify the underlying numeric values by casting to the underlying type, as the following example shows.

```
Day today = Day.Monday;
int dayNumber =(int)today;
Console.WriteLine("{0} is day number #{1}.", today, dayNumber);

Month thisMonth = Month.Dec;
byte monthNumber = (byte)thisMonth;
Console.WriteLine("{0} is month number #{1}.", thisMonth, monthNumber);

// Output:
// Monday is day number #1.
// Dec is month number #11.
```

The following are advantages of using an enum instead of a numeric type:

- You clearly specify for client code which values are valid for the variable.
- In Visual Studio, IntelliSense lists the defined values.

When you do not specify values for the elements in the enumerator list, the values are automatically incremented by 1. In the previous example, `Day.Sunday` has a value of 0, `Day.Monday` has a value of 1, and so on. When you create a new `Day` object, it will have a default value of `Day.Sunday` (0) if you do not explicitly assign it a value. When you create an enum, select the most logical default value and give it a value of zero. That will cause all enums to have that default value if they are not explicitly assigned a value when they are created.

If the variable `meetingDay` is of type `Day`, then (without an explicit cast) you can only assign it one of the values defined by `Day`. And if the meeting day changes, you can assign a new value from `Day` to `meetingDay`:

```
Day meetingDay = Day.Monday;
//...
meetingDay = Day.Friday;
```

NOTE

It's possible to assign any arbitrary integer value to `meetingDay`. For example, this line of code does not produce an error: `meetingDay = (Day) 42`. However, you should not do this because the implicit expectation is that an enum variable will only hold one of the values defined by the enum. To assign an arbitrary value to a variable of an enumeration type is to introduce a high risk for errors.

You can assign any values to the elements in the enumerator list of an enumeration type, and you can also use computed values:

```
enum MachineState
{
    PowerOff = 0,
    Running = 5,
    Sleeping = 10,
    Hibernating = Sleeping + 5
}
```

Enumeration types as bit flags

You can use an enumeration type to define bit flags, which enables an instance of the enumeration type to store any combination of the values that are defined in the enumerator list. (Of course, some combinations may not be meaningful or allowed in your program code.)

You create a bit flags enum by applying the System.FlagsAttribute attribute and defining the values appropriately so that `AND`, `OR`, `NOT` and `XOR` bitwise operations can be performed on them. In a bit flags enum, include a named constant with a value of zero that means "no flags are set." Do not give a flag a value of zero if it does not mean "no flags are set".

In the following example, another version of the `Day` enum, which is named `Days`, is defined. `Days` has the `Flags` attribute, and each value is assigned the next greater power of 2. This enables you to create a `Days` variable whose value is `Days.Tuesday | Days.Thursday`.

```
[Flags]
enum Days
{
    None = 0x0,
    Sunday = 0x1,
    Monday = 0x2,
    Tuesday = 0x4,
    Wednesday = 0x8,
    Thursday = 0x10,
    Friday = 0x20,
    Saturday = 0x40
}
class MyClass
{
    Days meetingDays = Days.Tuesday | Days.Thursday;
}
```

To set a flag on an enum, use the bitwise `OR` operator as shown in the following example:

```
// Initialize with two flags using bitwise OR.
meetingDays = Days.Tuesday | Days.Thursday;

// Set an additional flag using bitwise OR.
meetingDays = meetingDays | Days.Friday;

Console.WriteLine("Meeting days are {0}", meetingDays);
// Output: Meeting days are Tuesday, Thursday, Friday

// Remove a flag using bitwise XOR.
meetingDays = meetingDays ^ Days.Tuesday;
Console.WriteLine("Meeting days are {0}", meetingDays);
// Output: Meeting days are Thursday, Friday
```

To determine whether a specific flag is set, use a bitwise `AND` operation, as shown in the following example:

```
// Test value of flags using bitwise AND.
bool test = (meetingDays & Days.Thursday) == Days.Thursday;
Console.WriteLine("Thursday {0} a meeting day.", test == true ? "is" : "is not");
// Output: Thursday is a meeting day.
```

For more information about what to consider when you define enumeration types with the System.FlagsAttribute attribute, see System.Enum.

Using the System.Enum methods to discover and manipulate enum values

All enums are instances of the System.Enum type. You cannot derive new classes from System.Enum, but you can use its methods to discover information about and manipulate values in an enum instance.

```
string s = Enum.GetName(typeof(Day), 4);
Console.WriteLine(s);

Console.WriteLine("The values of the Day Enum are:");
foreach (int i in Enum.GetValues(typeof(Day)))
    Console.WriteLine(i);

Console.WriteLine("The names of the Day Enum are:");
foreach (string str in Enum.GetNames(typeof(Day)))
    Console.WriteLine(str);
```

For more information, see System.Enum.

You can also create a new method for an enum by using an extension method. For more information, see How to: Create a New Method for an Enumeration.

See also

- System.Enum
- C# Programming Guide
- enum

Delegates (C# Programming Guide)

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

A delegate is a type that represents references to methods with a particular parameter list and return type. When you instantiate a delegate, you can associate its instance with any method with a compatible signature and return type. You can invoke (or call) the method through the delegate instance.

Delegates are used to pass methods as arguments to other methods. Event handlers are nothing more than methods that are invoked through delegates. You create a custom method, and a class such as a windows control can call your method when a certain event occurs. The following example shows a delegate declaration:

```
public delegate int PerformCalculation(int x, int y);
```

Any method from any accessible class or struct that matches the delegate type can be assigned to the delegate. The method can be either static or an instance method. This makes it possible to programmatically change method calls, and also plug new code into existing classes.

> **NOTE**
> In the context of method overloading, the signature of a method does not include the return value. But in the context of delegates, the signature does include the return value. In other words, a method must have the same return type as the delegate.

This ability to refer to a method as a parameter makes delegates ideal for defining callback methods. For example, a reference to a method that compares two objects could be passed as an argument to a sort algorithm. Because the comparison code is in a separate procedure, the sort algorithm can be written in a more general way.

Delegates Overview

Delegates have the following properties:

- Delegates are similar to C++ function pointers, but delegates are fully object-oriented, and unlike C++ pointers to member functions, delegates encapsulate both an object instance and a method.
- Delegates allow methods to be passed as parameters.
- Delegates can be used to define callback methods.
- Delegates can be chained together; for example, multiple methods can be called on a single event.
- Methods do not have to match the delegate type exactly. For more information, see Using Variance in Delegates.
- C# version 2.0 introduced the concept of Anonymous Methods, which allow code blocks to be passed as parameters in place of a separately defined method. C# 3.0 introduced lambda expressions as a more concise way of writing inline code blocks. Both anonymous methods and lambda expressions (in certain contexts) are compiled to delegate types. Together, these features are now known as anonymous functions. For more information about lambda expressions, see Anonymous Functions.

In This Section

- Using Delegates
- When to Use Delegates Instead of Interfaces (C# Programming Guide)
- Delegates with Named vs. Anonymous Methods
- Anonymous Methods
- Using Variance in Delegates
- How to: Combine Delegates (Multicast Delegates)
- How to: Declare, Instantiate, and Use a Delegate

C# Language Specification

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

Featured Book Chapters

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

Delegates and Events in Learning C# 3.0: Master the fundamentals of C# 3.0

See also

- Delegate
- C# Programming Guide
- Events

Arrays (C# Programming Guide)

1/24/2019 • 2 minutes to read • Edit Online

You can store multiple variables of the same type in an array data structure. You declare an array by specifying the type of its elements.

```
type[] arrayName;
```

The following example creates single-dimensional, multidimensional, and jagged arrays:

```
class TestArraysClass
{
    static void Main()
    {
        // Declare a single-dimensional array
        int[] array1 = new int[5];

        // Declare and set array element values
        int[] array2 = new int[] { 1, 3, 5, 7, 9 };

        // Alternative syntax
        int[] array3 = { 1, 2, 3, 4, 5, 6 };

        // Declare a two dimensional array
        int[,] multiDimensionalArray1 = new int[2, 3];

        // Declare and set array element values
        int[,] multiDimensionalArray2 = { { 1, 2, 3 }, { 4, 5, 6 } };

        // Declare a jagged array
        int[][] jaggedArray = new int[6][];

        // Set the values of the first array in the jagged array structure
        jaggedArray[0] = new int[4] { 1, 2, 3, 4 };
    }
}
```

Array Overview

An array has the following properties:

- An array can be Single-Dimensional, Multidimensional or Jagged.
- The number of dimensions and the length of each dimension are established when the array instance is created. These values can't be changed during the lifetime of the instance.
- The default values of numeric array elements are set to zero, and reference elements are set to null.
- A jagged array is an array of arrays, and therefore its elements are reference types and are initialized to `null`.
- Arrays are zero indexed: an array with `n` elements is indexed from `0` to `n-1`.
- Array elements can be of any type, including an array type.
- Array types are reference types derived from the abstract base type Array. Since this type implements IEnumerable and IEnumerable<T>, you can use foreach iteration on all arrays in C#.

Related Sections

- Arrays as Objects
- Using foreach with Arrays
- Passing Arrays as Arguments

C# Language Specification

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

See also

- C# Programming Guide
- Collections

Strings (C# Programming Guide)

12/11/2018 • 12 minutes to read • Edit Online

A string is an object of type String whose value is text. Internally, the text is stored as a sequential read-only collection of Char objects. There is no null-terminating character at the end of a C# string; therefore a C# string can contain any number of embedded null characters ('\0'). The Length property of a string represents the number of `Char` objects it contains, not the number of Unicode characters. To access the individual Unicode code points in a string, use the StringInfo object.

string vs. System.String

In C#, the `string` keyword is an alias for String. Therefore, `String` and `string` are equivalent, and you can use whichever naming convention you prefer. The `String` class provides many methods for safely creating, manipulating, and comparing strings. In addition, the C# language overloads some operators to simplify common string operations. For more information about the keyword, see string. For more information about the type and its methods, see String.

Declaring and Initializing Strings

You can declare and initialize strings in various ways, as shown in the following example:

```
// Declare without initializing.
string message1;

// Initialize to null.
string message2 = null;

// Initialize as an empty string.
// Use the Empty constant instead of the literal "".
string message3 = System.String.Empty;

//Initialize with a regular string literal.
string oldPath = "c:\\Program Files\\Microsoft Visual Studio 8.0";

// Initialize with a verbatim string literal.
string newPath = @"c:\Program Files\Microsoft Visual Studio 9.0";

// Use System.String if you prefer.
System.String greeting = "Hello World!";

// In local variables (i.e. within a method body)
// you can use implicit typing.
var temp = "I'm still a strongly-typed System.String!";

// Use a const string to prevent 'message4' from
// being used to store another string value.
const string message4 = "You can't get rid of me!";

// Use the String constructor only when creating
// a string from a char*, char[], or sbyte*. See
// System.String documentation for details.
char[] letters = { 'A', 'B', 'C' };
string alphabet = new string(letters);
```

Note that you do not use the new operator to create a string object except when initializing the string with an array of chars.

Initialize a string with the Empty constant value to create a new String object whose string is of zero length. The string literal representation of a zero-length string is "". By initializing strings with the Empty value instead of null, you can reduce the chances of a NullReferenceException occurring. Use the static IsNullOrEmpty(String) method to verify the value of a string before you try to access it.

Immutability of String Objects

String objects are *immutable*: they cannot be changed after they have been created. All of the String methods and C# operators that appear to modify a string actually return the results in a new string object. In the following example, when the contents of `s1` and `s2` are concatenated to form a single string, the two original strings are unmodified. The `+=` operator creates a new string that contains the combined contents. That new object is assigned to the variable `s1`, and the original object that was assigned to `s1` is released for garbage collection because no other variable holds a reference to it.

```
string s1 = "A string is more ";
string s2 = "than the sum of its chars.";

// Concatenate s1 and s2. This actually creates a new
// string object and stores it in s1, releasing the
// reference to the original object.
s1 += s2;

System.Console.WriteLine(s1);
// Output: A string is more than the sum of its chars.
```

Because a string "modification" is actually a new string creation, you must use caution when you create references to strings. If you create a reference to a string, and then "modify" the original string, the reference will continue to point to the original object instead of the new object that was created when the string was modified. The following code illustrates this behavior:

```
string s1 = "Hello ";
string s2 = s1;
s1 += "World";

System.Console.WriteLine(s2);
//Output: Hello
```

For more information about how to create new strings that are based on modifications such as search and replace operations on the original string, see How to: Modify String Contents.

Regular and Verbatim String Literals

Use regular string literals when you must embed escape characters provided by C#, as shown in the following example:

```
string columns = "Column 1\tColumn 2\tColumn 3";
//Output: Column 1        Column 2        Column 3

string rows = "Row 1\r\nRow 2\r\nRow 3";
/* Output:
  Row 1
  Row 2
  Row 3
*/

string title = "\"The \u00C6olean Harp\", by Samuel Taylor Coleridge";
//Output: "The Æolean Harp", by Samuel Taylor Coleridge
```

Use verbatim strings for convenience and better readability when the string text contains backslash characters, for example in file paths. Because verbatim strings preserve new line characters as part of the string text, they can be used to initialize multiline strings. Use double quotation marks to embed a quotation mark inside a verbatim string. The following example shows some common uses for verbatim strings:

```
string filePath = @"C:\Users\scoleridge\Documents\";
//Output: C:\Users\scoleridge\Documents\

string text = @"My pensive SARA ! thy soft cheek reclined
    Thus on mine arm, most soothing sweet it is
    To sit beside our Cot,...";
/* Output:
My pensive SARA ! thy soft cheek reclined
   Thus on mine arm, most soothing sweet it is
   To sit beside our Cot,...
*/

string quote = @"Her name was ""Sara.""";
//Output: Her name was "Sara."
```

String Escape Sequences

ESCAPE SEQUENCE	CHARACTER NAME	UNICODE ENCODING
\'	Single quote	0x0027
\"	Double quote	0x0022
\\	Backslash	0x005C
\0	Null	0x0000
\a	Alert	0x0007
\b	Backspace	0x0008
\f	Form feed	0x000C
\n	New line	0x000A
\r	Carriage return	0x000D
\t	Horizontal tab	0x0009

ESCAPE SEQUENCE	CHARACTER NAME	UNICODE ENCODING
\U	Unicode escape sequence for surrogate pairs.	\Unnnnnnnn
\u	Unicode escape sequence	\u0041 = "A"
\v	Vertical tab	0x000B
\x	Unicode escape sequence similar to "\u" except with variable length.	\x0041 or \x41 = "A"

> **NOTE**
> At compile time, verbatim strings are converted to ordinary strings with all the same escape sequences. Therefore, if you view a verbatim string in the debugger watch window, you will see the escape characters that were added by the compiler, not the verbatim version from your source code. For example, the verbatim string @"C:\files.txt" will appear in the watch window as "C:\\files.txt".

Format Strings

A format string is a string whose contents are determined dynamically at runtime. Format strings are created by embedding *interpolated expressions* or placeholders inside of braces within a string. Everything inside the braces (`{...}`) will be resolved to a value and output as a formatted string at runtime. There are two methods to create format strings: string interpolation and composite formatting.

String Interpolation

Available in C# 6.0 and later, *interpolated strings* are identified by the `$` special character and include interpolated expressions in braces. If you are new to string interpolation, see the String interpolation - C# interactive tutorial for a quick overview.

Use string interpolation to improve the readability and maintainability of your code. String interpolation achieves the same results as the `String.Format` method, but improves ease of use and inline clarity.

```
var jh = (firstName: "Jupiter", lastName: "Hammon", born: 1711, published: 1761);
Console.WriteLine($"{jh.firstName} {jh.lastName} was an African American poet born in {jh.born}.");
Console.WriteLine($"He was first published in {jh.published} at the age of {jh.published - jh.born}.");
Console.WriteLine($"He'd be over {Math.Round((2018d - jh.born) / 100d) * 100d} years old today.");

// Output:
// Jupiter Hammon was an African American poet born in 1711.
// He was first published in 1761 at the age of 50.
// He'd be over 300 years old today.
```

Composite Formatting

The String.Format utilizes placeholders in braces to create a format string. This example results in similar output to the string interpolation method used above.

```
var pw = (firstName: "Phillis", lastName: "Wheatley", born: 1753, published: 1773);
Console.WriteLine("{0} {1} was an African American poet born in {2}.", pw.firstName, pw.lastName, pw.born);
Console.WriteLine("She was first published in {0} at the age of {1}.", pw.published, pw.published - pw.born);
Console.WriteLine("She'd be over {0} years old today.", Math.Round((2018d - pw.born) / 100d) * 100d);

// Output:
// Phillis Wheatley was an African American poet born in 1753.
// She was first published in 1773 at the age of 20.
// She'd be over 300 years old today.
```

For more information on formatting .NET types see Formatting Types in .NET.

Substrings

A substring is any sequence of characters that is contained in a string. Use the Substring method to create a new string from a part of the original string. You can search for one or more occurrences of a substring by using the IndexOf method. Use the Replace method to replace all occurrences of a specified substring with a new string. Like the Substring method, Replace actually returns a new string and does not modify the original string. For more information, see How to: search strings and How to: Modify String Contents.

```
string s3 = "Visual C# Express";
System.Console.WriteLine(s3.Substring(7, 2));
// Output: "C#"

System.Console.WriteLine(s3.Replace("C#", "Basic"));
// Output: "Visual Basic Express"

// Index values are zero-based
int index = s3.IndexOf("C");
// index = 7
```

Accessing Individual Characters

You can use array notation with an index value to acquire read-only access to individual characters, as in the following example:

```
string s5 = "Printing backwards";

for (int i = 0; i < s5.Length; i++)
{
    System.Console.Write(s5[s5.Length - i - 1]);
}
// Output: "sdrawkcab gnitnirP"
```

If the String methods do not provide the functionality that you must have to modify individual characters in a string, you can use a StringBuilder object to modify the individual chars "in-place", and then create a new string to store the results by using the StringBuilder methods. In the following example, assume that you must modify the original string in a particular way and then store the results for future use:

```
string question = "hOW DOES mICROSOFT wORD DEAL WITH THE cAPS lOCK KEY?";
System.Text.StringBuilder sb = new System.Text.StringBuilder(question);

for (int j = 0; j < sb.Length; j++)
{
    if (System.Char.IsLower(sb[j]) == true)
        sb[j] = System.Char.ToUpper(sb[j]);
    else if (System.Char.IsUpper(sb[j]) == true)
        sb[j] = System.Char.ToLower(sb[j]);
}
// Store the new string.
string corrected = sb.ToString();
System.Console.WriteLine(corrected);
// Output: How does Microsoft Word deal with the Caps Lock key?
```

Null Strings and Empty Strings

An empty string is an instance of a System.String object that contains zero characters. Empty strings are used often in various programming scenarios to represent a blank text field. You can call methods on empty strings because they are valid System.String objects. Empty strings are initialized as follows:

```
string s = String.Empty;
```

By contrast, a null string does not refer to an instance of a System.String object and any attempt to call a method on a null string causes a NullReferenceException. However, you can use null strings in concatenation and comparison operations with other strings. The following examples illustrate some cases in which a reference to a null string does and does not cause an exception to be thrown:

```
static void Main()
{
    string str = "hello";
    string nullStr = null;
    string emptyStr = String.Empty;

    string tempStr = str + nullStr;
    // Output of the following line: hello
    Console.WriteLine(tempStr);

    bool b = (emptyStr == nullStr);
    // Output of the following line: False
    Console.WriteLine(b);

    // The following line creates a new empty string.
    string newStr = emptyStr + nullStr;

    // Null strings and empty strings behave differently. The following
    // two lines display 0.
    Console.WriteLine(emptyStr.Length);
    Console.WriteLine(newStr.Length);
    // The following line raises a NullReferenceException.
    //Console.WriteLine(nullStr.Length);

    // The null character can be displayed and counted, like other chars.
    string s1 = "\x0" + "abc";
    string s2 = "abc" + "\x0";
    // Output of the following line: * abc*
    Console.WriteLine("*" + s1 + "*");
    // Output of the following line: *abc *
    Console.WriteLine("*" + s2 + "*");
    // Output of the following line: 4
    Console.WriteLine(s2.Length);
}
```

Using StringBuilder for Fast String Creation

String operations in .NET are highly optimized and in most cases do not significantly impact performance. However, in some scenarios such as tight loops that are executing many hundreds or thousands of times, string operations can affect performance. The StringBuilder class creates a string buffer that offers better performance if your program performs many string manipulations. The StringBuilder string also enables you to reassign individual characters, something the built-in string data type does not support. This code, for example, changes the content of a string without creating a new string:

```
System.Text.StringBuilder sb = new System.Text.StringBuilder("Rat: the ideal pet");
sb[0] = 'C';
System.Console.WriteLine(sb.ToString());
System.Console.ReadLine();

//Outputs Cat: the ideal pet
```

In this example, a StringBuilder object is used to create a string from a set of numeric types:

```
class TestStringBuilder
{
    static void Main()
    {
        System.Text.StringBuilder sb = new System.Text.StringBuilder();

        // Create a string composed of numbers 0 - 9
        for (int i = 0; i < 10; i++)
        {
            sb.Append(i.ToString());
        }
        System.Console.WriteLine(sb);  // displays 0123456789

        // Copy one character of the string (not possible with a System.String)
        sb[0] = sb[9];

        System.Console.WriteLine(sb);  // displays 9123456789
    }
}
```

Strings, Extension Methods and LINQ

Because the String type implements IEnumerable<T>, you can use the extension methods defined in the Enumerable class on strings. To avoid visual clutter, these methods are excluded from IntelliSense for the String type, but they are available nevertheless. You can also use LINQ query expressions on strings. For more information, see LINQ and Strings.

Related Topics

TOPIC	DESCRIPTION
How to: Modify String Contents	Illustrates techniques to transform strings and modify the contents of strings.
How to: Compare Strings	Shows how to perform ordinal and culture specific comparisons of strings.
How to: Concatenate Multiple Strings	Demonstrates various ways to join multiple strings into one.
How to: Parse Strings Using String.Split	Contains code examples that illustrate how to use the `String.Split` method to parse strings.
How to: Search Strings	Explains how to use search for specific text or patterns in strings.
How to: Determine Whether a String Represents a Numeric Value	Shows how to safely parse a string to see whether it has a valid numeric value.
String interpolation	Describes the string interpolation feature that provides a convenient syntax to format strings.
Basic String Operations	Provides links to topics that use System.String and System.Text.StringBuilder methods to perform basic string operations.
Parsing Strings	Describes how to convert string representations of .NET base types to instances of the corresponding types.

TOPIC	DESCRIPTION
Parsing Date and Time Strings in .NET	Shows how to convert a string such as "01/24/2008" to a System.DateTime object.
Comparing Strings	Includes information about how to compare strings and provides examples in C# and Visual Basic.
Using the StringBuilder Class	Describes how to create and modify dynamic string objects by using the StringBuilder class.
LINQ and Strings	Provides information about how to perform various string operations by using LINQ queries.
C# Programming Guide	Provides links to topics that explain programming constructs in C#.

Indexers (C# Programming Guide)

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

Indexers allow instances of a class or struct to be indexed just like arrays. The indexed value can be set or retrieved without explicitly specifying a type or instance member. Indexers resemble properties except that their accessors take parameters.

The following example defines a generic class with simple get and set accessor methods to assign and retrieve values. The `Program` class creates an instance of this class for storing strings.

```
using System;

class SampleCollection<T>
{
   // Declare an array to store the data elements.
   private T[] arr = new T[100];

   // Define the indexer to allow client code to use [] notation.
   public T this[int i]
   {
      get { return arr[i]; }
      set { arr[i] = value; }
   }
}

class Program
{
   static void Main()
   {
      var stringCollection = new SampleCollection<string>();
      stringCollection[0] = "Hello, World";
      Console.WriteLine(stringCollection[0]);
   }
}
// The example displays the following output:
//       Hello, World.
```

NOTE

For more examples, see Related Sections.

Expression Body Definitions

It is common for an indexer's get or set accessor to consist of a single statement that either returns or sets a value. Expression-bodied members provide a simplified syntax to support this scenario. Starting with C# 6, a read-only indexer can be implemented as an expression-bodied member, as the following example shows.

```
using System;

class SampleCollection<T>
{
   // Declare an array to store the data elements.
   private T[] arr = new T[100];
   int nextIndex = 0;

   // Define the indexer to allow client code to use [] notation.
   public T this[int i] => arr[i];

   public void Add(T value)
   {
      if (nextIndex >= arr.Length)
         throw new IndexOutOfRangeException($"The collection can hold only {arr.Length} elements.");
      arr[nextIndex++] = value;
   }
}

class Program
{
   static void Main()
   {
      var stringCollection = new SampleCollection<string>();
      stringCollection.Add("Hello, World");
      System.Console.WriteLine(stringCollection[0]);
   }
}
// The example displays the following output:
//       Hello, World.
```

Note that `=>` introduces the expression body, and that the `get` keyword is not used.

Starting with C# 7.0, both the get and set accessor can be an implemented as expression-bodied members. In this case, both `get` and `set` keywords must be used. For example:

```
using System;

class SampleCollection<T>
{
   // Declare an array to store the data elements.
   private T[] arr = new T[100];

   // Define the indexer to allow client code to use [] notation.
   public T this[int i]
   {
      get => arr[i];
      set => arr[i] = value;
   }
}

class Program
{
   static void Main()
   {
      var stringCollection = new SampleCollection<string>();
      stringCollection[0] = "Hello, World.";
      Console.WriteLine(stringCollection[0]);
   }
}
// The example displays the following output:
//       Hello, World.
```

Indexers Overview

- Indexers enable objects to be indexed in a similar manner to arrays.
- A `get` accessor returns a value. A `set` accessor assigns a value.
- The this keyword is used to define the indexer.
- The value keyword is used to define the value being assigned by the `set` indexer.
- Indexers do not have to be indexed by an integer value; it is up to you how to define the specific look-up mechanism.
- Indexers can be overloaded.
- Indexers can have more than one formal parameter, for example, when accessing a two-dimensional array.

Related Sections

- Using Indexers
- Indexers in Interfaces
- Comparison Between Properties and Indexers
- Restricting Accessor Accessibility

C# Language Specification

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

See also

- C# Programming Guide
- Properties

Events (C# Programming Guide)

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

Events enable a class or object to notify other classes or objects when something of interest occurs. The class that sends (or *raises*) the event is called the *publisher* and the classes that receive (or *handle*) the event are called *subscribers*.

In a typical C# Windows Forms or Web application, you subscribe to events raised by controls such as buttons and list boxes. You can use the Visual C# integrated development environment (IDE) to browse the events that a control publishes and select the ones that you want to handle. The IDE provides an easy way to automatically add an empty event handler method and the code to subscribe to the event. For more information, see How to: Subscribe to and Unsubscribe from Events.

Events Overview

Events have the following properties:

- The publisher determines when an event is raised; the subscribers determine what action is taken in response to the event.
- An event can have multiple subscribers. A subscriber can handle multiple events from multiple publishers.
- Events that have no subscribers are never raised.
- Events are typically used to signal user actions such as button clicks or menu selections in graphical user interfaces.
- When an event has multiple subscribers, the event handlers are invoked synchronously when an event is raised. To invoke events asynchronously, see Calling Synchronous Methods Asynchronously.
- In the .NET Framework class library, events are based on the EventHandler delegate and the EventArgs base class.

Related Sections

For more information, see:

- How to: Subscribe to and Unsubscribe from Events
- How to: Publish Events that Conform to .NET Framework Guidelines
- How to: Raise Base Class Events in Derived Classes
- How to: Implement Interface Events
- How to: Use a Dictionary to Store Event Instances
- How to: Implement Custom Event Accessors

C# Language Specification

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

Featured Book Chapters

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

Delegates and Events in Learning C# 3.0: Master the fundamentals of C# 3.0

See also

- EventHandler
- C# Programming Guide
- Delegates
- Creating Event Handlers in Windows Forms

Generics (C# Programming Guide)

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

Generics were added to version 2.0 of the C# language and the common language runtime (CLR). Generics introduce to the .NET Framework the concept of type parameters, which make it possible to design classes and methods that defer the specification of one or more types until the class or method is declared and instantiated by client code. For example, by using a generic type parameter T you can write a single class that other client code can use without incurring the cost or risk of runtime casts or boxing operations, as shown here:

```
// Declare the generic class.
public class GenericList<T>
{
    public void Add(T input) { }
}
class TestGenericList
{
    private class ExampleClass { }
    static void Main()
    {
        // Declare a list of type int.
        GenericList<int> list1 = new GenericList<int>();
        list1.Add(1);

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

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

Generics Overview

- Use generic types to maximize code reuse, type safety, and performance.
- The most common use of generics is to create collection classes.
- The .NET Framework class library contains several new generic collection classes in the System.Collections.Generic namespace. These should be used whenever possible instead of classes such as ArrayList in the System.Collections namespace.
- You can create your own generic interfaces, classes, methods, events and delegates.
- Generic classes may be constrained to enable access to methods on particular data types.
- Information on the types that are used in a generic data type may be obtained at run-time by using reflection.

Related Sections

For more information:

- Introduction to Generics

- Benefits of Generics
- Generic Type Parameters
- Constraints on Type Parameters
- Generic Classes
- Generic Interfaces
- Generic Methods
- Generic Delegates
- Differences Between C++ Templates and C# Generics
- Generics and Reflection
- Generics in the Run Time

C# Language Specification

For more information, see the C# Language Specification.

See also

- System.Collections.Generic
- C# Programming Guide
- Types
- <typeparam>
- <typeparamref>
- Generics in .NET

Namespaces (C# Programming Guide)

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

Namespaces are heavily used in C# programming in two ways. First, the .NET Framework uses namespaces to organize its many classes, as follows:

```
System.Console.WriteLine("Hello World!");
```

`System` is a namespace and `Console` is a class in that namespace. The `using` keyword can be used so that the complete name is not required, as in the following example:

```
using System;
```

```
Console.WriteLine("Hello");
Console.WriteLine("World!");
```

For more information, see the using Directive.

Second, declaring your own namespaces can help you control the scope of class and method names in larger programming projects. Use the namespace keyword to declare a namespace, as in the following example:

```
namespace SampleNamespace
{
    class SampleClass
    {
        public void SampleMethod()
        {
            System.Console.WriteLine(
              "SampleMethod inside SampleNamespace");
        }
    }
}
```

The name of the namespace must be a valid C# identifier name.

Namespaces Overview

Namespaces have the following properties:

- They organize large code projects.
- They are delimited by using the `.` operator.
- The `using` directive obviates the requirement to specify the name of the namespace for every class.
- The `global` namespace is the "root" namespace: `global::System` will always refer to the .NET System namespace.

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

- Using Namespaces
- How to: Use the Global Namespace Alias
- How to: Use the My Namespace
- C# Programming Guide
- Identifier names
- Namespace Keywords
- using Directive
- :: Operator
- . Operator

Nullable types (C# Programming Guide)

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

Nullable types are instances of the System.Nullable<T> struct. Nullable types can represent all the values of an underlying type `T`, and an additional null value. The underlying type `T` can be any non-nullable value type. `T` cannot be a reference type.

For example, you can assign `null` or any integer value from Int32.MinValue to Int32.MaxValue to a `Nullable<int>` and true, false, or `null` to a `Nullable<bool>`.

You use a nullable type when you need to represent the undefined value of an underlying type. A Boolean variable can have only two values: true and false. There is no "undefined" value. In many programming applications, most notably database interactions, a variable value can be undefined or missing. For example, a field in a database may contain the values true or false, or it may contain no value at all. You use a `Nullable<bool>` type in that case.

Nullable types have the following characteristics:

- Nullable types represent value-type variables that can be assigned the `null` value. You cannot create a nullable type based on a reference type. (Reference types already support the `null` value.)
- The syntax `T?` is shorthand for `Nullable<T>`. The two forms are interchangeable.
- Assign a value to a nullable type just as you would for an underlying value type: `int? x = 10;` or `double? d = 4.108;`. You also can assign the `null` value: `int? x = null;`.
- Use the Nullable<T>.HasValue and Nullable<T>.Value readonly properties to test for null and retrieve the value, as shown in the following example: `if (x.HasValue) y = x.Value;`
  - The HasValue property returns `true` if the variable contains a value, or `false` if it's `null`.
  - The Value property returns a value if HasValue returns `true`. Otherwise, an InvalidOperationException is thrown.
- You can also use the `==` and `!=` operators with a nullable type, as shown in the following example: `if (x != null) y = x.Value;`. If `a` and `b` are both null, `a == b` evaluates to `true`.
- Beginning with C# 7.0, you can use pattern matching to both examine and get a value of a nullable type: `if (x is int valueOfX) y = valueOfX;`.
- The default value of `T?` is an instance whose HasValue property returns `false`.
- Use the GetValueOrDefault() method to return either the assigned value, or the default value of the underlying value type if the value of the nullable type is `null`.
- Use the GetValueOrDefault(T) method to return either the assigned value, or the provided default value if the value of the nullable type is `null`.
- Use the null-coalescing operator, `??`, to assign a value to an underlying type based on a value of the nullable type: `int? x = null; int y = x ?? -1;`. In the example, since `x` is null, the result value of `y` is `-1`.
- If a user-defined conversion is defined between two data types, the same conversion can also be used with the nullable versions of these data types.
- Nested nullable types are not allowed. The following line doesn't compile: `Nullable<Nullable<int>> n;`

For more information, see the Using nullable types and How to: Identify a nullable type topics.

See also

- System.Nullable<T>
- System.Nullable
- ?? Operator
- C# Programming Guide
- C# Guide
- C# Reference
- Nullable Value Types (Visual Basic)

Unsafe Code and Pointers (C# Programming Guide)

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

To maintain type safety and security, C# does not support pointer arithmetic, by default. However, by using the unsafe keyword, you can define an unsafe context in which pointers can be used. For more information about pointers, see the topic Pointer types.

> **NOTE**
>
> In the common language runtime (CLR), unsafe code is referred to as unverifiable code. Unsafe code in C# is not necessarily dangerous; it is just code whose safety cannot be verified by the CLR. The CLR will therefore only execute unsafe code if it is in a fully trusted assembly. If you use unsafe code, it is your responsibility to ensure that your code does not introduce security risks or pointer errors.

Unsafe Code Overview

Unsafe code has the following properties:

- Methods, types, and code blocks can be defined as unsafe.
- In some cases, unsafe code may increase an application's performance by removing array bounds checks.
- Unsafe code is required when you call native functions that require pointers.
- Using unsafe code introduces security and stability risks.
- In order for C# to compile unsafe code, the application must be compiled with /unsafe.

Related Sections

For more information, see:

- Pointer types
- Fixed Size Buffers
- How to: Use Pointers to Copy an Array of Bytes
- unsafe

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

XML Documentation Comments (C# Programming Guide)

1/29/2019 • 2 minutes to read • Edit Online

In Visual C# you can create documentation for your code by including XML elements in special comment fields (indicated by triple slashes) in the source code directly before the code block to which the comments refer, for example:

```
/// <summary>
///  This class performs an important function.
/// </summary>
public class MyClass {}
```

When you compile with the /doc option, the compiler will search for all XML tags in the source code and create an XML documentation file. To create the final documentation based on the compiler-generated file, you can create a custom tool or use a tool such as DocFX or Sandcastle.

To refer to XML elements (for example, your function processes specific XML elements that you want to describe in an XML documentation comment), you can use the standard quoting mechanism (`<` and `>`). To refer to generic identifiers in code reference (`cref`) elements, you can use either the escape characters (for example, `cref="List&lt;T&gt;"`) or braces (`cref="List{T}"`). As a special case, the compiler parses the braces as angle brackets to make the documentation comment less cumbersome to author when referring to generic identifiers.

> **NOTE**
> The XML documentation comments are not metadata; they are not included in the compiled assembly and therefore they are not accessible through reflection.

In This Section

- Recommended Tags for Documentation Comments
- Processing the XML File
- Delimiters for Documentation Tags
- How to: Use the XML Documentation Features

Related Sections

For more information, see:

- /doc (Process Documentation Comments)

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

Exceptions and Exception Handling (C# Programming Guide)

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

The C# language's exception handling features help you deal with any unexpected or exceptional situations that occur when a program is running. Exception handling uses the `try`, `catch`, and `finally` keywords to try actions that may not succeed, to handle failures when you decide that it is reasonable to do so, and to clean up resources afterward. Exceptions can be generated by the common language runtime (CLR), by the .NET Framework or any third-party libraries, or by application code. Exceptions are created by using the `throw` keyword.

In many cases, an exception may be thrown not by a method that your code has called directly, but by another method further down in the call stack. When this happens, the CLR will unwind the stack, looking for a method with a `catch` block for the specific exception type, and it will execute the first such `catch` block that if finds. If it finds no appropriate `catch` block anywhere in the call stack, it will terminate the process and display a message to the user.

In this example, a method tests for division by zero and catches the error. Without the exception handling, this program would terminate with a **DivideByZeroException was unhandled** error.

```
class ExceptionTest
{
    static double SafeDivision(double x, double y)
    {
        if (y == 0)
            throw new System.DivideByZeroException();
        return x / y;
    }
    static void Main()
    {
        // Input for test purposes. Change the values to see
        // exception handling behavior.
        double a = 98, b = 0;
        double result = 0;

        try
        {
            result = SafeDivision(a, b);
            Console.WriteLine("{0} divided by {1} = {2}", a, b, result);
        }
        catch (DivideByZeroException e)
        {
            Console.WriteLine("Attempted divide by zero.");
        }
    }
}
```

Exceptions Overview

Exceptions have the following properties:

- Exceptions are types that all ultimately derive from `System.Exception`.
- Use a `try` block around the statements that might throw exceptions.
- Once an exception occurs in the `try` block, the flow of control jumps to the first associated exception

handler that is present anywhere in the call stack. In C#, the `catch` keyword is used to define an exception handler.

- If no exception handler for a given exception is present, the program stops executing with an error message.
- Do not catch an exception unless you can handle it and leave the application in a known state. If you catch `System.Exception`, rethrow it using the `throw` keyword at the end of the `catch` block.
- If a `catch` block defines an exception variable, you can use it to obtain more information about the type of exception that occurred.
- Exceptions can be explicitly generated by a program by using the `throw` keyword.
- Exception objects contain detailed information about the error, such as the state of the call stack and a text description of the error.
- Code in a `finally` block is executed even if an exception is thrown. Use a `finally` block to release resources, for example to close any streams or files that were opened in the `try` block.
- Managed exceptions in the .NET Framework are implemented on top of the Win32 structured exception handling mechanism. For more information, see Structured Exception Handling (C/C++) and A Crash Course on the Depths of Win32 Structured Exception Handling.

Related Sections

See the following topics for more information about exceptions and exception handling:

- Using Exceptions
- Exception Handling
- Creating and Throwing Exceptions
- Compiler-Generated Exceptions
- How to: Handle an Exception Using try/catch (C# Programming Guide)
- How to: Execute Cleanup Code Using finally

C# Language Specification

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

See also

- SystemException
- C# Programming Guide
- C# Keywords
- throw
- try-catch
- try-finally
- try-catch-finally
- Exceptions

File System and the Registry (C# Programming Guide)

12/11/2018 • 2 minutes to read • Edit Online

The following topics show how to use C# and the .NET Framework to perform various basic operations on files, folders, and the Registry.

In This Section

TITLE	DESCRIPTION
How to: Iterate Through a Directory Tree	Shows how to manually iterate through a directory tree.
How to: Get Information About Files, Folders, and Drives	Shows how to retrieve information such as creation times and size, about files, folders and drives.
How to: Create a File or Folder	Shows how to create a new file or folder.
How to: Copy, Delete, and Move Files and Folders (C# Programming Guide)	Shows how to copy, delete and move files and folders.
How to: Provide a Progress Dialog Box for File Operations	Shows how to display a standard Windows progress dialog for certain file operations.
How to: Write to a Text File	Shows how to write to a text file.
How to: Read From a Text File	Shows how to read from a text file.
How to: Read a Text File One Line at a Time	Shows how to retrieve text from a file one line at a time.
How to: Create a Key In the Registry	Shows how to write a key to the system registry.

Related Sections

File and Stream I/O

How to: Copy, Delete, and Move Files and Folders (C# Programming Guide)

C# Programming Guide

Files, Folders and Drives

System.IO

Interoperability (C# Programming Guide)

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

Interoperability enables you to preserve and take advantage of existing investments in unmanaged code. Code that runs under the control of the common language runtime (CLR) is called *managed code*, and code that runs outside the CLR is called *unmanaged code*. COM, COM+, C++ components, ActiveX components, and Microsoft Win32 API are examples of unmanaged code.

The .NET Framework enables interoperability with unmanaged code through platform invoke services, the System.Runtime.InteropServices namespace, C++ interoperability, and COM interoperability (COM interop).

In This Section

Interoperability Overview
Describes methods to interoperate between C# managed code and unmanaged code.

How to: Access Office Interop Objects by Using Visual C# Features
Describes features that are introduced in Visual C# to facilitate Office programming.

How to: Use Indexed Properties in COM Interop Programming
Describes how to use indexed properties to access COM properties that have parameters.

How to: Use Platform Invoke to Play a Wave File
Describes how to use platform invoke services to play a .wav sound file on the Windows operating system.

Walkthrough: Office Programming
Shows how to create an Excel workbook and a Word document that contains a link to the workbook.

Example COM Class
Demonstrates how to expose a C# class as a COM object.

C# Language Specification

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

See also

- Marshal.ReleaseComObject
- C# Programming Guide
- Interoperating with Unmanaged Code
- Walkthrough: Office Programming