

Contents

Tour of C#

[Program Structure](#)

[Types and Variables](#)

[Expressions](#)

[Statements](#)

[Classes and objects](#)

[Structs](#)

[Arrays](#)

[Interfaces](#)

[Enums](#)

[Delegates](#)

[Attributes](#)

A Tour of the C# Language

4/11/2019 • 4 minutes to read • [Edit Online](#)

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++, Java, and JavaScript programmers. This tour provides an overview of the major components of the language. If you want to explore the language through interactive examples, try our [introduction to C#](#) tutorials.

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 support directly 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 unreachable 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.

Hello world

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

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

C# source files typically have the file extension `.cs`. Assuming that the "Hello, World" program is stored in the file `hello.cs`, the program might be compiled using the command line:

```
csc hello.cs
```

which produces an executable assembly named `hello.exe`. The output produced by this application when it is run is:

Hello, World

IMPORTANT

The `csc` command compiles for the full framework, and may not be available on all platforms.

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

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

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

There's a lot more to learn about C#. The following topics provide an overview of the elements of the C# language. These overviews will provide basic information about all elements of the language and give you the information necessary to dive deeper into elements of the C# language:

- [Program Structure](#)
 - Learn the key organizational concepts in the C# language: **programs**, **namespaces**, **types**, **members**, and **assemblies**.
- [Types and Variables](#)
 - Learn about **value types**, **reference types**, and **variables** in the C# language.
- [Expressions](#)
 - **Expressions** are constructed from **operands** and **operators**. Expressions produce a value.
- [Statements](#)
 - You use **statements** to express the actions of a program.
- [Classes and objects](#)
 - **Classes** are the most fundamental of C#'s types. **Objects** are instances of a class. Classes are built using **members**, which are also covered in this topic.
- [Structs](#)
 - **Structs** are data structures that, unlike classes, are value types.
- [Arrays](#)
 - An **array** is a data structure that contains a number of variables that are accessed through computed indices.
- [Interfaces](#)
 - An **interface** defines a contract that can be implemented by classes and structs. An interface can contain methods, properties, events, and indexers. An interface does not provide implementations of the members it defines—it merely specifies the members that must be supplied by classes or structs that implement the interface.
- [Enums](#)

- An **enum type** is a distinct value type with a set of named constants.
- [Delegates](#)
 - A **delegate type** represents references to methods with a particular parameter list and return type. Delegates make it possible to treat methods as entities that can be assigned to variables and passed as parameters. Delegates are similar to the concept of function pointers found in some other languages, but unlike function pointers, delegates are object-oriented and type-safe.
- [Attributes](#)
 - **Attributes** enable programs to specify additional declarative information about types, members, and other entities.

NEXT

Program Structure

5/15/2019 • 3 minutes to read • [Edit Online](#)

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

The example declares a class named `Stack` in a namespace called `Acme.Collections`:

```
using System;
namespace Acme.Collections
{
    public class Stack
    {
        Entry top;
        public void Push(object data)
        {
            top = new Entry(top, data);
        }

        public object Pop()
        {
            if (top == null)
            {
                throw new InvalidOperationException();
            }
            object result = top.data;
            top = top.next;
            return result;
        }

        class Entry
        {
            public Entry next;
            public object data;
            public Entry(Entry next, object data)
            {
                this.next = next;
                this.data = data;
            }
        }
    }
}
```

The fully qualified name of this class is `Acme.Collections.Stack`. The class contains several members: a field named `top`, two methods named `Push` and `Pop`, and a nested class named `Entry`. The `Entry` class further contains three members: a field named `next`, a field named `data`, and a constructor. Assuming that the source code of the example is stored in the file `acme.cs`, the command line

```
csc /t:library acme.cs
```

compiles the example as a library (code without a `Main` entry point) and produces an assembly named `acme.dll`.

IMPORTANT

The examples above use `csc` as the command line C# compiler. This compiler is a Windows executable. To use C# across other platforms, you should use the tools for .NET Core. The .NET Core ecosystem uses the `dotnet` CLI to manage command line builds. This includes managing dependencies, and invoking the C# compiler. See [this tutorial](#) for a full description of those tools on the platforms supported by .NET Core.

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

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

```
using System;
using Acme.Collections;
class Example
{
    static void Main()
    {
        Stack s = new Stack();
        s.Push(1);
        s.Push(10);
        s.Push(100);
        Console.WriteLine(s.Pop());
        Console.WriteLine(s.Pop());
        Console.WriteLine(s.Pop());
    }
}
```

If the program is stored in the file `example.cs`, when `example.cs` is compiled, the `acme.dll` assembly can be referenced using the compiler's `/r` option:

```
csc /r:acme.dll example.cs
```

This creates an executable assembly named `example.exe`, which, when run, produces the output:

```
100
10
1
```

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

Types and variables

5/18/2019 • 5 minutes to read • [Edit Online](#)

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 thus possible for operations on one variable to affect the object referenced by the other variable. With value types, the variables each have their own copy of the data, and it is not possible for operations on one to affect the other (except in the case of `ref` and `out` parameter variables).

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

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

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

For more information about numeric types, see [Integral types table](#) and [Floating-point types table](#).

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.

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

A `class` type defines a data structure that contains data members (fields) and function members (methods, properties, and others). Class types support single inheritance and polymorphism, mechanisms whereby derived classes can extend and specialize base classes.

A `struct` type is similar to a class type in that it represents a structure with data members and function members. However, unlike classes, structs are value types and do not typically require heap allocation. Struct types do not support user-specified inheritance, and all struct types implicitly inherit from type `object`.

An `interface` type defines a contract as a named set of public function members. A `class` or `struct` that implements an `interface` must provide implementations of the interface's function members. An `interface` may inherit from multiple base interfaces, and a `class` or `struct` may implement multiple interfaces.

A `delegate` type represents references to methods with a particular parameter list and return type. Delegates make it possible to treat methods as entities that can be assigned to variables and passed as parameters. Delegates are analogous to function types provided by functional languages. They are also similar to the concept of function pointers found in some other languages, but unlike function pointers, delegates are object-oriented and type-safe.

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

An `enum` type is a distinct type with named constants. Every `enum` type has an underlying type, which must be one of the eight integral types. The set of values of an `enum` type is the same as the set of values of the underlying type.

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

Nullable value types also do not have to be declared before they can be used. For each non-nullable value type `T` there is a corresponding nullable value type `T?`, which can hold an additional value, `null`. For instance, `int?` is a type that can hold any 32-bit integer or the value `null`.

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

```
using System;
class BoxingExample
{
    static void Main()
    {
        int i = 123;
        object o = i;    // Boxing
        int j = (int)o;  // Unboxing
    }
}
```

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

C#'s unified type system effectively means that value types can become objects "on demand." Because of the unification, general-purpose libraries that use type `object` can be used with both reference types and value types.

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

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

[PREVIOUS](#)[NEXT](#)

Expressions

4/26/2019 • 2 minutes to read • [Edit Online](#)

Expressions are constructed from *operands* and *operators*. The operators of an expression indicate which operations to apply to the operands. Examples of operators include `+`, `-`, `*`, `/`, and `new`. Examples of operands include literals, fields, local variables, and expressions.

When an expression contains multiple operators, the *precedence* of the operators controls the order in which the individual operators are evaluated. For example, the expression `x + y * z` is evaluated as `x + (y * z)` because the `*` operator has higher precedence than the `+` operator.

When an operand occurs between two operators with the same precedence, the *associativity* of the operators controls the order in which the operations are performed:

- Except for the assignment operators, all binary operators are *left-associative*, meaning that operations are performed from left to right. For example, `x + y + z` is evaluated as `(x + y) + z`.
- The assignment operators and the conditional operator (`?:`) are *right-associative*, meaning that operations are performed from right to left. For example, `x = y = z` is evaluated as `x = (y = z)`.

Precedence and associativity can be controlled using parentheses. For example, `x + y * z` first multiplies `y` by `z` and then adds the result to `x`, but `(x + y) * z` first adds `x` and `y` and then multiplies the result by `z`.

Most operators can be *overloaded*. Operator overloading permits user-defined operator implementations to be specified for operations where one or both of the operands are of a user-defined class or struct type.

C# provides a number of operators to perform [arithmetic](#), [logical](#), [bitwise and shift](#) operations and [equality](#) and [order](#) comparisons.

For the complete list of C# operators ordered by precedence level, see [C# operators](#).

[PREVIOUS](#)[NEXT](#)

Statements

5/15/2019 • 3 minutes to read • [Edit Online](#)

The actions of a program are expressed using *statements*. C# supports several different kinds of statements, a number of which are defined in terms of embedded statements.

A *block* permits multiple statements to be written in contexts where a single statement is allowed. A block consists of a list of statements written between the delimiters `{` and `}`.

Declaration statements are used to declare local variables and constants.

Expression statements are used to evaluate expressions. Expressions that can be used as statements include method invocations, object allocations using the `new` operator, assignments using `=` and the compound assignment operators, increment and decrement operations using the `++` and `--` operators and `await` expressions.

Selection statements are used to select one of a number of possible statements for execution based on the value of some expression. In this group are the `if` and `switch` statements.

Iteration statements are used to execute repeatedly an embedded statement. In this group are the `while`, `do`, `for`, and `foreach` statements.

Jump statements are used to transfer control. In this group are the `break`, `continue`, `goto`, `throw`, `return`, and `yield` statements.

The `try ... catch` statement is used to catch exceptions that occur during execution of a block, and the `try ... finally` statement is used to specify finalization code that is always executed, whether an exception occurred or not.

The `checked` and `unchecked` statements are used to control the overflow-checking context for integral-type arithmetic operations and conversions.

The `lock` statement is used to obtain the mutual-exclusion lock for a given object, execute a statement, and then release the lock.

The `using` statement is used to obtain a resource, execute a statement, and then dispose of that resource.

The following lists the kinds of statements that can be used, and provides an example for each.

- Local variable declaration:

```
static void Declarations(string[] args)
{
    int a;
    int b = 2, c = 3;
    a = 1;
    Console.WriteLine(a + b + c);
}
```

- Local constant declaration:

```
static void ConstantDeclarations(string[] args)
{
    const float pi = 3.1415927f;
    const int r = 25;
    Console.WriteLine(pi * r * r);
}
```

- Expression statement:

```
static void Expressions(string[] args)
{
    int i;
    i = 123;           // Expression statement
    Console.WriteLine(i); // Expression statement
    i++;              // Expression statement
    Console.WriteLine(i); // Expression statement
}
```

- `if` statement:

```
static void IfStatement(string[] args)
{
    if (args.Length == 0)
    {
        Console.WriteLine("No arguments");
    }
    else
    {
        Console.WriteLine("One or more arguments");
    }
}
```

- `switch` statement:

```
static void SwitchStatement(string[] args)
{
    int n = args.Length;
    switch (n)
    {
        case 0:
            Console.WriteLine("No arguments");
            break;
        case 1:
            Console.WriteLine("One argument");
            break;
        default:
            Console.WriteLine($"{n} arguments");
            break;
    }
}
```

- `while` statement:

```
static void WhileStatement(string[] args)
{
    int i = 0;
    while (i < args.Length)
    {
        Console.WriteLine(args[i]);
        i++;
    }
}
```

- `do` statement:

```
static void DoStatement(string[] args)
{
    string s;
    do
    {
        s = Console.ReadLine();
        Console.WriteLine(s);
    } while (!string.IsNullOrEmpty(s));
}
```

- `for` statement:

```
static void ForStatement(string[] args)
{
    for (int i = 0; i < args.Length; i++)
    {
        Console.WriteLine(args[i]);
    }
}
```

- `foreach` statement:

```
static void ForEachStatement(string[] args)
{
    foreach (string s in args)
    {
        Console.WriteLine(s);
    }
}
```

- `break` statement:

```
static void BreakStatement(string[] args)
{
    while (true)
    {
        string s = Console.ReadLine();
        if (string.IsNullOrEmpty(s))
            break;
        Console.WriteLine(s);
    }
}
```

- `continue` statement:

```
static void ContinueStatement(string[] args)
{
    for (int i = 0; i < args.Length; i++)
    {
        if (args[i].StartsWith("/"))
            continue;
        Console.WriteLine(args[i]);
    }
}
```

- `goto` statement:

```
static void GoToStatement(string[] args)
{
    int i = 0;
    goto check;
loop:
    Console.WriteLine(args[i++]);
check:
    if (i < args.Length)
        goto loop;
}
```

- `return` statement:

```
static int Add(int a, int b)
{
    return a + b;
}
static void ReturnStatement(string[] args)
{
    Console.WriteLine(Add(1, 2));
    return;
}
```

- `yield` statement:

```
static System.Collections.Generic.IEnumerable<int> Range(int from, int to)
{
    for (int i = from; i < to; i++)
    {
        yield return i;
    }
    yield break;
}
static void YieldStatement(string[] args)
{
    foreach (int i in Range(-10,10))
    {
        Console.WriteLine(i);
    }
}
```

- `throw` statements and `try` statements:

```

static double Divide(double x, double y)
{
    if (y == 0)
        throw new DivideByZeroException();
    return x / y;
}
static void TryCatch(string[] args)
{
    try
    {
        if (args.Length != 2)
        {
            throw new InvalidOperationException("Two numbers required");
        }
        double x = double.Parse(args[0]);
        double y = double.Parse(args[1]);
        Console.WriteLine(Divide(x, y));
    }
    catch (InvalidOperationException e)
    {
        Console.WriteLine(e.Message);
    }
    finally
    {
        Console.WriteLine("Good bye!");
    }
}

```

- `checked` and `unchecked` statements:

```

static void CheckedUnchecked(string[] args)
{
    int x = int.MaxValue;
    unchecked
    {
        Console.WriteLine(x + 1); // Overflow
    }
    checked
    {
        Console.WriteLine(x + 1); // Exception
    }
}

```

- `lock` statement:

```

class Account
{
    decimal balance;
    private readonly object sync = new object();
    public void Withdraw(decimal amount)
    {
        lock (sync)
        {
            if (amount > balance)
            {
                throw new Exception(
                    "Insufficient funds");
            }
            balance -= amount;
        }
    }
}

```

- `using` statement:

```
static void UsingStatement(string[] args)
{
    using (TextWriter w = File.CreateText("test.txt"))
    {
        w.WriteLine("Line one");
        w.WriteLine("Line two");
        w.WriteLine("Line three");
    }
}
```

[PREVIOUS](#)[NEXT](#)

Classes and objects

4/25/2019 • 23 minutes to read • [Edit Online](#)

Classes are the most fundamental of C#'s types. A class is a data structure that combines state (fields) and actions (methods and other function members) in a single unit. A class provides a definition for dynamically created *instances* of the class, also known as *objects*. Classes support *inheritance* and *polymorphism*, mechanisms whereby *derived classes* can extend and specialize *base classes*.

New classes are created using class declarations. A class declaration starts with a header that specifies the attributes and modifiers of the class, the name of the class, the base class (if given), and the interfaces implemented by the class. The header is followed by the class body, which consists of a list of member declarations written between the delimiters `{` and `}`.

The following is a declaration of a simple class named `Point`:

```
public class Point
{
    public int x, y;
    public Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}
```

Instances of classes are created using the `new` operator, which allocates memory for a new instance, invokes a constructor to initialize the instance, and returns a reference to the instance. The following statements create two `Point` objects and store references to those objects in two variables:

```
Point p1 = new Point(0, 0);
Point p2 = new Point(10, 20);
```

The memory occupied by an object is automatically reclaimed when the object is no longer reachable. It is neither necessary nor possible to explicitly deallocate objects in C#.

Members

The members of a class are either static members or instance members. Static members belong to classes, and instance members belong to objects (instances of classes).

The following provides an overview of the kinds of members a class can contain.

- Constants
 - Constant values associated with the class
- Fields
 - Variables of the class
- Methods
 - Computations and actions that can be performed by the class
- Properties
 - Actions associated with reading and writing named properties of the class
- Indexers

- Actions associated with indexing instances of the class like an array
- Events
 - Notifications that can be generated by the class
- Operators
 - Conversions and expression operators supported by the class
- Constructors
 - Actions required to initialize instances of the class or the class itself
- Finalizers
 - Actions to perform before instances of the class are permanently discarded
- Types
 - Nested types declared by the class

Accessibility

Each member of a class has an associated accessibility, which controls the regions of program text that are able to access the member. There are six possible forms of accessibility. These are summarized below.

- `public`
 - Access not limited
- `protected`
 - Access limited to this class or classes derived from this class
- `internal`
 - Access limited to the current assembly (.exe, .dll, etc.)
- `protected internal`
 - Access limited to the containing class, classes derived from the containing class, or classes within the same assembly
- `private`
 - Access limited to this class
- `private protected`
 - Access limited to the containing class or classes derived from the containing type within the same assembly

Type parameters

A class definition may specify a set of type parameters by following the class name with angle brackets enclosing a list of type parameter names. The type parameters can then be used in the body of the class declarations to define the members of the class. In the following example, the type parameters of `Pair` are `TFirst` and `TSecond`:

```
public class Pair<TFirst,TSecond>
{
    public TFirst First;
    public TSecond Second;
}
```

A class type that is declared to take type parameters is called a *generic class type*. Struct, interface and delegate types can also be generic. When the generic class is used, type arguments must be provided for each of the type parameters:

```
Pair<int,string> pair = new Pair<int,string> { First = 1, Second = "two" };
int i = pair.First;    // TFirst is int
string s = pair.Second; // TSecond is string
```

A generic type with type arguments provided, like `Pair<int,string>` above, is called a *constructed type*.

Base classes

A class declaration may specify a base class by following the class name and type parameters with a colon and the name of the base class. Omitting a base class specification is the same as deriving from type `object`. In the following example, the base class of `Point3D` is `Point`, and the base class of `Point` is `object`:

```
public class Point
{
    public int x, y;
    public Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}
public class Point3D: Point
{
    public int z;
    public Point3D(int x, int y, int z) :
        base(x, y)
    {
        this.z = z;
    }
}
```

A class inherits the members of its base class. Inheritance means that a class implicitly contains all members of its base class, except for the instance and static constructors, and the finalizers of the base class. A derived class can add new members to those it inherits, but it cannot remove the definition of an inherited member. In the previous example, `Point3D` inherits the `x` and `y` fields from `Point`, and every `Point3D` instance contains three fields, `x`, `y`, and `z`.

An implicit conversion exists from a class type to any of its base class types. Therefore, a variable of a class type can reference an instance of that class or an instance of any derived class. For example, given the previous class declarations, a variable of type `Point` can reference either a `Point` or a `Point3D`:

```
Point a = new Point(10, 20);
Point b = new Point3D(10, 20, 30);
```

Fields

A *field* is a variable that is associated with a class or with an instance of a class.

A field declared with the static modifier defines a static field. A static field identifies exactly one storage location. No matter how many instances of a class are created, there is only ever one copy of a static field.

A field declared without the static modifier defines an instance field. Every instance of a class contains a separate copy of all the instance fields of that class.

In the following example, each instance of the `Color` class has a separate copy of the `r`, `g`, and `b` instance fields, but there is only one copy of the `Black`, `White`, `Red`, `Green`, and `Blue` static fields:

```

public class Color
{
    public static readonly Color Black = new Color(0, 0, 0);
    public static readonly Color White = new Color(255, 255, 255);
    public static readonly Color Red = new Color(255, 0, 0);
    public static readonly Color Green = new Color(0, 255, 0);
    public static readonly Color Blue = new Color(0, 0, 255);
    private byte r, g, b;
    public Color(byte r, byte g, byte b)
    {
        this.r = r;
        this.g = g;
        this.b = b;
    }
}

```

As shown in the previous example, *read-only fields* may be declared with a `readonly` modifier. Assignment to a `readonly` field can only occur as part of the field's declaration or in a constructor in the same class.

Methods

A *method* is a member that implements a computation or action that can be performed by an object or class. *Static methods* are accessed through the class. *Instance methods* are accessed through instances of the class.

Methods may have a list of *parameters*, which represent values or variable references passed to the method, and a *return type*, which specifies the type of the value computed and returned by the method. A method's return type is `void` if it does not return a value.

Like types, methods may also have a set of type parameters, for which type arguments must be specified when the method is called. Unlike types, the type arguments can often be inferred from the arguments of a method call and need not be explicitly given.

The *signature* of a method must be unique in the class in which the method is declared. The signature of a method consists of the name of the method, the number of type parameters and the number, modifiers, and types of its parameters. The signature of a method does not include the return type.

Parameters

Parameters are used to pass values or variable references to methods. The parameters of a method get their actual values from the *arguments* that are specified when the method is invoked. There are four kinds of parameters: value parameters, reference parameters, output parameters, and parameter arrays.

A *value parameter* is used for passing input arguments. A value parameter corresponds to a local variable that gets its initial value from the argument that was passed for the parameter. Modifications to a value parameter do not affect the argument that was passed for the parameter.

Value parameters can be optional, by specifying a default value so that corresponding arguments can be omitted.

A *reference parameter* is used for passing arguments by reference. The argument passed for a reference parameter must be a variable with a definite value, and during execution of the method, the reference parameter represents the same storage location as the argument variable. A reference parameter is declared with the `ref` modifier. The following example shows the use of `ref` parameters.

```

using System;
class RefExample
{
    static void Swap(ref int x, ref int y)
    {
        int temp = x;
        x = y;
        y = temp;
    }
    public static void SwapExample()
    {
        int i = 1, j = 2;
        Swap(ref i, ref j);
        Console.WriteLine($"{i} {j}");    // Outputs "2 1"
    }
}

```

An *output parameter* is used for passing arguments by reference. It's similar to a reference parameter, except that it doesn't require that you explicitly assign a value to the caller-provided argument. An output parameter is declared with the `out` modifier. The following example shows the use of `out` parameters using the syntax introduced in C# 7.

```

using System;
class OutExample
{
    static void Divide(int x, int y, out int result, out int remainder)
    {
        result = x / y;
        remainder = x % y;
    }
    public static void OutUsage()
    {
        Divide(10, 3, out int res, out int rem);
        Console.WriteLine("{0} {1}", res, rem); // Outputs "3 1"
    }
}

```

A *parameter array* permits a variable number of arguments to be passed to a method. A parameter array is declared with the `params` modifier. Only the last parameter of a method can be a parameter array, and the type of a parameter array must be a single-dimensional array type. The `Write` and `WriteLine` methods of the [System.Console](#) class are good examples of parameter array usage. They are declared as follows.

```

public class Console
{
    public static void Write(string fmt, params object[] args) { }
    public static void WriteLine(string fmt, params object[] args) { }
    // ...
}

```

Within a method that uses a parameter array, the parameter array behaves exactly like a regular parameter of an array type. However, in an invocation of a method with a parameter array, it is possible to pass either a single argument of the parameter array type or any number of arguments of the element type of the parameter array. In the latter case, an array instance is automatically created and initialized with the given arguments. This example

```

Console.WriteLine("x={0} y={1} z={2}", x, y, z);

```

is equivalent to writing the following.

```
string s = "x={0} y={1} z={2}";
object[] args = new object[3];
args[0] = x;
args[1] = y;
args[2] = z;
Console.WriteLine(s, args);
```

Method body and local variables

A method's body specifies the statements to execute when the method is invoked.

A method body can declare variables that are specific to the invocation of the method. Such variables are called *local variables*. A local variable declaration specifies a type name, a variable name, and possibly an initial value. The following example declares a local variable `i` with an initial value of zero and a local variable `j` with no initial value.

```
using System;
class Squares
{
    public static void WriteSquares()
    {
        int i = 0;
        int j;
        while (i < 10)
        {
            j = i * i;
            Console.WriteLine($"{i} x {i} = {j}");
            i = i + 1;
        }
    }
}
```

C# requires a local variable to be *definitely assigned* before its value can be obtained. For example, if the declaration of the previous `i` did not include an initial value, the compiler would report an error for the subsequent usages of `i` because `i` would not be definitely assigned at those points in the program.

A method can use `return` statements to return control to its caller. In a method returning `void`, `return` statements cannot specify an expression. In a method returning non-void, `return` statements must include an expression that computes the return value.

Static and instance methods

A method declared with a static modifier is a *static method*. A static method does not operate on a specific instance and can only directly access static members.

A method declared without a static modifier is an *instance method*. An instance method operates on a specific instance and can access both static and instance members. The instance on which an instance method was invoked can be explicitly accessed as `this`. It is an error to refer to `this` in a static method.

The following `Entity` class has both static and instance members.

```

class Entity
{
    static int nextSerialNo;
    int serialNo;
    public Entity()
    {
        serialNo = nextSerialNo++;
    }
    public int GetSerialNo()
    {
        return serialNo;
    }
    public static int GetNextSerialNo()
    {
        return nextSerialNo;
    }
    public static void SetNextSerialNo(int value)
    {
        nextSerialNo = value;
    }
}

```

Each `Entity` instance contains a serial number (and presumably some other information that is not shown here). The `Entity` constructor (which is like an instance method) initializes the new instance with the next available serial number. Because the constructor is an instance member, it is permitted to access both the `serialNo` instance field and the `nextSerialNo` static field.

The `GetNextSerialNo` and `SetNextSerialNo` static methods can access the `nextSerialNo` static field, but it would be an error for them to directly access the `serialNo` instance field.

The following example shows the use of the `Entity` class.

```

using System;
class EntityExample
{
    public static void Usage()
    {
        Entity.SetNextSerialNo(1000);
        Entity e1 = new Entity();
        Entity e2 = new Entity();
        Console.WriteLine(e1.GetSerialNo());           // Outputs "1000"
        Console.WriteLine(e2.GetSerialNo());           // Outputs "1001"
        Console.WriteLine(Entity.GetNextSerialNo());   // Outputs "1002"
    }
}

```

Note that the `SetNextSerialNo` and `GetNextSerialNo` static methods are invoked on the class whereas the `GetSerialNo` instance method is invoked on instances of the class.

Virtual, override, and abstract methods

When an instance method declaration includes a `virtual` modifier, the method is said to be a *virtual method*.

When no virtual modifier is present, the method is said to be a *nonvirtual method*.

When a virtual method is invoked, the *run-time type* of the instance for which that invocation takes place determines the actual method implementation to invoke. In a nonvirtual method invocation, the *compile-time type* of the instance is the determining factor.

A virtual method can be *overridden* in a derived class. When an instance method declaration includes an override modifier, the method overrides an inherited virtual method with the same signature. Whereas a virtual method declaration introduces a new method, an override method declaration specializes an existing inherited virtual

method by providing a new implementation of that method.

An *abstract method* is a virtual method with no implementation. An abstract method is declared with the abstract modifier and is permitted only in a class that is also declared abstract. An abstract method must be overridden in every non-abstract derived class.

The following example declares an abstract class, `Expression`, which represents an expression tree node, and three derived classes, `Constant`, `VariableReference`, and `Operation`, which implement expression tree nodes for constants, variable references, and arithmetic operations. (This is similar to, but not to be confused with the expression tree types).


```

using System;
using System.Collections.Generic;
public abstract class Expression
{
    public abstract double Evaluate(Dictionary<string,object> vars);
}
public class Constant: Expression
{
    double value;
    public Constant(double value)
    {
        this.value = value;
    }
    public override double Evaluate(Dictionary<string,object> vars)
    {
        return value;
    }
}
public class VariableReference: Expression
{
    string name;
    public VariableReference(string name)
    {
        this.name = name;
    }
    public override double Evaluate(Dictionary<string,object> vars)
    {
        object value = vars[name];
        if (value == null)
        {
            throw new Exception("Unknown variable: " + name);
        }
        return Convert.ToDouble(value);
    }
}
public class Operation: Expression
{
    Expression left;
    char op;
    Expression right;
    public Operation(Expression left, char op, Expression right)
    {
        this.left = left;
        this.op = op;
        this.right = right;
    }
    public override double Evaluate(Dictionary<string,object> vars)
    {
        double x = left.Evaluate(vars);
        double y = right.Evaluate(vars);
        switch (op) {
            case '+': return x + y;
            case '-': return x - y;
            case '*': return x * y;
            case '/': return x / y;
        }
        throw new Exception("Unknown operator");
    }
}

```

The previous four classes can be used to model arithmetic expressions. For example, using instances of these classes, the expression `x + 3` can be represented as follows.

```
Expression e = new Operation(
    new VariableReference("x"),
    '+',
    new Constant(3));
```

The `Evaluate` method of an `Expression` instance is invoked to evaluate the given expression and produce a `double` value. The method takes a `Dictionary` argument that contains variable names (as keys of the entries) and values (as values of the entries). Because `Evaluate` is an abstract method, non-abstract classes derived from `Expression` must override `Evaluate`.

A `Constant`'s implementation of `Evaluate` simply returns the stored constant. A `VariableReference`'s implementation looks up the variable name in the dictionary and returns the resulting value. An `Operation`'s implementation first evaluates the left and right operands (by recursively invoking their `Evaluate` methods) and then performs the given arithmetic operation.

The following program uses the `Expression` classes to evaluate the expression $x * (y + 2)$ for different values of `x` and `y`.

```
using System;
using System.Collections.Generic;
class InheritanceExample
{
    public static void ExampleUsage()
    {
        Expression e = new Operation(
            new VariableReference("x"),
            '*',
            new Operation(
                new VariableReference("y"),
                '+',
                new Constant(2)
            )
        );
        Dictionary<string,object> vars = new Dictionary<string, object>();
        vars["x"] = 3;
        vars["y"] = 5;
        Console.WriteLine(e.Evaluate(vars)); // Outputs "21"
        vars["x"] = 1.5;
        vars["y"] = 9;
        Console.WriteLine(e.Evaluate(vars)); // Outputs "16.5"
    }
}
```

Method overloading

Method *overloading* permits multiple methods in the same class to have the same name as long as they have unique signatures. When compiling an invocation of an overloaded method, the compiler uses *overload resolution* to determine the specific method to invoke. Overload resolution finds the one method that best matches the arguments or reports an error if no single best match can be found. The following example shows overload resolution in effect. The comment for each invocation in the `UsageExample` method shows which method is actually invoked.

```

using System;
class OverloadingExample
{
    static void F()
    {
        Console.WriteLine("F()");
    }
    static void F(object x)
    {
        Console.WriteLine("F(object)");
    }
    static void F(int x)
    {
        Console.WriteLine("F(int)");
    }
    static void F(double x)
    {
        Console.WriteLine("F(double)");
    }
    static void F<T>(T x)
    {
        Console.WriteLine("F<T>(T)");
    }
    static void F(double x, double y)
    {
        Console.WriteLine("F(double, double)");
    }
    public static void UsageExample()
    {
        F();           // Invokes F()
        F(1);          // Invokes F(int)
        F(1.0);        // Invokes F(double)
        F("abc");      // Invokes F<string>(string)
        F((double)1);  // Invokes F(double)
        F((object)1);  // Invokes F(object)
        F<int>(1);     // Invokes F<int>(int)
        F(1, 1);      // Invokes F(double, double)
    }
}

```

As shown by the example, a particular method can always be selected by explicitly casting the arguments to the exact parameter types and/or explicitly supplying type arguments.

Other function members

Members that contain executable code are collectively known as the *function members* of a class. The preceding section describes methods, which are the primary kind of function members. This section describes the other kinds of function members supported by C#: constructors, properties, indexers, events, operators, and finalizers.

The following shows a generic class called `MyList<T>`, which implements a growable list of objects. The class contains several examples of the most common kinds of function members.

NOTE

This example creates a `MyList` class, which is not the same as the .NET standard [System.Collections.Generic.List<T>](#). It does illustrate the concepts needed for this tour, but is not a replacement for that class.

```

public class MyList<T>
{
    // Constant
    const int defaultCapacity = 4;
}

```

```

// Fields
T[] items;
int count;

// Constructor
public MyList(int capacity = defaultCapacity)
{
    items = new T[capacity];
}

// Properties
public int Count => count;

public int Capacity
{
    get { return items.Length; }
    set
    {
        if (value < count) value = count;
        if (value != items.Length)
        {
            T[] newItems = new T[value];
            Array.Copy(items, 0, newItems, 0, count);
            items = newItems;
        }
    }
}

// Indexer
public T this[int index]
{
    get
    {
        return items[index];
    }
    set
    {
        items[index] = value;
        OnChanged();
    }
}

// Methods
public void Add(T item)
{
    if (count == Capacity) Capacity = count * 2;
    items[count] = item;
    count++;
    OnChanged();
}

protected virtual void OnChanged() =>
    Changed?.Invoke(this, EventArgs.Empty);

public override bool Equals(object other) =>
    Equals(this, other as MyList<T>);

static bool Equals(MyList<T> a, MyList<T> b)
{
    if (Object.ReferenceEquals(a, null)) return Object.ReferenceEquals(b, null);
    if (Object.ReferenceEquals(b, null) || a.count != b.count)
        return false;
    for (int i = 0; i < a.count; i++)
    {
        if (!Object.Equals(a.items[i], b.items[i]))
        {
            return false;
        }
    }
}

```

```

    return true;
}

// Event
public event EventHandler Changed;

// Operators
public static bool operator ==(MyList<T> a, MyList<T> b) =>
    Equals(a, b);

public static bool operator !=(MyList<T> a, MyList<T> b) =>
    !Equals(a, b);
}

```

Constructors

C# supports both instance and static constructors. An *instance constructor* is a member that implements the actions required to initialize an instance of a class. A *static constructor* is a member that implements the actions required to initialize a class itself when it is first loaded.

A constructor is declared like a method with no return type and the same name as the containing class. If a constructor declaration includes a static modifier, it declares a static constructor. Otherwise, it declares an instance constructor.

Instance constructors can be overloaded and can have optional parameters. For example, the `MyList<T>` class declares one instance constructor with a single optional `int` parameter. Instance constructors are invoked using the `new` operator. The following statements allocate two `MyList<string>` instances using the constructor of the `MyList` class with and without the optional argument.

```

MyList<string> list1 = new MyList<string>();
MyList<string> list2 = new MyList<string>(10);

```

Unlike other members, instance constructors are not inherited, and a class has no instance constructors other than those actually declared in the class. If no instance constructor is supplied for a class, then an empty one with no parameters is automatically provided.

Properties

Properties are a natural extension of fields. Both are named members with associated types, and the syntax for accessing fields and properties is the same. However, unlike fields, properties do not denote storage locations. Instead, properties have *accessors* that specify the statements to be executed when their values are read or written.

A property is declared like a field, except that the declaration ends with a get accessor and/or a set accessor written between the delimiters `{` and `}` instead of ending in a semicolon. A property that has both a get accessor and a set accessor is a *read-write property*, a property that has only a get accessor is a *read-only property*, and a property that has only a set accessor is a *write-only property*.

A get accessor corresponds to a parameterless method with a return value of the property type. Except as the target of an assignment, when a property is referenced in an expression, the get accessor of the property is invoked to compute the value of the property.

A set accessor corresponds to a method with a single parameter named `value` and no return type. When a property is referenced as the target of an assignment or as the operand of `++` or `--`, the set accessor is invoked with an argument that provides the new value.

The `MyList<T>` class declares two properties, `Count` and `Capacity`, which are read-only and read-write, respectively. The following is an example of use of these properties:

```
MyList<string> names = new MyList<string>();
names.Capacity = 100; // Invokes set accessor
int i = names.Count; // Invokes get accessor
int j = names.Capacity; // Invokes get accessor
```

Similar to fields and methods, C# supports both instance properties and static properties. Static properties are declared with the static modifier, and instance properties are declared without it.

The accessor(s) of a property can be virtual. When a property declaration includes a `virtual`, `abstract`, or `override` modifier, it applies to the accessor(s) of the property.

Indexers

An *indexer* is a member that enables objects to be indexed in the same way as an array. An indexer is declared like a property except that the name of the member is this followed by a parameter list written between the delimiters `[` and `]`. The parameters are available in the accessor(s) of the indexer. Similar to properties, indexers can be read-write, read-only, and write-only, and the accessor(s) of an indexer can be virtual.

The `MyList<T>` class declares a single read-write indexer that takes an `int` parameter. The indexer makes it possible to index `MyList<T>` instances with `int` values. For example:

```
MyList<string> names = new MyList<string>();
names.Add("Liz");
names.Add("Martha");
names.Add("Beth");
for (int i = 0; i < names.Count; i++)
{
    string s = names[i];
    names[i] = s.ToUpper();
}
```

Indexers can be overloaded, meaning that a class can declare multiple indexers as long as the number or types of their parameters differ.

Events

An *event* is a member that enables a class or object to provide notifications. An event is declared like a field except that the declaration includes an event keyword and the type must be a delegate type.

Within a class that declares an event member, the event behaves just like a field of a delegate type (provided the event is not abstract and does not declare accessors). The field stores a reference to a delegate that represents the event handlers that have been added to the event. If no event handlers are present, the field is `null`.

The `MyList<T>` class declares a single event member called `Changed`, which indicates that a new item has been added to the list. The `Changed` event is raised by the `OnChanged` virtual method, which first checks whether the event is `null` (meaning that no handlers are present). The notion of raising an event is precisely equivalent to invoking the delegate represented by the event—thus, there are no special language constructs for raising events.

Clients react to events through *event handlers*. Event handlers are attached using the `+=` operator and removed using the `-=` operator. The following example attaches an event handler to the `Changed` event of a

```
MyList<string>.
```

```

class EventExample
{
    static int changeCount;
    static void ListChanged(object sender, EventArgs e)
    {
        changeCount++;
    }
    public static void Usage()
    {
        MyList<string> names = new MyList<string>();
        names.Changed += new EventHandler(ListChanged);
        names.Add("Liz");
        names.Add("Martha");
        names.Add("Beth");
        Console.WriteLine(changeCount); // Outputs "3"
    }
}

```

For advanced scenarios where control of the underlying storage of an event is desired, an event declaration can explicitly provide `add` and `remove` accessors, which are somewhat similar to the `set` accessor of a property.

Operators

An *operator* is a member that defines the meaning of applying a particular expression operator to instances of a class. Three kinds of operators can be defined: unary operators, binary operators, and conversion operators. All operators must be declared as `public` and `static`.

The `MyList<T>` class declares two operators, `operator ==` and `operator !=`, and thus gives new meaning to expressions that apply those operators to `MyList` instances. Specifically, the operators define equality of two `MyList<T>` instances as comparing each of the contained objects using their `Equals` methods. The following example uses the `==` operator to compare two `MyList<int>` instances.

```

MyList<int> a = new MyList<int>();
a.Add(1);
a.Add(2);
MyList<int> b = new MyList<int>();
b.Add(1);
b.Add(2);
Console.WriteLine(a == b); // Outputs "True"
b.Add(3);
Console.WriteLine(a == b); // Outputs "False"

```

The first `Console.WriteLine` outputs `True` because the two lists contain the same number of objects with the same values in the same order. Had `MyList<T>` not defined `operator ==`, the first `Console.WriteLine` would have output `False` because `a` and `b` reference different `MyList<int>` instances.

Finalizers

A *finalizer* is a member that implements the actions required to finalize an instance of a class. Finalizers cannot have parameters, they cannot have accessibility modifiers, and they cannot be invoked explicitly. The finalizer for an instance is invoked automatically during garbage collection.

The garbage collector is allowed wide latitude in deciding when to collect objects and run finalizers. Specifically, the timing of finalizer invocations is not deterministic, and finalizers may be executed on any thread. For these and other reasons, classes should implement finalizers only when no other solutions are feasible.

The `using` statement provides a better approach to object destruction.

[PREVIOUS](#)[NEXT](#)

Structs

1/25/2019 • 2 minutes to read • [Edit Online](#)

Like classes, **structs** are data structures that can contain data members and function members, but unlike classes, structs are value types and do not require heap allocation. A variable of a struct type directly stores the data of the struct, whereas a variable of a class type stores a reference to a dynamically allocated object. Struct types do not support user-specified inheritance, and all struct types implicitly inherit from type `ValueType`, which in turn implicitly inherits from `object`.

Structs are particularly useful for small data structures that have value semantics. Complex numbers, points in a coordinate system, or key-value pairs in a dictionary are all good examples of structs. The use of structs rather than classes for small data structures can make a large difference in the number of memory allocations an application performs. For example, the following program creates and initializes an array of 100 points. With `Point` implemented as a class, 101 separate objects are instantiated—one for the array and one each for the 100 elements.

```
public class PointExample
{
    public static void Main()
    {
        Point[] points = new Point[100];
        for (int i = 0; i < 100; i++)
            points[i] = new Point(i, i);
    }
}
```

An alternative is to make `Point` a struct.

```
struct Point
{
    public int x, y;
    public Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}
```

Now, only one object is instantiated—the one for the array—and the `Point` instances are stored in-line in the array.

Struct constructors are invoked with the `new` operator, similar to a class constructor. But, instead of dynamically allocating an object on the managed heap and returning a reference to it, a struct constructor simply returns the struct value itself (typically in a temporary location on the stack), and this value is then copied as necessary.

With classes, it is possible for two variables to reference the same object and thus possible for operations on one variable to affect the object referenced by the other variable. With structs, the variables each have their own copy of the data, and it is not possible for operations on one to affect the other. For example, the output produced by the following code fragment depends on whether `Point` is a class or a struct.

```
Point a = new Point(10, 10);  
Point b = a;  
a.x = 20;  
Console.WriteLine(b.x);
```

If `Point` is a class, the output is 20 because `a` and `b` reference the same object. If `Point` is a struct, the output is 10 because the assignment of `a` to `b` creates a copy of the value, and this copy is unaffected by the subsequent assignment to `a.x`.

The previous example highlights two of the limitations of structs. First, copying an entire struct is typically less efficient than copying an object reference, so assignment and value parameter passing can be more expensive with structs than with reference types. Second, except for `in`, `ref`, and `out` parameters, it is not possible to create references to structs, which rules out their usage in a number of situations.

[PREVIOUS](#)[NEXT](#)

Arrays

5/15/2019 • 2 minutes to read • [Edit Online](#)

An **array** is a data structure that contains a number of variables that are accessed through computed indices. The variables contained in an array, also called the **elements** of the array, are all of the same type, and this type is called the **element type** of the array.

Array types are reference types, and the declaration of an array variable simply sets aside space for a reference to an array instance. Actual array instances are created dynamically at runtime using the `new` operator. The `new` operation specifies the **length** of the new array instance, which is then fixed for the lifetime of the instance. The indices of the elements of an array range from `0` to `Length - 1`. The `new` operator automatically initializes the elements of an array to their default value, which, for example, is zero for all numeric types and `null` for all reference types.

The following example creates an array of `int` elements, initializes the array, and prints out the contents of the array.

```
using System;
class ArrayExample
{
    static void Main()
    {
        int[] a = new int[10];
        for (int i = 0; i < a.Length; i++)
        {
            a[i] = i * i;
        }
        for (int i = 0; i < a.Length; i++)
        {
            Console.WriteLine($"a[{i}] = {a[i]}");
        }
    }
}
```

This example creates and operates on a **single-dimensional array**. C# also supports **multi-dimensional arrays**. The number of dimensions of an array type, also known as the **rank** of the array type, is one plus the number of commas written between the square brackets of the array type. The following example allocates a single-dimensional, a two-dimensional, and a three-dimensional array, respectively.

```
int[] a1 = new int[10];
int[,] a2 = new int[10, 5];
int[,,] a3 = new int[10, 5, 2];
```

The `a1` array contains 10 elements, the `a2` array contains 50 (10×5) elements, and the `a3` array contains 100 ($10 \times 5 \times 2$) elements. The element type of an array can be any type, including an array type. An array with elements of an array type is sometimes called a **jagged array** because the lengths of the element arrays do not all have to be the same. The following example allocates an array of arrays of `int`:

```
int[][] a = new int[3][];
a[0] = new int[10];
a[1] = new int[5];
a[2] = new int[20];
```

The first line creates an array with three elements, each of type `int[]` and each with an initial value of `null`. The subsequent lines then initialize the three elements with references to individual array instances of varying lengths.

The new operator permits the initial values of the array elements to be specified using an **array initializer**, which is a list of expressions written between the delimiters `{` and `}`. The following example allocates and initializes an `int[]` with three elements.

```
int[] a = new int[] {1, 2, 3};
```

Note that the length of the array is inferred from the number of expressions between `{` and `}`. Local variable and field declarations can be shortened further such that the array type does not have to be restated.

```
int[] a = {1, 2, 3};
```

Both of the previous examples are equivalent to the following:

```
int[] t = new int[3];  
t[0] = 1;  
t[1] = 2;  
t[2] = 3;  
int[] a = t;
```

[PREVIOUS](#)[NEXT](#)

Interfaces

5/15/2019 • 2 minutes to read • [Edit Online](#)

An **interface** defines a contract that can be implemented by classes and structs. An interface can contain methods, properties, events, and indexers. An interface does not provide implementations of the members it defines—it merely specifies the members that must be supplied by classes or structs that implement the interface.

Interfaces may employ **multiple inheritance**. In the following example, the interface `ICombobox` inherits from both `ITextBox` and `IListBox`.

```
interface IControl
{
    void Paint();
}
interface ITextBox: IControl
{
    void SetText(string text);
}
interface IListBox: IControl
{
    void SetItems(string[] items);
}
interface IComboBox: ITextBox, IListBox {}
```

Classes and structs can implement multiple interfaces. In the following example, the class `EditBox` implements both `IControl` and `IDataBound`.

```
interface IDataBound
{
    void Bind(Binder b);
}
public class EditBox: IControl, IDataBound
{
    public void Paint() { }
    public void Bind(Binder b) { }
}
```

When a class or struct implements a particular interface, instances of that class or struct can be implicitly converted to that interface type. For example

```
EditBox editBox = new EditBox();
IControl control = editBox;
IDataBound dataBound = editBox;
```

In cases where an instance is not statically known to implement a particular interface, dynamic type casts can be used. For example, the following statements use dynamic type casts to obtain an object's `IControl` and `IDataBound` interface implementations. Because the run-time actual type of the object is `EditBox`, the casts succeed.

```
object obj = new EditBox();
IControl control = (IControl)obj;
IDataBound dataBound = (IDataBound)obj;
```

In the previous `TextBox` class, the `Paint` method from the `IControl` interface and the `Bind` method from the `IDataBound` interface are implemented using public members. C# also supports explicit ***interface member implementations***, enabling the class or struct to avoid making the members public. An explicit interface member implementation is written using the fully qualified interface member name. For example, the `TextBox` class could implement the `IControl.Paint` and `IDataBound.Bind` methods using explicit interface member implementations as follows.

```
public class TextBox: IControl, IDataBound
{
    void IControl.Paint() { }
    void IDataBound.Bind(Binder b) { }
}
```

Explicit interface members can only be accessed via the interface type. For example, the implementation of `IControl.Paint` provided by the previous `TextBox` class can only be invoked by first converting the `TextBox` reference to the `IControl` interface type.

```
TextBox textBox = new TextBox();
textBox.Paint();           // Error, no such method
IControl control = textBox;
control.Paint();           // Ok
```

[PREVIOUS](#)[NEXT](#)

Enums

3/14/2019 • 2 minutes to read • [Edit Online](#)

An **enum type** is a distinct value type with a set of named constants. You define enums when you need to define a type that can have a set of discrete values. They use one of the integral value types as their underlying storage. They provide semantic meaning to the discrete values.

The following example declares and uses an `enum` type named `Color` with three constant values, `Red`, `Green`, and `Blue`.

```
using System;
enum Color
{
    Red,
    Green,
    Blue
}
class EnumExample
{
    static void PrintColor(Color color)
    {
        switch (color)
        {
            case Color.Red:
                Console.WriteLine("Red");
                break;
            case Color.Green:
                Console.WriteLine("Green");
                break;
            case Color.Blue:
                Console.WriteLine("Blue");
                break;
            default:
                Console.WriteLine("Unknown color");
                break;
        }
    }
    static void Main()
    {
        Color c = Color.Red;
        PrintColor(c);
        PrintColor(Color.Blue);
    }
}
```

Each `enum` type has a corresponding integral type called the **underlying type** of the `enum` type. An `enum` type that does not explicitly declare an underlying type has an underlying type of `int`. An `enum` type's storage format and range of possible values are determined by its underlying type. The set of values that an `enum` type can take on is not limited by its `enum` members. In particular, any value of the underlying type of an `enum` can be cast to the `enum` type and is a distinct valid value of that `enum` type.

The following example declares an `enum` type named `Alignment` with an underlying type of `sbyte`.

```
enum Alignment: sbyte
{
    Left = -1,
    Center = 0,
    Right = 1
}
```

As shown by the previous example, an `enum` member declaration can include a constant expression that specifies the value of the member. The constant value for each `enum` member must be in the range of the underlying type of the `enum`. When an `enum` member declaration does not explicitly specify a value, the member is given the value zero (if it is the first member in the `enum` type) or the value of the textually preceding `enum` member plus one.

`Enum` values can be converted to integral values and vice versa using type casts. For example:

```
int i = (int)Color.Blue;    // int i = 2;
Color c = (Color)2;        // Color c = Color.Blue;
```

The default value of any `enum` type is the integral value zero converted to the `enum` type. In cases where variables are automatically initialized to a default value, this is the value given to variables of `enum` types. In order for the default value of an `enum` type to be easily available, the literal `0` implicitly converts to any `enum` type. Thus, the following is permitted.

```
Color c = 0;
```

[PREVIOUS](#)[NEXT](#)

Delegates

5/15/2019 • 2 minutes to read • [Edit Online](#)

A **delegate type** represents references to methods with a particular parameter list and return type. Delegates make it possible to treat methods as entities that can be assigned to variables and passed as parameters. Delegates are similar to the concept of function pointers found in some other languages, but unlike function pointers, delegates are object-oriented and type-safe.

The following example declares and uses a delegate type named `Function`.

```
using System;
delegate double Function(double x);
class Multiplier
{
    double factor;
    public Multiplier(double factor)
    {
        this.factor = factor;
    }
    public double Multiply(double x)
    {
        return x * factor;
    }
}
class DelegateExample
{
    static double Square(double x)
    {
        return x * x;
    }
    static double[] Apply(double[] a, Function f)
    {
        double[] result = new double[a.Length];
        for (int i = 0; i < a.Length; i++) result[i] = f(a[i]);
        return result;
    }
    static void Main()
    {
        double[] a = {0.0, 0.5, 1.0};
        double[] squares = Apply(a, Square);
        double[] sines = Apply(a, Math.Sin);
        Multiplier m = new Multiplier(2.0);
        double[] doubles = Apply(a, m.Multiply);
    }
}
```

An instance of the `Function` delegate type can reference any method that takes a `double` argument and returns a `double` value. The `Apply` method applies a given Function to the elements of a `double[]`, returning a `double[]` with the results. In the `Main` method, `Apply` is used to apply three different functions to a `double[]`.

A delegate can reference either a static method (such as `Square` or `Math.Sin` in the previous example) or an instance method (such as `m.Multiply` in the previous example). A delegate that references an instance method also references a particular object, and when the instance method is invoked through the delegate, that object becomes `this` in the invocation.

Delegates can also be created using anonymous functions, which are "inline methods" that are created on the fly. Anonymous functions can see the local variables of the surrounding methods. Thus, the multiplier example above

can be written more easily without using a Multiplier class:

```
double[] doubles = Apply(a, (double x) => x * 2.0);
```

An interesting and useful property of a delegate is that it does not know or care about the class of the method it references; all that matters is that the referenced method has the same parameters and return type as the delegate.

[PREVIOUS](#)[NEXT](#)

Attributes

5/15/2019 • 2 minutes to read • [Edit Online](#)

Types, members, and other entities in a C# program support modifiers that control certain aspects of their behavior. For example, the accessibility of a method is controlled using the `public`, `protected`, `internal`, and `private` modifiers. C# generalizes this capability such that user-defined types of declarative information can be attached to program entities and retrieved at run-time. Programs specify this additional declarative information by defining and using **attributes**.

The following example declares a `HelpAttribute` attribute that can be placed on program entities to provide links to their associated documentation.

```
using System;

public class HelpAttribute: Attribute
{
    string url;
    string topic;
    public HelpAttribute(string url)
    {
        this.url = url;
    }

    public string Url => url;

    public string Topic {
        get { return topic; }
        set { topic = value; }
    }
}
```

All attribute classes derive from the `Attribute` base class provided by the standard library. Attributes can be applied by giving their name, along with any arguments, inside square brackets just before the associated declaration. If an attribute's name ends in `Attribute`, that part of the name can be omitted when the attribute is referenced. For example, the `HelpAttribute` can be used as follows.

```
[Help("https://docs.microsoft.com/dotnet/csharp/tour-of-csharp/attributes")]
public class Widget
{
    [Help("https://docs.microsoft.com/dotnet/csharp/tour-of-csharp/attributes",
        Topic = "Display")]
    public void Display(string text) {}
}
```

This example attaches a `HelpAttribute` to the `Widget` class. It adds another `HelpAttribute` to the `Display` method in the class. The public constructors of an attribute class control the information that must be provided when the attribute is attached to a program entity. Additional information can be provided by referencing public read-write properties of the attribute class (such as the reference to the `Topic` property previously).

The metadata defined by attributes can be read and manipulated at runtime using reflection. When a particular attribute is requested using this technique, the constructor for the attribute class is invoked with the information provided in the program source, and the resulting attribute instance is returned. If additional information was provided through properties, those properties are set to the given values before the attribute instance is returned.

The following code sample demonstrates how to get the `HelpAttribute` instances associated to the `Widget` class and its `Display` method.

```
Type widgetType = typeof(Widget);

//Gets every HelpAttribute defined for the Widget type
object[] widgetClassAttributes = widgetType.GetCustomAttributes(typeof(HelpAttribute), false);

if (widgetClassAttributes.Length > 0)
{
    HelpAttribute attr = (HelpAttribute)widgetClassAttributes[0];
    Console.WriteLine($"Widget class help URL : {attr.Url} - Related topic : {attr.Topic}");
}

System.Reflection.MethodInfo displayMethod = widgetType.GetMethod(nameof(Widget.Display));

//Gets every HelpAttribute defined for the Widget.Display method
object[] displayMethodAttributes = displayMethod.GetCustomAttributes(typeof(HelpAttribute), false);

if (displayMethodAttributes.Length > 0)
{
    HelpAttribute attr = (HelpAttribute)displayMethodAttributes[0];
    Console.WriteLine($"Display method help URL : {attr.Url} - Related topic : {attr.Topic}");
}

Console.ReadLine();
```

[PREVIOUS](#)