Contents

Classes and Structs (C# Programming Guide)

8/22/2019 • 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

Classes (C# Programming Guide)

8/19/2019 • 5 minutes to read • Edit Online

Reference types

A type that is defined as a class 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 instance of the class by using the new operator, or assign it an object of a compatible type that may have been created elsewhere, as shown in the following example:

```
//Declaring an object of type MyClass.
MyClass mc = new MyClass();

//Declaring another object of the same type, assigning it the value of the first object.
MyClass mc2 = mc;
```

When the object is created, enough memory is allocated on the managed heap for that specific object, and the variable holds only a reference to the location of said 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 and garbage collection.

Declaring Classes

Classes are declared by using the class keyword followed by a unique identifier, as shown in the following example:

```
//[access modifier] - [class] - [identifier]
public class Customer
{
   // Fields, properties, methods and events go here...
}
```

The `class` keyword is preceded by the access level. Because public is used in this case, anyone can create instances of this class. The name of the class follows the `class` keyword. The name of the class must be a valid C# identifier name. The remainder of the definition is the class body, where the behavior and data are defined. Fields, properties, methods, and events on a class are collectively referred to as *class members*.

Creating objects

Although they are sometimes used interchangeably, a class and an object are different things. A class defines a type of object, but it is not an object itself. An object is a concrete entity based on a class, and is sometimes referred to as an instance of a class.

Objects can be created by using the new keyword followed by the name of the class that the object will be based on, like this:

```
Customer object1 = new Customer();
```

When an instance of a class is created, a reference to the object is passed back to the programmer. In the previous

example, `object1` is a reference to an object that is based on `Customer`. This reference refers to the new object but does not contain the object data itself. In fact, you can create an object reference without creating an object at all:

```
Customer object2;
```

We don't recommend creating object references such as this one that don't refer to an object because trying to access an object through such a reference will fail at run time. However, such a reference can be made to refer to an object, either by creating a new object, or by assigning it to an existing object, such as this:

```
Customer object3 = new Customer();
Customer object4 = object3;
```

This code creates two object references that both refer to the same object. Therefore, any changes to the object made through `object3` are reflected in subsequent uses of `object4`. Because objects that are based on classes are referred to by reference, classes are known as reference types.

Class inheritance

Classes fully support *inheritance*, a fundamental characteristic of object-oriented programming. 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 class virtual methods.

Inheritance is accomplished by using a *derivation*, which means a class is declared by using a *base class* from which it inherits data and behavior. A base class is specified by appending a colon and the name of the base class following the derived class name, like this:

```
public class Manager : Employee
{
    // Employee fields, properties, methods and events are inherited
    // New Manager fields, properties, methods and events go here...
}
```

When a class declares a base class, it inherits all the members of the base class except the constructors. For more information, see Inheritance.

Unlike C++, a class in C# can only directly inherit from one base class. However, because a base class may itself inherit from another class, a class may indirectly inherit multiple base classes. Furthermore, a class can directly implement more than one interface. For more information, see Interfaces.

A class can be declared abstract. An abstract class contains abstract methods that have a signature definition but no implementation. Abstract classes cannot be instantiated. They can only be used through derived classes that implement the abstract methods. By contrast, a sealed class does not allow other classes to derive from it. For more information, see Abstract and Sealed Classes and Class Members.

Class definitions can be split between different source files. For more information, see Partial Classes and Methods.

Example

The following example defines a public class that contains an auto-implemented property, a method, and a special method called a constructor. For more information, see Properties, Methods, and Constructors topics. The instances of the class are then instantiated with the `new` keyword.

```
using System;

public class Person
{
    // Constructor that takes no arguments:
    public Person()
    {
        Name = "unknown";
    }

    // Constructor that takes one argument:
    public Person(string name)
    {
        Name = name;
    }

    // Auto-implemented readonly property:
    public string Name { get; }

    // Method that overrides the base class (System.Object) implementation.
    public override string ToString()
    {
        return Name;
    }
}
class TestPerson
{
    static void Main()
    {
        // Call the constructor that has no parameters.
        var person1 = new Person();
        Console.WriteLine(person1.Name);

        // Call the constructor that has one parameter.
        var person2 = new Person("Sarah Jones");
        Console.WriteLine(person2.Name);
        // Get the string representation of the person2 instance.
        Console.WriteLine(person2);

        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
// Output:
// unknown
// Sarah Jones
// Sarah Jones
```

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
- Object-Oriented Programming
- Polymorphism
- Identifier names
- Members
- Methods

- Constructors
- Finalizers
- Objects

Objects (C# Programming Guide)

8/19/2019 • 5 minutes to read • Edit Online

A class or struct definition is like a blueprint that specifies what the type can do. An object is basically a block of memory that has been allocated and configured according to the blueprint. A program may create many objects of the same class. Objects are also called instances, and they can be stored in either a named variable or in an array or collection. Client code is the code that uses these variables to call the methods and access the public properties of the object. In an object-oriented language such as C#, a typical program consists of multiple objects interacting dynamically.

> **NOTE**
>
> Static types behave differently than what is described here. For more information, see Static Classes and Static Class Members.

Struct Instances vs. Class Instances

Because classes are reference types, a variable of a class object holds a reference to the address of the object on the managed heap. If a second object of the same type is assigned to the first object, then both variables refer to the object at that address. This point is discussed in more detail later in this topic.

Instances of classes are created by using the new operator. In the following example, `Person` is the type and `person1` and `person 2` are instances, or objects, of that type.

```
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
    public Person(string name, int age)
    {
        Name = name;
        Age = age;
    }
    // Other properties, methods, events...
}

class Program
{
    static void Main()
    {
        Person person1 = new Person("Leopold", 6);
        Console.WriteLine("person1 Name = {0} Age = {1}", person1.Name, person1.Age);

        // Declare new person, assign person1 to it.
        Person person2 = person1;

        // Change the name of person2, and person1 also changes.
        person2.Name = "Molly";
        person2.Age = 16;

        Console.WriteLine("person2 Name = {0} Age = {1}", person2.Name, person2.Age);
        Console.WriteLine("person1 Name = {0} Age = {1}", person1.Name, person1.Age);

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

    }
}
/*
    Output:
    person1 Name = Leopold Age = 6
    person2 Name = Molly Age = 16
    person1 Name = Molly Age = 16
*/
```

Because structs are value types, a variable of a struct object holds a copy of the entire object. Instances of structs can also be created by using the `new` operator, but this is not required, as shown in the following example:

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

public class Application
{
    static void Main()
    {
        // Create  struct instance and initialize by using "new".
        // Memory is allocated on thread stack.
        Person p1 = new Person("Alex", 9);
        Console.WriteLine("p1 Name = {0} Age = {1}", p1.Name, p1.Age);

        // Create  new struct object. Note that  struct can be initialized
        // without using "new".
        Person p2 = p1;

        // Assign values to p2 members.
        p2.Name = "Spencer";
        p2.Age = 7;
        Console.WriteLine("p2 Name = {0} Age = {1}", p2.Name, p2.Age);

        // p1 values remain unchanged because p2 is  copy.
        Console.WriteLine("p1 Name = {0} Age = {1}", p1.Name, p1.Age);

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/*
  Output:
    p1 Name = Alex Age = 9
    p2 Name = Spencer Age = 7
    p1 Name = Alex Age = 9
*/
```

The memory for both `p1` and `p2` is allocated on the thread stack. That memory is reclaimed along with the type or method in which it is declared. This is one reason why structs are copied on assignment. By contrast, the memory that is allocated for a class instance is automatically reclaimed (garbage collected) by the common language runtime when all references to the object have gone out of scope. It is not possible to deterministically destroy a class object like you can in C++. For more information about garbage collection in the .NET Framework, see Garbage Collection.

> **NOTE**
> The allocation and deallocation of memory on the managed heap is highly optimized in the common language runtime. In most cases there is no significant difference in the performance cost of allocating a class instance on the heap versus allocating a struct instance on the stack.

Object Identity vs. Value Equality

When you compare two objects for equality, you must first distinguish whether you want to know whether the two variables represent the same object in memory, or whether the values of one or more of their fields are equivalent.

If you are intending to compare values, you must consider whether the objects are instances of value types (structs) or reference types (classes, delegates, arrays).

- To determine whether two class instances refer to the same location in memory (which means that they have the same *identity*), use the static Equals method. (System.Object is the implicit base class for all value types and reference types, including user-defined structs and classes.)
- To determine whether the instance fields in two struct instances have the same values, use the ValueType.Equals method. Because all structs implicitly inherit from System.ValueType, you call the method directly on your object as shown in the following example:

```
// Person is defined in the previous example.

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

Person p1 = new Person("Wallace", 75);
Person p2;
p2.Name = "Wallace";
p2.Age = 75;

if (p2.Equals(p1))
    Console.WriteLine("p2 and p1 have the same values.");

// Output: p2 and p1 have the same values.
```

The System.ValueType implementation of `Equals` uses reflection because it must be able to determine what the fields are in any struct. When creating your own structs, override the `Equals` method to provide an efficient equality algorithm that is specific to your type.

- To determine whether the values of the fields in two class instances are equal, you might be able to use the Equals method or the == operator. However, only use them if the class has overridden or overloaded them to provide a custom definition of what "equality" means for objects of that type. The class might also implement the IEquatable<T> interface or the IEqualityComparer<T> interface. Both interfaces provide methods that can be used to test value equality. When designing your own classes that override `Equals`, make sure to follow the guidelines stated in How to: Define Value Equality for a Type and Object.Equals(Object).

Related Sections

For more information:

- Classes
- Structs
- Constructors
- Finalizers
- Events

See also

- C# Programming Guide
- object
- Inheritance
- class
- struct
- new Operator
- Common Type System

Structs (C# Programming Guide)

7/10/2019 • 2 minutes to read • Edit Online

Structs are defined by using the struct keyword, for example:

```
public struct PostalAddress
{
    // Fields, properties, methods and events go here...
}
```

Structs share most of the same syntax as classes. The name of the struct must be a valid C# identifier name. Structs are more limited than classes in the following ways:

- Within a struct declaration, fields cannot be initialized unless they are declared as const or static.
- A struct cannot declare a parameterless constructor (a constructor without parameters) or a finalizer.
- Structs are copied on assignment. When a struct is assigned to a new variable, all the data is copied, and any modification to the new copy does not change the data for the original copy. This is important to remember when working with collections of value types such as `Dictionary<string, myStruct>`.
- Structs are value types, unlike classes, which are reference types.
- Unlike classes, structs can be instantiated without using a `new` operator.
- Structs can declare constructors that have parameters.
- A struct cannot inherit from another struct or class, and it cannot be the base of a class. All structs inherit directly from ValueType, which inherits from Object.
- A struct can implement interfaces.
- A struct cannot be `null`, and a struct variable cannot be assigned `null` unless the variable is declared as a nullable type.

See also

- C# Programming Guide
- Classes and Structs
- Classes
- Nullable Types
- Identifier names
- Using Structs
- How to: Know the Difference Between Passing a Struct and Passing a Class Reference to a Method

Using Structs (C# Programming Guide)

8/19/2019 • 3 minutes to read • Edit Online

The `struct` type is suitable for representing lightweight objects such as `Point`, `Rectangle`, and `Color`. Although it is just as convenient to represent a point as a class with Auto-Implemented Properties, a struct might be more efficient in some scenarios. For example, if you declare an array of 1000 `Point` objects, you will allocate additional memory for referencing each object; in this case, a struct would be less expensive. Because the .NET Framework contains an object called Point, the struct in this example is named "Coords" instead.

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

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

It is an error to define a default (parameterless) constructor for a struct. It is also an error to initialize an instance field in a struct body. You can initialize externally accessible struct members only by using a parameterized constructor, the implicit, parameterless constructor, an object initializer, or by accessing the members individually after the struct is declared. Any private or otherwise inaccessible members require the use of constructors exclusively.

When you create a struct object using the new operator, it gets created and the appropriate constructor is called according to the constructor's signature. Unlike classes, structs can be instantiated without using the `new` operator. In such a case, there is no constructor call, which makes the allocation more efficient. However, the fields will remain unassigned and the object cannot be used until all of the fields are initialized. This includes the inability to get or set values through properties.

If you instantiate a struct object using the default, parameterless constructor, all members are assigned according to their default values.

When writing a constructor with parameters for a struct, you must explicitly initialize all members; otherwise one or more members remain unassigned and the struct cannot be used, producing compiler error CS0171.

There is no inheritance for structs as there is for classes. A struct cannot inherit from another struct or class, and it cannot be the base of a class. Structs, however, inherit from the base class Object. A struct can implement interfaces, and it does that exactly as classes do.

You cannot declare a class using the keyword `struct`. In C#, classes and structs are semantically different. A struct is a value type, while a class is a reference type. For more information, see Value Types.

Unless you need reference-type semantics, a small class may be more efficiently handled by the system if you declare it as a struct instead.

Example 1

Description

This example demonstrates `struct` initialization using both default and parameterized constructors.

Code

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

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

```
// Declare and initialize struct objects.
class TestCoords
{
    static void Main()
    {
        // Initialize.
        var coords1 = new Coords();
        var coords2 = new Coords(10, 10);

        // Display results.
        Console.Write("Coords 1: ");
        Console.WriteLine($"x = {coords1.x}, y = {coords1.y}");

        Console.Write("Coords 2: ");
        Console.WriteLine($"x = {coords2.x}, y = {coords2.y}");

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
    Coords 1: x = 0, y = 0
    Coords 2: x = 10, y = 10
*/
```

Example 2

Description

This example demonstrates a feature that is unique to structs. It creates a Coords object without using the `new` operator. If you replace the word `struct` with the word `class`, the program will not compile.

Code

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

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

```
// Declare a struct object without "new".
class TestCoordsNoNew
{
    static void Main()
    {
        // Declare an object.
        Coords coords1;

        // Initialize.
        coords1.x = 10;
        coords1.y = 20;

        // Display results.
        Console.Write("Coords 1: ");
        Console.WriteLine($"x = {coords1.x}, y = {coords1.y}");

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
// Output: Coords 1: x = 10, y = 20
```

See also

- C# Programming Guide
- Classes and Structs
- Structs

Inheritance (C# Programming Guide)

8/22/2019 • 7 minutes to read • Edit Online

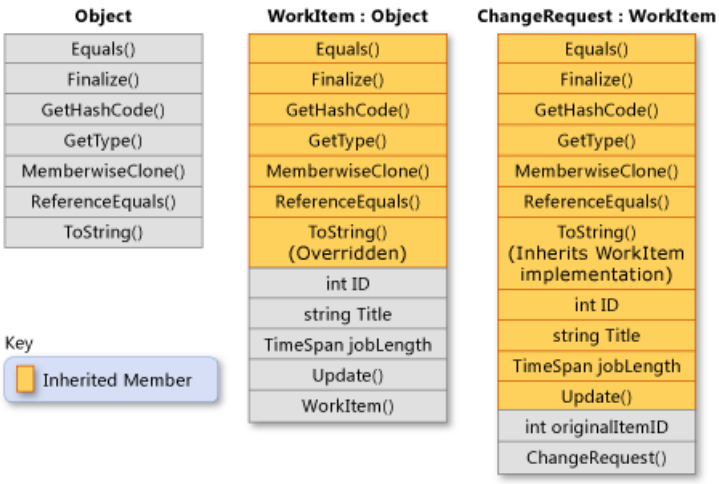
Inheritance, together with encapsulation and polymorphism, is one of the three primary characteristics of object-oriented programming. Inheritance enables you to create new classes that reuse, extend, and modify the behavior that is defined in other classes. The class whose members are inherited is called the *base class*, and the class that inherits those members is called the *derived class*. A derived class can have only one direct base class. However, inheritance is transitive. If ClassC is derived from ClassB, and ClassB is derived from ClassA, ClassC inherits the members declared in ClassB and ClassA.

> **NOTE**
>
> Structs do not support inheritance, but they can implement interfaces. For more information, see Interfaces.

Conceptually, a derived class is a specialization of the base class. For example, if you have a base class `Animal`, you might have one derived class that is named `Mammal` and another derived class that is named `Reptile`. A `Mammal` is an `Animal`, and a `Reptile` is an `Animal`, but each derived class represents different specializations of the base class.

When you define a class to derive from another class, the derived class implicitly gains all the members of the base class, except for its constructors and finalizers. The derived class can thereby reuse the code in the base class without having to re-implement it. In the derived class, you can add more members. In this manner, the derived class extends the functionality of the base class.

The following illustration shows a class `WorkItem` that represents an item of work in some business process. Like all classes, it derives from System.Object and inherits all its methods. `WorkItem` adds five members of its own. These include a constructor, because constructors are not inherited. Class `ChangeRequest` inherits from `WorkItem` and represents a particular kind of work item. `ChangeRequest` adds two more members to the members that it inherits from `WorkItem` and from Object. It must add its own constructor, and it also adds `originalItemID`. Property `originalItemID` enables the `ChangeRequest` instance to be associated with the original `WorkItem` to which the change request applies.



The following example shows how the class relationships demonstrated in the previous illustration are expressed in C#. The example also shows how `WorkItem` overrides the virtual method Object.ToString, and how the `ChangeRequest` class inherits the `WorkItem` implementation of the method.

```
// WorkItem implicitly inherits from the Object class.
```

```
// WorkItem implicitly inherits from the Object class.
public class WorkItem
{
    // Static field currentID stores the job ID of the last WorkItem that
    // has been created.
    private static int currentID;

    //Properties.
    protected int ID { get; set; }
    protected string Title { get; set; }
    protected string Description { get; set; }
    protected TimeSpan jobLength { get; set; }

    // Default constructor. If a derived class does not invoke a base-
    // class constructor explicitly, the default constructor is called
    // implicitly.
    public WorkItem()
    {
        ID = 0;
        Title = "Default title";
        Description = "Default description.";
        jobLength = new TimeSpan();
    }

    // Instance constructor that has three parameters.
    public WorkItem(string title, string desc, TimeSpan joblen)
    {
        this.ID = GetNextID();
        this.Title = title;
        this.Description = desc;
        this.jobLength = joblen;
    }

    // Static constructor to initialize the static member, currentID. This
    // constructor is called one time, automatically, before any instance
    // of WorkItem or ChangeRequest is created, or currentID is referenced.
    static WorkItem()
    {
        currentID = 0;
    }


    protected int GetNextID()
    {
        // currentID is a static field. It is incremented each time a new
        // instance of WorkItem is created.
        return ++currentID;
    }

    // Method Update enables you to update the title and job length of an
    // existing WorkItem object.
    public void Update(string title, TimeSpan joblen)
    {
        this.Title = title;
        this.jobLength = joblen;
    }

    // Virtual method override of the ToString method that is inherited
    // from System.Object.
    public override string ToString()
    {
        return $"{this.ID} - {this.Title}";
    }
}

// ChangeRequest derives from WorkItem and adds a property (originalItemID)
// and two constructors.
public class ChangeRequest : WorkItem
{
```

```
    protected int originalItemID { get; set; }

    // Constructors. Because neither constructor calls a base-class
    // constructor explicitly, the default constructor in the base class
    // is called implicitly. The base class must contain a default
    // constructor.

    // Default constructor for the derived class.
    public ChangeRequest() { }

    // Instance constructor that has four parameters.
    public ChangeRequest(string title, string desc, TimeSpan jobLen,
                         int originalID)
    {
        // The following properties and the GetNexID method are inherited
        // from WorkItem.
        this.ID = GetNextID();
        this.Title = title;
        this.Description = desc;
        this.jobLength = jobLen;

        // Property originalItemId is a member of ChangeRequest, but not
        // of WorkItem.
        this.originalItemID = originalID;
    }
}

class Program
{
    static void Main()
    {
        // Create an instance of WorkItem by using the constructor in the
        // base class that takes three arguments.
        WorkItem item = new WorkItem("Fix Bugs",
                                     "Fix all bugs in my code branch",
                                     new TimeSpan(3, 4, 0, 0));

        // Create an instance of ChangeRequest by using the constructor in
        // the derived class that takes four arguments.
        ChangeRequest change = new ChangeRequest("Change Base Class Design",
                                                 "Add members to the class",
                                                 new TimeSpan(4, 0, 0),
                                                 1);

        // Use the ToString method defined in WorkItem.
        Console.WriteLine(item.ToString());

        // Use the inherited Update method to change the title of the
        // ChangeRequest object.
        change.Update("Change the Design of the Base Class",
            new TimeSpan(4, 0, 0));

        // ChangeRequest inherits WorkItem's override of ToString.
        Console.WriteLine(change.ToString());

        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
    1 - Fix Bugs
    2 - Change the Design of the Base Class
*/
```

Abstract and Virtual Methods

When a base class declares a method as virtual, a derived class can override the method with its own implementation. If a base class declares a member as abstract, that method must be overridden in any non-abstract class that directly inherits from that class. If a derived class is itself abstract, it inherits abstract members without implementing them. Abstract and virtual members are the basis for polymorphism, which is the second primary characteristic of object-oriented programming. For more information, see Polymorphism.

Abstract Base Classes

You can declare a class as abstract if you want to prevent direct instantiation by using the new operator. If you do this, the class can be used only if a new class is derived from it. An abstract class can contain one or more method signatures that themselves are declared as abstract. These signatures specify the parameters and return value but have no implementation (method body). An abstract class does not have to contain abstract members; however, if a class does contain an abstract member, the class itself must be declared as abstract. Derived classes that are not abstract themselves must provide the implementation for any abstract methods from an abstract base class. For more information, see Abstract and Sealed Classes and Class Members.

Interfaces

An *interface* is a reference type that is somewhat similar to an abstract base class that consists of only abstract members. When a class implements an interface, it must provide an implementation for all the members of the interface. A class can implement multiple interfaces even though it can derive from only a single direct base class.

Interfaces are used to define specific capabilities for classes that do not necessarily have an "is a" relationship. For example, the System.IEquatable<T> interface can be implemented by any class or struct that has to enable client code to determine whether two objects of the type are equivalent (however the type defines equivalence). IEquatable<T> does not imply the same kind of "is a" relationship that exists between a base class and a derived class (for example, a `Mammal` is an `Animal`). For more information, see Interfaces.

Preventing Further Derivation

A class can prevent other classes from inheriting from it, or from any of its members, by declaring itself or the member as sealed. For more information, see Abstract and Sealed Classes and Class Members.

Derived Class Hiding of Base Class Members

A derived class can hide base class members by declaring members with the same name and signature. The new modifier can be used to explicitly indicate that the member is not intended to be an override of the base member. The use of new is not required, but a compiler warning will be generated if new is not used. For more information, see Versioning with the Override and New Keywords and Knowing When to Use Override and New Keywords.

See also

- C# Programming Guide
- Classes and Structs
- class
- struct

Polymorphism (C# Programming Guide)

8/22/2019 • 7 minutes to read • Edit Online

Polymorphism is often referred to as the third pillar of object-oriented programming, after encapsulation and inheritance. Polymorphism is a Greek word that means "many-shaped" and it has two distinct aspects:

- At run time, objects of a derived class may be treated as objects of a base class in places such as method parameters and collections or arrays. When this occurs, the object's declared type is no longer identical to its run-time type.
- Base classes may define and implement virtual *methods*, and derived classes can override them, which means they provide their own definition and implementation. At run-time, when client code calls the method, the CLR looks up the run-time type of the object, and invokes that override of the virtual method. Thus in your source code you can call a method on a base class, and cause a derived class's version of the method to be executed.

Virtual methods enable you to work with groups of related objects in a uniform way. For example, suppose you have a drawing application that enables a user to create various kinds of shapes on a drawing surface. You do not know at compile time which specific types of shapes the user will create. However, the application has to keep track of all the various types of shapes that are created, and it has to update them in response to user mouse actions. You can use polymorphism to solve this problem in two basic steps:

1. Create a class hierarchy in which each specific shape class derives from a common base class.
2. Use a virtual method to invoke the appropriate method on any derived class through a single call to the base class method.

First, create a base class called `Shape`, and derived classes such as `Rectangle`, `Circle`, and `Triangle`. Give the `Shape` class a virtual method called `Draw`, and override it in each derived class to draw the particular shape that the class represents. Create a `List<Shape>` object and add a Circle, Triangle and Rectangle to it. To update the drawing surface, use a foreach loop to iterate through the list and call the `Draw` method on each `Shape` object in the list. Even though each object in the list has a declared type of `Shape`, it is the run-time type (the overridden version of the method in each derived class) that will be invoked.

```
using System;
using System.Collections.Generic;

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

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

class Circle : Shape
{
    public override void Draw()
    {
```

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

class Program
{
    static void Main(string[] args)
    {
        // Polymorphism at work #1: a Rectangle, Triangle and Circle
        // can all be used whereever a Shape is expected. No cast is
        // required because an implicit conversion exists from a derived
        // class to its base class.
        var shapes = new List<Shape>
        {
            new Rectangle(),
            new Triangle(),
            new Circle()
        };

        // Polymorphism at work #2: the virtual method Draw is
        // invoked on each of the derived classes, not the base class.
        foreach (var shape in shapes)
        {
            shape.Draw();
        }

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

}

/* Output:
    Drawing a rectangle
    Performing base class drawing tasks
    Drawing a triangle
    Performing base class drawing tasks
    Drawing a circle
    Performing base class drawing tasks
 */
```

In C#, every type is polymorphic because all types, including user-defined types, inherit from Object.

Polymorphism Overview

Virtual Members

When a derived class inherits from a base class, it gains all the methods, fields, properties and events of the base class. The designer of the derived class can choose whether to

- override virtual members in the base class,
- inherit the closest base class method without overriding it
- define new non-virtual implementation of those members that hide the base class implementations

A derived class can override a base class member only if the base class member is declared as virtual or abstract. The derived member must use the override keyword to explicitly indicate that the method is intended to participate in virtual invocation. The following code provides an example:

```
public class BaseClass
{
    public virtual void DoWork() { }
    public virtual int WorkProperty
    {
        get { return 0; }
    }
}
public class DerivedClass : BaseClass
{
    public override void DoWork() { }
    public override int WorkProperty
    {
        get { return 0; }
    }
}
```

Fields cannot be virtual; only methods, properties, events and indexers can be virtual. When a derived class overrides a virtual member, that member is called even when an instance of that class is being accessed as an instance of the base class. The following code provides an example:

```
DerivedClass B = new DerivedClass();
B.DoWork();  // Calls the new method.

BaseClass A = (BaseClass)B;
A.DoWork();  // Also calls the new method.
```

Virtual methods and properties enable derived classes to extend a base class without needing to use the base class implementation of a method. For more information, see Versioning with the Override and New Keywords. An interface provides another way to define a method or set of methods whose implementation is left to derived classes. For more information, see Interfaces.

Hiding Base Class Members with New Members

If you want your derived member to have the same name as a member in a base class, but you do not want it to participate in virtual invocation, you can use the new keyword. The `new` keyword is put before the return type of a class member that is being replaced. The following code provides an example:

```
public class BaseClass
{
    public void DoWork() { WorkField++; }
    public int WorkField;
    public int WorkProperty
    {
        get { return 0; }
    }
}

public class DerivedClass : BaseClass
{
    public new void DoWork() { WorkField++; }
    public new int WorkField;
    public new int WorkProperty
    {
        get { return 0; }
    }
}
```

Hidden base class members can still be accessed from client code by casting the instance of the derived class to an instance of the base class. For example:

```
DerivedClass B = new DerivedClass();
B.DoWork();  // Calls the new method.

BaseClass A = (BaseClass)B;
A.DoWork();  // Calls the old method.
```

Preventing Derived Classes from Overriding Virtual Members

Virtual members remain virtual indefinitely, regardless of how many classes have been declared between the virtual member and the class that originally declared it. If class A declares a virtual member, and class B derives from A, and class C derives from B, class C inherits the virtual member, and has the option to override it, regardless of whether class B declared an override for that member. The following code provides an example:

```
public class A
{
    public virtual void DoWork() { }
}
public class B : A
{
    public override void DoWork() { }
}
```

A derived class can stop virtual inheritance by declaring an override as sealed. This requires putting the `sealed` keyword before the `override` keyword in the class member declaration. The following code provides an example:

```
public class C : B
{
    public sealed override void DoWork() { }
}
```

In the previous example, the method `DoWork` is no longer virtual to any class derived from C. It is still virtual for instances of C, even if they are cast to type B or type A. Sealed methods can be replaced by derived classes by using the `new` keyword, as the following example shows:

```
public class D : C
{
    public new void DoWork() { }
}
```

In this case, if `DoWork` is called on D using a variable of type D, the new `DoWork` is called. If a variable of type C, B, or A is used to access an instance of D, a call to `DoWork` will follow the rules of virtual inheritance, routing those calls to the implementation of `DoWork` on class C.

Accessing Base Class Virtual Members from Derived Classes

A derived class that has replaced or overridden a method or property can still access the method or property on the base class using the `base` keyword. The following code provides an example:

```
public class Base
{
    public virtual void DoWork() {/*...*/ }
}
public class Derived : Base
{
    public override void DoWork()
    {
        //Perform Derived's work here
        //...
        // Call DoWork on base class
        base.DoWork();
    }
}
```

For more information, see base.

> **NOTE**
>
> It is recommended that virtual members use `base` to call the base class implementation of that member in their own implementation. Letting the base class behavior occur enables the derived class to concentrate on implementing behavior specific to the derived class. If the base class implementation is not called, it is up to the derived class to make their behavior compatible with the behavior of the base class.

In This Section

- Versioning with the Override and New Keywords
- Knowing When to Use Override and New Keywords
- How to: Override the ToString Method

See also

- C# Programming Guide
- Inheritance
- Abstract and Sealed Classes and Class Members
- Methods
- Events
- Properties
- Indexers
- Types

Versioning with the Override and New Keywords (C# Programming Guide)

8/19/2019 • 5 minutes to read • Edit Online

The C# language is designed so that versioning between base and derived classes in different libraries can evolve and maintain backward compatibility. This means, for example, that the introduction of a new member in a base class with the same name as a member in a derived class is completely supported by C# and does not lead to unexpected behavior. It also means that a class must explicitly state whether a method is intended to override an inherited method, or whether a method is a new method that hides a similarly named inherited method.

In C#, derived classes can contain methods with the same name as base class methods.

- The base class method must be defined virtual.
- If the method in the derived class is not preceded by new or override keywords, the compiler will issue a warning and the method will behave as if the `new` keyword were present.
- If the method in the derived class is preceded with the `new` keyword, the method is defined as being independent of the method in the base class.
- If the method in the derived class is preceded with the `override` keyword, objects of the derived class will call that method instead of the base class method.
- The base class method can be called from within the derived class using the `base` keyword.
- The `override`, `virtual`, and `new` keywords can also be applied to properties, indexers, and events.

By default, C# methods are not virtual. If a method is declared as virtual, any class inheriting the method can implement its own version. To make a method virtual, the `virtual` modifier is used in the method declaration of the base class. The derived class can then override the base virtual method by using the `override` keyword or hide the virtual method in the base class by using the `new` keyword. If neither the `override` keyword nor the `new` keyword is specified, the compiler will issue a warning and the method in the derived class will hide the method in the base class.

To demonstrate this in practice, assume for a moment that Company A has created a class named `GraphicsClass`, which your program uses. The following is `GraphicsClass`:

```
class GraphicsClass
{
    public virtual void DrawLine() { }
    public virtual void DrawPoint() { }
}
```

Your company uses this class, and you use it to derive your own class, adding a new method:

```
class YourDerivedGraphicsClass : GraphicsClass
{
    public void DrawRectangle() { }
}
```

Your application is used without problems, until Company A releases a new version of `GraphicsClass`, which resembles the following code:

```
class GraphicsClass
{
    public virtual void DrawLine() { }
    public virtual void DrawPoint() { }
    public virtual void DrawRectangle() { }
}
```

The new version of `GraphicsClass` now contains a method named `DrawRectangle`. Initially, nothing occurs. The new version is still binary compatible with the old version. Any software that you have deployed will continue to work, even if the new class is installed on those computer systems. Any existing calls to the method `DrawRectangle` will continue to reference your version, in your derived class.

However, as soon as you recompile your application by using the new version of `GraphicsClass`, you will receive a warning from the compiler, CS0108. This warning informs you that you have to consider how you want your `DrawRectangle` method to behave in your application.

If you want your method to override the new base class method, use the `override` keyword:

```
class YourDerivedGraphicsClass : GraphicsClass
{
    public override void DrawRectangle() { }
}
```

The `override` keyword makes sure that any objects derived from `YourDerivedGraphicsClass` will use the derived class version of `DrawRectangle`. Objects derived from `YourDerivedGraphicsClass` can still access the base class version of `DrawRectangle` by using the base keyword:

```
base.DrawRectangle();
```

If you do not want your method to override the new base class method, the following considerations apply. To avoid confusion between the two methods, you can rename your method. This can be time-consuming and error-prone, and just not practical in some cases. However, if your project is relatively small, you can use Visual Studio's Refactoring options to rename the method. For more information, see Refactoring Classes and Types (Class Designer).

Alternatively, you can prevent the warning by using the keyword `new` in your derived class definition:

```
class YourDerivedGraphicsClass : GraphicsClass
{
    public new void DrawRectangle() { }
}
```

Using the `new` keyword tells the compiler that your definition hides the definition that is contained in the base class. This is the default behavior.

Override and Method Selection

When a method is named on a class, the C# compiler selects the best method to call if more than one method is compatible with the call, such as when there are two methods with the same name, and parameters that are compatible with the parameter passed. The following methods would be compatible:

```
public class Derived : Base
{
    public override void DoWork(int param) { }
    public void DoWork(double param) { }
}
```

When `DoWork` is called on an instance of `Derived`, the C# compiler will first try to make the call compatible with the versions of `DoWork` declared originally on `Derived`. Override methods are not considered as declared on a class, they are new implementations of a method declared on a base class. Only if the C# compiler cannot match the method call to an original method on `Derived` will it try to match the call to an overridden method with the same name and compatible parameters. For example:

```
int val = 5;
Derived d = new Derived();
d.DoWork(val);  // Calls DoWork(double).
```

Because the variable `val` can be converted to a double implicitly, the C# compiler calls `DoWork(double)` instead of `DoWork(int)`. There are two ways to avoid this. First, avoid declaring new methods with the same name as virtual methods. Second, you can instruct the C# compiler to call the virtual method by making it search the base class method list by casting the instance of `Derived` to `Base`. Because the method is virtual, the implementation of `DoWork(int)` on `Derived` will be called. For example:

```
((Base)d).DoWork(val);  // Calls DoWork(int) on Derived.
```

For more examples of `new` and `override`, see Knowing When to Use Override and New Keywords.

See also

- C# Programming Guide
- Classes and Structs
- Methods
- Inheritance

Knowing When to Use Override and New Keywords (C# Programming Guide)

8/19/2019 • 10 minutes to read • Edit Online

In C#, a method in a derived class can have the same name as a method in the base class. You can specify how the methods interact by using the new and override keywords. The `override` modifier *extends* the base class `virtual` method, and the `new` modifier *hides* an accessible base class method. The difference is illustrated in the examples in this topic.

In a console application, declare the following two classes, `BaseClass` and `DerivedClass`. `DerivedClass` inherits from `BaseClass`.

```
class BaseClass
{
    public void Method1()
    {
        Console.WriteLine("Base - Method1");
    }
}

class DerivedClass : BaseClass
{
    public void Method2()
    {
        Console.WriteLine("Derived - Method2");
    }
}
```

In the `Main` method, declare variables `bc`, `dc`, and `bcdc`.

- `bc` is of type `BaseClass`, and its value is of type `BaseClass`.
- `dc` is of type `DerivedClass`, and its value is of type `DerivedClass`.
- `bcdc` is of type `BaseClass`, and its value is of type `DerivedClass`. This is the variable to pay attention to.

Because `bc` and `bcdc` have type `BaseClass`, they can only directly access `Method1`, unless you use casting. Variable `dc` can access both `Method1` and `Method2`. These relationships are shown in the following code.

```
class Program
{
    static void Main(string[] args)
    {
        BaseClass bc = new BaseClass();
        DerivedClass dc = new DerivedClass();
        BaseClass bcdc = new DerivedClass();

        bc.Method1();
        dc.Method1();
        dc.Method2();
        bcdc.Method1();
    }
    // Output:
    // Base - Method1
    // Base - Method1
    // Derived - Method2
    // Base - Method1
}
```

Next, add the following `Method2` method to `BaseClass`. The signature of this method matches the signature of the `Method2` method in `DerivedClass`.

```
public void Method2()
{
    Console.WriteLine("Base - Method2");
}
```

Because `BaseClass` now has a `Method2` method, a second calling statement can be added for `BaseClass` variables `bc` and `bcdc`, as shown in the following code.

```
bc.Method1();
bc.Method2();
dc.Method1();
dc.Method2();
bcdc.Method1();
bcdc.Method2();
```

When you build the project, you see that the addition of the `Method2` method in `BaseClass` causes a warning. The warning says that the `Method2` method in `DerivedClass` hides the `Method2` method in `BaseClass`. You are advised to use the `new` keyword in the `Method2` definition if you intend to cause that result. Alternatively, you could rename one of the `Method2` methods to resolve the warning, but that is not always practical.

Before adding `new`, run the program to see the output produced by the additional calling statements. The following results are displayed.

```
// Output:
// Base - Method1
// Base - Method2
// Base - Method1
// Derived - Method2
// Base - Method1
// Base - Method2
```

The `new` keyword preserves the relationships that produce that output, but it suppresses the warning. The variables that have type `BaseClass` continue to access the members of `BaseClass`, and the variable that has type `DerivedClass` continues to access members in `DerivedClass` first, and then to consider members inherited from

`BaseClass`.

To suppress the warning, add the `new` modifier to the definition of `Method2` in `DerivedClass`, as shown in the following code. The modifier can be added before or after `public`.

```
public new void Method2()
{
    Console.WriteLine("Derived - Method2");
}
```

Run the program again to verify that the output has not changed. Also verify that the warning no longer appears. By using `new`, you are asserting that you are aware that the member that it modifies hides a member that is inherited from the base class. For more information about name hiding through inheritance, see new Modifier.

To contrast this behavior to the effects of using `override`, add the following method to `DerivedClass`. The `override` modifier can be added before or after `public`.

```
public override void Method1()
{
    Console.WriteLine("Derived - Method1");
}
```

Add the `virtual` modifier to the definition of `Method1` in `BaseClass`. The `virtual` modifier can be added before or after `public`.

```
public virtual void Method1()
{
    Console.WriteLine("Base - Method1");
}
```

Run the project again. Notice especially the last two lines of the following output.

```
// Output:
// Base - Method1
// Base - Method2
// Derived - Method1
// Derived - Method2
// Derived - Method1
// Base - Method2
```

The use of the `override` modifier enables `bcdc` to access the `Method1` method that is defined in `DerivedClass`. Typically, that is the desired behavior in inheritance hierarchies. You want objects that have values that are created from the derived class to use the methods that are defined in the derived class. You achieve that behavior by using `override` to extend the base class method.

The following code contains the full example.

```
using System;
using System.Text;

namespace OverrideAndNew
{
    class Program
    {
        static void Main(string[] args)
        {
            BaseClass bc = new BaseClass();
            DerivedClass dc = new DerivedClass();
            BaseClass bcdc = new DerivedClass();

            // The following two calls do what you would expect. They call
            // the methods that are defined in BaseClass.
            bc.Method1();
            bc.Method2();
            // Output:
            // Base - Method1
            // Base - Method2

            // The following two calls do what you would expect. They call
            // the methods that are defined in DerivedClass.
            dc.Method1();
            dc.Method2();
            // Output:
            // Derived - Method1
            // Derived - Method2

            // The following two calls produce different results, depending
            // on whether override (Method1) or new (Method2) is used.
            bcdc.Method1();
            bcdc.Method2();
            // Output:
            // Derived - Method1
            // Base - Method2
        }
    }

    class BaseClass
    {
        public virtual void Method1()
        {
            Console.WriteLine("Base - Method1");
        }

        public virtual void Method2()
        {
            Console.WriteLine("Base - Method2");
        }
    }

    class DerivedClass : BaseClass
    {
        public override void Method1()
        {
            Console.WriteLine("Derived - Method1");
        }

        public new void Method2()
        {
            Console.WriteLine("Derived - Method2");
        }
    }
}
```

The following example illustrates similar behavior in a different context. The example defines three classes: a base class named `Car` and two classes that are derived from it, `ConvertibleCar` and `Minivan`. The base class contains a `DescribeCar` method. The method displays a basic description of a car, and then calls `ShowDetails` to provide additional information. Each of the three classes defines a `ShowDetails` method. The `new` modifier is used to define `ShowDetails` in the `ConvertibleCar` class. The `override` modifier is used to define `ShowDetails` in the `Minivan` class.

```
// Define the base class, Car. The class defines two methods,
// DescribeCar and ShowDetails. DescribeCar calls ShowDetails, and each derived
// class also defines a ShowDetails method. The example tests which version of
// ShowDetails is selected, the base class method or the derived class method.
class Car
{
    public void DescribeCar()
    {
        System.Console.WriteLine("Four wheels and an engine.");
        ShowDetails();
    }

    public virtual void ShowDetails()
    {
        System.Console.WriteLine("Standard transportation.");
    }
}

// Define the derived classes.

// Class ConvertibleCar uses the new modifier to acknowledge that ShowDetails
// hides the base class method.
class ConvertibleCar : Car
{
    public new void ShowDetails()
    {
        System.Console.WriteLine("A roof that opens up.");
    }
}

// Class Minivan uses the override modifier to specify that ShowDetails
// extends the base class method.
class Minivan : Car
{
    public override void ShowDetails()
    {
        System.Console.WriteLine("Carries seven people.");
    }
}
```

The example tests which version of `ShowDetails` is called. The following method, `TestCars1`, declares an instance of each class, and then calls `DescribeCar` on each instance.

```
public static void TestCars1()
{
    System.Console.WriteLine("\nTestCars1");
    System.Console.WriteLine("----------");

    Car car1 = new Car();
    car1.DescribeCar();
    System.Console.WriteLine("----------");

    // Notice the output from this test case. The new modifier is
    // used in the definition of ShowDetails in the ConvertibleCar
    // class.

    ConvertibleCar car2 = new ConvertibleCar();
    car2.DescribeCar();
    System.Console.WriteLine("----------");

    Minivan car3 = new Minivan();
    car3.DescribeCar();
    System.Console.WriteLine("----------");
}
```

`TestCars1` produces the following output. Notice especially the results for `car2`, which probably are not what you expected. The type of the object is `ConvertibleCar`, but `DescribeCar` does not access the version of `ShowDetails` that is defined in the `ConvertibleCar` class because that method is declared with the `new` modifier, not the `override` modifier. As a result, a `ConvertibleCar` object displays the same description as a `Car` object. Contrast the results for `car3`, which is a `Minivan` object. In this case, the `ShowDetails` method that is declared in the `Minivan` class overrides the `ShowDetails` method that is declared in the `Car` class, and the description that is displayed describes a minivan.

```
// TestCars1
// ----------
// Four wheels and an engine.
// Standard transportation.
// ----------
// Four wheels and an engine.
// Standard transportation.
// ----------
// Four wheels and an engine.
// Carries seven people.
// ----------
```

`TestCars2` creates a list of objects that have type `Car`. The values of the objects are instantiated from the `Car`, `ConvertibleCar`, and `Minivan` classes. `DescribeCar` is called on each element of the list. The following code shows the definition of `TestCars2`.

```
public static void TestCars2()
{
    System.Console.WriteLine("\nTestCars2");
    System.Console.WriteLine("----------");

    var cars = new List<Car> { new Car(), new ConvertibleCar(),
        new Minivan() };

    foreach (var car in cars)
    {
        car.DescribeCar();
        System.Console.WriteLine("----------");
    }
}
```

The following output is displayed. Notice that it is the same as the output that is displayed by `TestCars1`. The `ShowDetails` method of the `ConvertibleCar` class is not called, regardless of whether the type of the object is `ConvertibleCar`, as in `TestCars1`, or `Car`, as in `TestCars2`. Conversely, `car3` calls the `ShowDetails` method from the `Minivan` class in both cases, whether it has type `Minivan` or type `Car`.

```
// TestCars2
// ----------
// Four wheels and an engine.
// Standard transportation.
// ----------
// Four wheels and an engine.
// Standard transportation.
// ----------
// Four wheels and an engine.
// Carries seven people.
// ----------
```

Methods `TestCars3` and `TestCars4` complete the example. These methods call `ShowDetails` directly, first from objects declared to have type `ConvertibleCar` and `Minivan` (`TestCars3`), then from objects declared to have type `Car` (`TestCars4`). The following code defines these two methods.

```
public static void TestCars3()
{
    System.Console.WriteLine("\nTestCars3");
    System.Console.WriteLine("----------");
    ConvertibleCar car2 = new ConvertibleCar();
    Minivan car3 = new Minivan();
    car2.ShowDetails();
    car3.ShowDetails();
}

public static void TestCars4()
{
    System.Console.WriteLine("\nTestCars4");
    System.Console.WriteLine("----------");
    Car car2 = new ConvertibleCar();
    Car car3 = new Minivan();
    car2.ShowDetails();
    car3.ShowDetails();
}
```

The methods produce the following output, which corresponds to the results from the first example in this topic.

```
// TestCars3
// ----------
// A roof that opens up.
// Carries seven people.

// TestCars4
// ----------
// Standard transportation.
// Carries seven people.
```

The following code shows the complete project and its output.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace OverrideAndNew2
```

```
namespace OverrideAndNew2
{
    class Program
    {
        static void Main(string[] args)
        {
            // Declare objects of the derived classes and test which version
            // of ShowDetails is run, base or derived.
            TestCars1();

            // Declare objects of the base class, instantiated with the
            // derived classes, and repeat the tests.
            TestCars2();

            // Declare objects of the derived classes and call ShowDetails
            // directly.
            TestCars3();

            // Declare objects of the base class, instantiated with the
            // derived classes, and repeat the tests.
            TestCars4();
        }

        public static void TestCars1()
        {
            System.Console.WriteLine("\nTestCars1");
            System.Console.WriteLine("----------");

            Car car1 = new Car();
            car1.DescribeCar();
            System.Console.WriteLine("----------");

            // Notice the output from this test case. The new modifier is
            // used in the definition of ShowDetails in the ConvertibleCar
            // class.
            ConvertibleCar car2 = new ConvertibleCar();
            car2.DescribeCar();
            System.Console.WriteLine("----------");

            Minivan car3 = new Minivan();
            car3.DescribeCar();
            System.Console.WriteLine("----------");
        }
        // Output:
        // TestCars1
        // ----------
        // Four wheels and an engine.
        // Standard transportation.
        // ----------
        // Four wheels and an engine.
        // Standard transportation.
        // ----------
        // Four wheels and an engine.
        // Carries seven people.
        // ----------

        public static void TestCars2()
        {
            System.Console.WriteLine("\nTestCars2");
            System.Console.WriteLine("----------");

            var cars = new List<Car> { new Car(), new ConvertibleCar(),
                new Minivan() };

            foreach (var car in cars)
            {
                car.DescribeCar();
                System.Console.WriteLine("----------");
            }
```

```
        }
        // Output:
        // TestCars2
        // ----------
        // Four wheels and an engine.
        // Standard transportation.
        // ----------
        // Four wheels and an engine.
        // Standard transportation.
        // ----------
        // Four wheels and an engine.
        // Carries seven people.
        // ----------

        public static void TestCars3()
        {
            System.Console.WriteLine("\nTestCars3");
            System.Console.WriteLine("----------");
            ConvertibleCar car2 = new ConvertibleCar();
            Minivan car3 = new Minivan();
            car2.ShowDetails();
            car3.ShowDetails();
        }
        // Output:
        // TestCars3
        // ----------
        // A roof that opens up.
        // Carries seven people.

        public static void TestCars4()
        {
            System.Console.WriteLine("\nTestCars4");
            System.Console.WriteLine("----------");
            Car car2 = new ConvertibleCar();
            Car car3 = new Minivan();
            car2.ShowDetails();
            car3.ShowDetails();
        }
        // Output:
        // TestCars4
        // ----------
        // Standard transportation.
        // Carries seven people.
    }

    // Define the base class, Car. The class defines two virtual methods,
    // DescribeCar and ShowDetails. DescribeCar calls ShowDetails, and each derived
    // class also defines a ShowDetails method. The example tests which version of
    // ShowDetails is used, the base class method or the derived class method.
    class Car
    {
        public virtual void DescribeCar()
        {
            System.Console.WriteLine("Four wheels and an engine.");
            ShowDetails();
        }

        public virtual void ShowDetails()
        {
            System.Console.WriteLine("Standard transportation.");
        }
    }

    // Define the derived classes.

    // Class ConvertibleCar uses the new modifier to acknowledge that ShowDetails
    // hides the base class method.
    class ConvertibleCar : Car
    {
```

```
        public new void ShowDetails()
        {
            System.Console.WriteLine("A roof that opens up.");
        }
    }

    // Class Minivan uses the override modifier to specify that ShowDetails
    // extends the base class method.
    class Minivan : Car
    {
        public override void ShowDetails()
        {
            System.Console.WriteLine("Carries seven people.");
        }
    }

}
```

See also

- C# Programming Guide
- Classes and Structs
- Versioning with the Override and New Keywords
- base
- abstract

How to: Override the ToString Method (C# Programming Guide)

8/19/2019 • 2 minutes to read • Edit Online

Every class or struct in C# implicitly inherits the Object class. Therefore, every object in C# gets the ToString method, which returns a string representation of that object. For example, all variables of type `int` have a `ToString` method, which enables them to return their contents as a string:

```
int x = 42;
string strx = x.ToString();
Console.WriteLine(strx);
// Output:
// 42
```

When you create a custom class or struct, you should override the ToString method in order to provide information about your type to client code.

For information about how to use format strings and other types of custom formatting with the `ToString` method, see Formatting Types.

> **IMPORTANT**
>
> When you decide what information to provide through this method, consider whether your class or struct will ever be used by untrusted code. Be careful to ensure that you do not provide any information that could be exploited by malicious code.

To override the `ToString` method in your class or struct:

1. Declare a `ToString` method with the following modifiers and return type:

   ```
   public override string ToString(){}
   ```

2. Implement the method so that it returns a string.

   The following example returns the name of the class in addition to the data specific to a particular instance of the class.

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

       public override string ToString()
       {
           return "Person: " + Name + " " + Age;
       }
   }
   ```

   You can test the `ToString` method as shown in the following code example:

```
Person person = new Person { Name = "John", Age = 12 };
Console.WriteLine(person);
// Output:
// Person: John 12
```

See also

- IFormattable
- C# Programming Guide
- Classes and Structs
- Strings
- string
- override
- virtual
- Formatting Types

Members (C# Programming Guide)

8/22/2019 • 2 minutes to read • Edit Online

Classes and structs have members that represent their data and behavior. A class's members include all the members declared in the class, along with all members (except constructors and finalizers) declared in all classes in its inheritance hierarchy. Private members in base classes are inherited but are not accessible from derived classes.

The following table lists the kinds of members a class or struct may contain:

MEMBER	DESCRIPTION
Fields	Fields are variables declared at class scope. A field may be a built-in numeric type or an instance of another class. For example, a calendar class may have a field that contains the current date.
Constants	Constants are fields whose value is set at compile time and cannot be changed.
Properties	Properties are methods on a class that are accessed as if they were fields on that class. A property can provide protection for a class field to keep it from being changed without the knowledge of the object.
Methods	Methods define the actions that a class can perform. Methods can take parameters that provide input data, and can return output data through parameters. Methods can also return a value directly, without using a parameter.
Events	Events provide notifications about occurrences, such as button clicks or the successful completion of a method, to other objects. Events are defined and triggered by using delegates.
Operators	Overloaded operators are considered type members. When you overload an operator, you define it as a public static method in a type. For more information, see Operator overloading.
Indexers	Indexers enable an object to be indexed in a manner similar to arrays.
Constructors	Constructors are methods that are called when the object is first created. They are often used to initialize the data of an object.
Finalizers	Finalizers are used very rarely in C#. They are methods that are called by the runtime execution engine when the object is about to be removed from memory. They are generally used to make sure that any resources which must be released are handled appropriately.
Nested Types	Nested types are types declared within another type. Nested types are often used to describe objects that are used only by the types that contain them.

See also

- C# Programming Guide
- Classes

Abstract and Sealed Classes and Class Members (C# Programming Guide)

8/19/2019 • 2 minutes to read • Edit Online

The abstract keyword enables you to create classes and class members that are incomplete and must be implemented in a derived class.

The sealed keyword enables you to prevent the inheritance of a class or certain class members that were previously marked virtual.

Abstract Classes and Class Members

Classes can be declared as abstract by putting the keyword `abstract` before the class definition. For example:

```
public abstract class A
{
    // Class members here.
}
```

An abstract class cannot be instantiated. The purpose of an abstract class is to provide a common definition of a base class that multiple derived classes can share. For example, a class library may define an abstract class that is used as a parameter to many of its functions, and require programmers using that library to provide their own implementation of the class by creating a derived class.

Abstract classes may also define abstract methods. This is accomplished by adding the keyword `abstract` before the return type of the method. For example:

```
public abstract class A
{
    public abstract void DoWork(int i);
}
```

Abstract methods have no implementation, so the method definition is followed by a semicolon instead of a normal method block. Derived classes of the abstract class must implement all abstract methods. When an abstract class inherits a virtual method from a base class, the abstract class can override the virtual method with an abstract method. For example:

```
// compile with: -target:library
public class D
{
    public virtual void DoWork(int i)
    {
        // Original implementation.
    }
}

public abstract class E : D
{
    public abstract override void DoWork(int i);
}

public class F : E
{
    public override void DoWork(int i)
    {
        // New implementation.
    }
}
```

If a `virtual` method is declared `abstract`, it is still virtual to any class inheriting from the abstract class. A class inheriting an abstract method cannot access the original implementation of the method—in the previous example, `DoWork` on class F cannot call `DoWork` on class D. In this way, an abstract class can force derived classes to provide new method implementations for virtual methods.

Sealed Classes and Class Members

Classes can be declared as sealed by putting the keyword `sealed` before the class definition. For example:

```
public sealed class D
{
    // Class members here.
}
```

A sealed class cannot be used as a base class. For this reason, it cannot also be an abstract class. Sealed classes prevent derivation. Because they can never be used as a base class, some run-time optimizations can make calling sealed class members slightly faster.

A method, indexer, property, or event, on a derived class that is overriding a virtual member of the base class can declare that member as sealed. This negates the virtual aspect of the member for any further derived class. This is accomplished by putting the `sealed` keyword before the override keyword in the class member declaration. For example:

```
public class D : C
{
    public sealed override void DoWork() { }
}
```

See also

- C# Programming Guide
- Classes and Structs
- Inheritance
- Methods

- Fields
- How to: Define Abstract Properties

Static Classes and Static Class Members (C# Programming Guide)

8/22/2019 • 5 minutes to read • Edit Online

A static class is basically the same as a non-static class, but there is one difference: a static class cannot be instantiated. In other words, you cannot use the new operator to create a variable of the class type. Because there is no instance variable, you access the members of a static class by using the class name itself. For example, if you have a static class that is named `UtilityClass` that has a public static method named `MethodA`, you call the method as shown in the following example:

```
UtilityClass.MethodA();
```

A static class can be used as a convenient container for sets of methods that just operate on input parameters and do not have to get or set any internal instance fields. For example, in the .NET Framework Class Library, the static System.Math class contains methods that perform mathematical operations, without any requirement to store or retrieve data that is unique to a particular instance of the Math class. That is, you apply the members of the class by specifying the class name and the method name, as shown in the following example.

```
double dub = -3.14;
Console.WriteLine(Math.Abs(dub));
Console.WriteLine(Math.Floor(dub));
Console.WriteLine(Math.Round(Math.Abs(dub)));

// Output:
// 3.14
// -4
// 3
```

As is the case with all class types, the type information for a static class is loaded by the .NET Framework common language runtime (CLR) when the program that references the class is loaded. The program cannot specify exactly when the class is loaded. However, it is guaranteed to be loaded and to have its fields initialized and its static constructor called before the class is referenced for the first time in your program. A static constructor is only called one time, and a static class remains in memory for the lifetime of the application domain in which your program resides.

> **NOTE**
> To create a non-static class that allows only one instance of itself to be created, see Implementing Singleton in C#.

The following list provides the main features of a static class:

- Contains only static members.
- Cannot be instantiated.
- Is sealed.
- Cannot contain Instance Constructors.

Creating a static class is therefore basically the same as creating a class that contains only static members and a private constructor. A private constructor prevents the class from being instantiated. The advantage of using a

static class is that the compiler can check to make sure that no instance members are accidentally added. The compiler will guarantee that instances of this class cannot be created.

Static classes are sealed and therefore cannot be inherited. They cannot inherit from any class except Object. Static classes cannot contain an instance constructor; however, they can contain a static constructor. Non-static classes should also define a static constructor if the class contains static members that require non-trivial initialization. For more information, see Static Constructors.

Example

Here is an example of a static class that contains two methods that convert temperature from Celsius to Fahrenheit and from Fahrenheit to Celsius:

```
public static class TemperatureConverter
{
    public static double CelsiusToFahrenheit(string temperatureCelsius)
    {
        // Convert argument to double for calculations.
        double celsius = Double.Parse(temperatureCelsius);

        // Convert Celsius to Fahrenheit.
        double fahrenheit = (celsius * 9 / 5) + 32;

        return fahrenheit;
    }

    public static double FahrenheitToCelsius(string temperatureFahrenheit)
    {
        // Convert argument to double for calculations.
        double fahrenheit = Double.Parse(temperatureFahrenheit);

        // Convert Fahrenheit to Celsius.
        double celsius = (fahrenheit - 32) * 5 / 9;

        return celsius;
    }
}

class TestTemperatureConverter
{
    static void Main()
    {
        Console.WriteLine("Please select the convertor direction");
        Console.WriteLine("1. From Celsius to Fahrenheit.");
        Console.WriteLine("2. From Fahrenheit to Celsius.");
        Console.Write(":");

        string selection = Console.ReadLine();
        double F, C = 0;

        switch (selection)
        {
            case "1":
                Console.Write("Please enter the Celsius temperature: ");
                F = TemperatureConverter.CelsiusToFahrenheit(Console.ReadLine());
                Console.WriteLine("Temperature in Fahrenheit: {0:F2}", F);
                break;

            case "2":
                Console.Write("Please enter the Fahrenheit temperature: ");
                C = TemperatureConverter.FahrenheitToCelsius(Console.ReadLine());
                Console.WriteLine("Temperature in Celsius: {0:F2}", C);
                break;

            default:
```

```
                Console.WriteLine("Please select a convertor.");
                break;
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Example Output:
    Please select the convertor direction
    1. From Celsius to Fahrenheit.
    2. From Fahrenheit to Celsius.
    :2
    Please enter the Fahrenheit temperature: 20
    Temperature in Celsius: -6.67
    Press any key to exit.
 */
```

Static Members

A non-static class can contain static methods, fields, properties, or events. The static member is callable on a class even when no instance of the class has been created. The static member is always accessed by the class name, not the instance name. Only one copy of a static member exists, regardless of how many instances of the class are created. Static methods and properties cannot access non-static fields and events in their containing type, and they cannot access an instance variable of any object unless it is explicitly passed in a method parameter.

It is more typical to declare a non-static class with some static members, than to declare an entire class as static. Two common uses of static fields are to keep a count of the number of objects that have been instantiated, or to store a value that must be shared among all instances.

Static methods can be overloaded but not overridden, because they belong to the class, and not to any instance of the class.

Although a field cannot be declared as `static const`, a const field is essentially static in its behavior. It belongs to the type, not to instances of the type. Therefore, const fields can be accessed by using the same `ClassName.MemberName` notation that is used for static fields. No object instance is required.

C# does not support static local variables (variables that are declared in method scope).

You declare static class members by using the `static` keyword before the return type of the member, as shown in the following example:

```
public class Automobile
{
    public static int NumberOfWheels = 4;
    public static int SizeOfGasTank
    {
        get
        {
            return 15;
        }
    }
    public static void Drive() { }
    public static event EventType RunOutOfGas;

    // Other non-static fields and properties...
}
```

Static members are initialized before the static member is accessed for the first time and before the static constructor, if there is one, is called. To access a static class member, use the name of the class instead of a variable

name to specify the location of the member, as shown in the following example:

```
Automobile.Drive();
int i = Automobile.NumberOfWheels;
```

If your class contains static fields, provide a static constructor that initializes them when the class is loaded.

A call to a static method generates a call instruction in Microsoft intermediate language (MSIL), whereas a call to an instance method generates a `callvirt` instruction, which also checks for a null object references. However, most of the time the performance difference between the two is not significant.

C# Language Specification

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

See also

- C# Programming Guide
- static
- Classes
- class
- Static Constructors
- Instance Constructors

Access Modifiers (C# Programming Guide)

8/22/2019 • 4 minutes to read • Edit Online

All types and type members have an accessibility level, which controls whether they can be used from other code in your assembly or other assemblies. You can use the following access modifiers to specify the accessibility of a type or member when you declare it:

public
The type or member can be accessed by any other code in the same assembly or another assembly that references it.

private
The type or member can be accessed only by code in the same class or struct.

protected
The type or member can be accessed only by code in the same class, or in a class that is derived from that class.
internal
The type or member can be accessed by any code in the same assembly, but not from another assembly.

protected internal The type or member can be accessed by any code in the assembly in which it is declared, or from within a derived class in another assembly.

private protected The type or member can be accessed only within its declaring assembly, by code in the same class or in a type that is derived from that class.

The following examples demonstrate how to specify access modifiers on a type and member:

```
public class Bicycle
{
    public void Pedal() { }
}
```

Not all access modifiers can be used by all types or members in all contexts, and in some cases the accessibility of a type member is constrained by the accessibility of its containing type. The following sections provide more details about accessibility.

Class and Struct Accessibility

Classes and structs that are declared directly within a namespace (in other words, that are not nested within other classes or structs) can be either public or internal. Internal is the default if no access modifier is specified.

Struct members, including nested classes and structs, can be declared as public, internal, or private. Class members, including nested classes and structs, can be public, protected internal, protected, internal, private protected or private. The access level for class members and struct members, including nested classes and structs, is private by default. Private nested types are not accessible from outside the containing type.

Derived classes cannot have greater accessibility than their base types. In other words, you cannot have a public class `B` that derives from an internal class `A`. If this were allowed, it would have the effect of making `A` public, because all protected or internal members of `A` are accessible from the derived class.

You can enable specific other assemblies to access your internal types by using the InternalsVisibleToAttribute. For more information, see Friend Assemblies.

Class and Struct Member Accessibility

Class members (including nested classes and structs) can be declared with any of the six types of access. Struct members cannot be declared as protected because structs do not support inheritance.

Normally, the accessibility of a member is not greater than the accessibility of the type that contains it. However, a public member of an internal class might be accessible from outside the assembly if the member implements interface methods or overrides virtual methods that are defined in a public base class.

The type of any member that is a field, property, or event must be at least as accessible as the member itself. Similarly, the return type and the parameter types of any member that is a method, indexer, or delegate must be at least as accessible as the member itself. For example, you cannot have a public method `M` that returns a class `C` unless `C` is also public. Likewise, you cannot have a protected property of type `A` if `A` is declared as private.

User-defined operators must always be declared as public and static. For more information, see Operator overloading.

Finalizers cannot have accessibility modifiers.

To set the access level for a class or struct member, add the appropriate keyword to the member declaration, as shown in the following example.

```
// public class:
public class Tricycle
{
    // protected method:
    protected void Pedal() { }

    // private field:
    private int wheels = 3;

    // protected internal property:
    protected internal int Wheels
    {
        get { return wheels; }
    }
}
```

> **NOTE**
> The protected internal accessibility level means protected OR internal, not protected AND internal. In other words, a protected internal member can be accessed from any class in the same assembly, including derived classes. To limit accessibility to only derived classes in the same assembly, declare the class itself internal, and declare its members as protected. Also, starting with C# 7.2, you can use the private protected access modifier to achieve the same result without need to make the containing class internal.

Other Types

Interfaces declared directly within a namespace can be declared as public or internal and, just like classes and structs, interfaces default to internal access. Interface members are always public because the purpose of an interface is to enable other types to access a class or struct. No access modifiers can be applied to interface members.

Enumeration members are always public, and no access modifiers can be applied.

Delegates behave like classes and structs. By default, they have internal access when declared directly within a namespace, and private access when nested.

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
- Classes and Structs
- Interfaces
- private
- public
- internal
- protected
- protected internal
- private protected
- class
- struct
- interface

Fields (C# Programming Guide)

8/22/2019 • 3 minutes to read • Edit Online

A *field* is a variable of any type that is declared directly in a class or struct. Fields are *members* of their containing type.

A class or struct may have instance fields or static fields or both. Instance fields are specific to an instance of a type. If you have a class T, with an instance field F, you can create two objects of type T, and modify the value of F in each object without affecting the value in the other object. By contrast, a static field belongs to the class itself, and is shared among all instances of that class. Changes made from instance A will be visibly immediately to instances B and C if they access the field.

Generally, you should use fields only for variables that have private or protected accessibility. Data that your class exposes to client code should be provided through methods, properties and indexers. By using these constructs for indirect access to internal fields, you can guard against invalid input values. A private field that stores the data exposed by a public property is called a *backing store* or *backing field*.

Fields typically store the data that must be accessible to more than one class method and must be stored for longer than the lifetime of any single method. For example, a class that represents a calendar date might have three integer fields: one for the month, one for the day, and one for the year. Variables that are not used outside the scope of a single method should be declared as *local variables* within the method body itself.

Fields are declared in the class block by specifying the access level of the field, followed by the type of the field, followed by the name of the field. For example:

```
public class CalendarEntry
{
    // private field
    private DateTime date;

    // public field (Generally not recommended.)
    public string day;

    // Public property exposes date field safely.
    public DateTime Date
    {
        get
        {
            return date;
        }
        set
        {
            // Set some reasonable boundaries for likely birth dates.
            if (value.Year > 1900 && value.Year <= DateTime.Today.Year)
            {
                date = value;
            }
            else
                throw new ArgumentOutOfRangeException();
        }

    }

    // Public method also exposes date field safely.
    // Example call: birthday.SetDate("1975, 6, 30");
    public void SetDate(string dateString)
    {
        DateTime dt = Convert.ToDateTime(dateString);

        // Set some reasonable boundaries for likely birth dates.
        if (dt.Year > 1900 && dt.Year <= DateTime.Today.Year)
        {
            date = dt;
        }
        else
            throw new ArgumentOutOfRangeException();
    }

    public TimeSpan GetTimeSpan(string dateString)
    {
        DateTime dt = Convert.ToDateTime(dateString);

        if (dt != null && dt.Ticks < date.Ticks)
        {
            return date - dt;
        }
        else
            throw new ArgumentOutOfRangeException();

    }
}
```

To access a field in an object, add a period after the object name, followed by the name of the field, as in `objectname.fieldname`. For example:

```
CalendarEntry birthday = new CalendarEntry();
birthday.day = "Saturday";
```

A field can be given an initial value by using the assignment operator when the field is declared. To automatically

assign the `day` field to `"Monday"`, for example, you would declare `day` as in the following example:

```
public class CalendarDateWithInitialization
{
    public string day = "Monday";
    //...
}
```

Fields are initialized immediately before the constructor for the object instance is called. If the constructor assigns the value of a field, it will overwrite any value given during field declaration. For more information, see Using Constructors.

> **NOTE**
> A field initializer cannot refer to other instance fields.

Fields can be marked as public, private, protected, internal, protected internal or private protected. These access modifiers define how users of the class can access the fields. For more information, see Access Modifiers.

A field can optionally be declared static. This makes the field available to callers at any time, even if no instance of the class exists. For more information, see Static Classes and Static Class Members.

A field can be declared readonly. A read-only field can only be assigned a value during initialization or in a constructor. A `static readonly` field is very similar to a constant, except that the C# compiler does not have access to the value of a static read-only field at compile time, only at run time. For more information, see Constants.

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
- Classes and Structs
- Using Constructors
- Inheritance
- Access Modifiers
- Abstract and Sealed Classes and Class Members

Constants (C# Programming Guide)

8/22/2019 • 2 minutes to read • Edit Online

Constants are immutable values which are known at compile time and do not change for the life of the program. Constants are declared with the const modifier. Only the C# built-in types (excluding System.Object) may be declared as `const`. For a list of the built-in types, see Built-In Types Table. User-defined types, including classes, structs, and arrays, cannot be `const`. Use the readonly modifier to create a class, struct, or array that is initialized one time at runtime (for example in a constructor) and thereafter cannot be changed.

C# does not support `const` methods, properties, or events.

The enum type enables you to define named constants for integral built-in types (for example `int`, `uint`, `long`, and so on). For more information, see enum.

Constants must be initialized as they are declared. For example:

```
class Calendar1
{
    public const int Months = 12;
}
```

In this example, the constant `months` is always 12, and it cannot be changed even by the class itself. In fact, when the compiler encounters a constant identifier in C# source code (for example, `months`), it substitutes the literal value directly into the intermediate language (IL) code that it produces. Because there is no variable address associated with a constant at run time, `const` fields cannot be passed by reference and cannot appear as an l-value in an expression.

> **NOTE**
> Use caution when you refer to constant values defined in other code such as DLLs. If a new version of the DLL defines a new value for the constant, your program will still hold the old literal value until it is recompiled against the new version.

Multiple constants of the same type can be declared at the same time, for example:

```
class Calendar2
{
    public const int Months = 12, Weeks = 52, Days = 365;
}
```

The expression that is used to initialize a constant can refer to another constant if it does not create a circular reference. For example:

```
class Calendar3
{
    public const int Months = 12;
    public const int Weeks = 52;
    public const int Days = 365;

    public const double DaysPerWeek = (double) Days / (double) Weeks;
    public const double DaysPerMonth = (double) Days / (double) Months;
}
```

Constants can be marked as public, private, protected, internal, protected internal or private protected. These access modifiers define how users of the class can access the constant. For more information, see Access Modifiers.

Constants are accessed as if they were static fields because the value of the constant is the same for all instances of the type. You do not use the `static` keyword to declare them. Expressions that are not in the class that defines the constant must use the class name, a period, and the name of the constant to access the constant. For example:

```
int birthstones = Calendar.Months;
```

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
- Classes and Structs
- Properties
- Types
- readonly
- Immutability in C# Part One: Kinds of Immutability

How to: Define Abstract Properties (C# Programming Guide)

8/19/2019 • 2 minutes to read • Edit Online

The following example shows how to define abstract properties. An abstract property declaration does not provide an implementation of the property accessors -- it declares that the class supports properties, but leaves the accessor implementation to derived classes. The following example demonstrates how to implement the abstract properties inherited from a base class.

This sample consists of three files, each of which is compiled individually and its resulting assembly is referenced by the next compilation:

- abstractshape.cs: the `Shape` class that contains an abstract `Area` property.
- shapes.cs: The subclasses of the `Shape` class.
- shapetest.cs: A test program to display the areas of some `Shape`-derived objects.

To compile the example, use the following command:

```
csc abstractshape.cs shapes.cs shapetest.cs
```

This will create the executable file shapetest.exe.

Example

This file declares the `Shape` class that contains the `Area` property of the type `double`.

```
// compile with: csc -target:library abstractshape.cs
public abstract class Shape
{
    private string name;

    public Shape(string s)
    {
        // calling the set accessor of the Id property.
        Id = s;
    }

    public string Id
    {
        get
        {
            return name;
        }

        set
        {
            name = value;
        }
    }

    // Area is a read-only property - only a get accessor is needed:
    public abstract double Area
    {
        get;
    }

    public override string ToString()
    {
        return $"{Id} Area = {Area:F2}";
    }
}
```

- Modifiers on the property are placed on the property declaration itself. For example:

  ```
  public abstract double Area
  ```

- When declaring an abstract property (such as `Area` in this example), you simply indicate what property accessors are available, but do not implement them. In this example, only a get accessor is available, so the property is read-only.

Example

The following code shows three subclasses of `Shape` and how they override the `Area` property to provide their own implementation.

```
// compile with: csc -target:library -reference:abstractshape.dll shapes.cs
public class Square : Shape
{
    private int side;

    public Square(int side, string id)
        : base(id)
    {
        this.side = side;
    }

    public override double Area
    {
        get
        {
            // Given the side, return the area of a square:
            return side * side;
        }
    }
}

public class Circle : Shape
{
    private int radius;

    public Circle(int radius, string id)
        : base(id)
    {
        this.radius = radius;
    }

    public override double Area
    {
        get
        {
            // Given the radius, return the area of a circle:
            return radius * radius * System.Math.PI;
        }
    }
}

public class Rectangle : Shape
{
    private int width;
    private int height;

    public Rectangle(int width, int height, string id)
        : base(id)
    {
        this.width = width;
        this.height = height;
    }

    public override double Area
    {
        get
        {
            // Given the width and height, return the area of a rectangle:
            return width * height;
        }
    }
}
```

Example

The following code shows a test program that creates a number of `Shape`-derived objects and prints out their areas.

```
// compile with: csc -reference:abstractshape.dll;shapes.dll shapetest.cs
class TestClass
{
    static void Main()
    {
        Shape[] shapes =
        {
            new Square(5, "Square #1"),
            new Circle(3, "Circle #1"),
            new Rectangle( 4, 5, "Rectangle #1")
        };

        System.Console.WriteLine("Shapes Collection");
        foreach (Shape s in shapes)
        {
            System.Console.WriteLine(s);
        }
    }
}
/* Output:
    Shapes Collection
    Square #1 Area = 25.00
    Circle #1 Area = 28.27
    Rectangle #1 Area = 20.00
*/
```

See also

- C# Programming Guide
- Classes and Structs
- Abstract and Sealed Classes and Class Members
- Properties
- How to: Create and Use Assemblies Using the Command Line

How to: Define Constants in C#

8/22/2019 • 2 minutes to read • Edit Online

Constants are fields whose values are set at compile time and can never be changed. Use constants to provide meaningful names instead of numeric literals ("magic numbers") for special values.

> **NOTE**
> In C# the #define preprocessor directive cannot be used to define constants in the way that is typically used in C and C++.

To define constant values of integral types (`int` , `byte` , and so on) use an enumerated type. For more information, see enum.

To define non-integral constants, one approach is to group them in a single static class named `Constants` . This will require that all references to the constants be prefaced with the class name, as shown in the following example.

Example

```
static class Constants
{
    public const double Pi = 3.14159;
    public const int SpeedOfLight = 300000; // km per sec.
}
class Program
{
    static void Main()
    {
        double radius = 5.3;
        double area = Constants.Pi * (radius * radius);
        int secsFromSun = 149476000 / Constants.SpeedOfLight; // in km
    }
}
```

The use of the class name qualifier helps ensure that you and others who use the constant understand that it is constant and cannot be modified.

See also

- Classes and Structs

Properties (C# Programming Guide)

8/19/2019 • 4 minutes to read • Edit Online

A property is a member that provides a flexible mechanism to read, write, or compute the value of a private field. Properties can be used as if they are public data members, but they are actually special methods called *accessors*. This enables data to be accessed easily and still helps promote the safety and flexibility of methods.

Properties overview

- Properties enable a class to expose a public way of getting and setting values, while hiding implementation or verification code.
- A get property accessor is used to return the property value, and a set property accessor is used to assign a new value. These accessors can have different access levels. For more information, see Restricting Accessor Accessibility.
- The value keyword is used to define the value being assigned by the `set` accessor.
- Properties can be *read-write* (they have both a `get` and a `set` accessor), *read-only* (they have a `get` accessor but no `set` accessor), or *write-only* (they have a `set` accessor, but no `get` accessor). Write-only properties are rare and are most commonly used to restrict access to sensitive data.
- Simple properties that require no custom accessor code can be implemented either as expression body definitions or as auto-implemented properties.

Properties with backing fields

One basic pattern for implementing a property involves using a private backing field for setting and retrieving the property value. The `get` accessor returns the value of the private field, and the `set` accessor may perform some data validation before assigning a value to the private field. Both accessors may also perform some conversion or computation on the data before it is stored or returned.

The following example illustrates this pattern. In this example, the `TimePeriod` class represents an interval of time. Internally, the class stores the time interval in seconds in a private field named `_seconds`. A read-write property named `Hours` allows the customer to specify the time interval in hours. Both the `get` and the `set` accessors perform the necessary conversion between hours and seconds. In addition, the `set` accessor validates the data and throws an ArgumentOutOfRangeException if the number of hours is invalid.

```
using System;

class TimePeriod
{
   private double _seconds;

   public double Hours
   {
       get { return _seconds / 3600; }
       set {
          if (value < 0 || value > 24)
             throw new ArgumentOutOfRangeException(
                   $"{nameof(value)} must be between 0 and 24.");

          _seconds = value * 3600;
       }
   }
}

class Program
{
   static void Main()
   {
       TimePeriod t = new TimePeriod();
       // The property assignment causes the 'set' accessor to be called.
       t.Hours = 24;

       // Retrieving the property causes the 'get' accessor to be called.
       Console.WriteLine($"Time in hours: {t.Hours}");
   }
}
// The example displays the following output:
//    Time in hours: 24
```

Expression body definitions

Property accessors often consist of single-line statements that just assign or return the result of an expression. You can implement these properties as expression-bodied members. Expression body definitions consist of the `=>` symbol followed by the expression to assign to or retrieve from the property.

Starting with C# 6, read-only properties can implement the `get` accessor as an expression-bodied member. In this case, neither the `get` accessor keyword nor the `return` keyword is used. The following example implements the read-only `Name` property as an expression-bodied member.

```
using System;

public class Person
{
   private string _firstName;
   private string _lastName;

   public Person(string first, string last)
   {
      _firstName = first;
      _lastName = last;
   }

   public string Name => $"{_firstName} {_lastName}";
}

public class Example
{
   public static void Main()
   {
      var person = new Person("Isabelle", "Butts");
      Console.WriteLine(person.Name);
   }
}
// The example displays the following output:
//      Isabelle Butts
```

Starting with C# 7.0, both the `get` and the `set` accessor can be implemented as expression-bodied members. In this case, the `get` and `set` keywords must be present. The following example illustrates the use of expression body definitions for both accessors. Note that the `return` keyword is not used with the `get` accessor.

```
using System;

public class SaleItem
{
   string _name;
   decimal _cost;

   public SaleItem(string name, decimal cost)
   {
      _name = name;
      _cost = cost;
   }

   public string Name
   {
      get => _name;
      set => _name = value;
   }

   public decimal Price
   {
      get => _cost;
      set => _cost = value;
   }
}

class Program
{
   static void Main(string[] args)
   {
      var item = new SaleItem("Shoes", 19.95m);
      Console.WriteLine($"{item.Name}: sells for {item.Price:C2}");
   }
}
// The example displays output like the following:
//       Shoes: sells for $19.95
```

Auto-implemented properties

In some cases, property `get` and `set` accessors just assign a value to or retrieve a value from a backing field without including any additional logic. By using auto-implemented properties, you can simplify your code while having the C# compiler transparently provide the backing field for you.

If a property has both a `get` and a `set` accessor, both must be auto-implemented. You define an auto-implemented property by using the `get` and `set` keywords without providing any implementation. The following example repeats the previous one, except that `Name` and `Price` are auto-implemented properties. Note that the example also removes the parameterized constructor, so that `SaleItem` objects are now initialized with a call to the parameterless constructor and an object initializer.

```
using System;

public class SaleItem
{
   public string Name
   { get; set; }

   public decimal Price
   { get; set; }
}

class Program
{
   static void Main(string[] args)
   {
      var item = new SaleItem{ Name = "Shoes", Price = 19.95m };
      Console.WriteLine($"{item.Name}: sells for {item.Price:C2}");
   }
}
// The example displays output like the following:
//       Shoes: sells for $19.95
```

Related sections

- Using Properties
- Interface Properties
- Comparison Between Properties and Indexers
- Restricting Accessor Accessibility
- Auto-Implemented Properties

C# Language Specification

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

See also

- C# Programming Guide
- Using Properties
- Indexers
- get keyword
- set keyword

Using Properties (C# Programming Guide)

8/22/2019 • 8 minutes to read • Edit Online

Properties combine aspects of both fields and methods. To the user of an object, a property appears to be a field, accessing the property requires the same syntax. To the implementer of a class, a property is one or two code blocks, representing a get accessor and/or a set accessor. The code block for the `get` accessor is executed when the property is read; the code block for the `set` accessor is executed when the property is assigned a new value. A property without a `set` accessor is considered read-only. A property without a `get` accessor is considered write-only. A property that has both accessors is read-write.

Unlike fields, properties are not classified as variables. Therefore, you cannot pass a property as a ref or out parameter.

Properties have many uses: they can validate data before allowing a change; they can transparently expose data on a class where that data is actually retrieved from some other source, such as a database; they can take an action when data is changed, such as raising an event, or changing the value of other fields.

Properties are declared in the class block by specifying the access level of the field, followed by the type of the property, followed by the name of the property, and followed by a code block that declares a `get`-accessor and/or a `set` accessor. For example:

```
public class Date
{
    private int month = 7;  // Backing store

    public int Month
    {
        get
        {
            return month;
        }
        set
        {
            if ((value > 0) && (value < 13))
            {
                month = value;
            }
        }
    }
}
```

In this example, `Month` is declared as a property so that the `set` accessor can make sure that the `Month` value is set between 1 and 12. The `Month` property uses a private field to track the actual value. The real location of a property's data is often referred to as the property's "backing store." It is common for properties to use private fields as a backing store. The field is marked private in order to make sure that it can only be changed by calling the property. For more information about public and private access restrictions, see Access Modifiers.

Auto-implemented properties provide simplified syntax for simple property declarations. For more information, see Auto-Implemented Properties.

The get Accessor

The body of the `get` accessor resembles that of a method. It must return a value of the property type. The execution of the `get` accessor is equivalent to reading the value of the field. For example, when you are returning

the private variable from the `get` accessor and optimizations are enabled, the call to the `get` accessor method is inlined by the compiler so there is no method-call overhead. However, a virtual `get` accessor method cannot be inlined because the compiler does not know at compile-time which method may actually be called at run time. The following is a `get` accessor that returns the value of a private field `name`:

```
class Person
{
    private string name;  // the name field
    public string Name    // the Name property
    {
        get
        {
            return name;
        }
    }
}
```

When you reference the property, except as the target of an assignment, the `get` accessor is invoked to read the value of the property. For example:

```
Person person = new Person();
//...

System.Console.Write(person.Name);  // the get accessor is invoked here
```

The `get` accessor must end in a return or throw statement, and control cannot flow off the accessor body.

It is a bad programming style to change the state of the object by using the `get` accessor. For example, the following accessor produces the side effect of changing the state of the object every time that the `number` field is accessed.

```
private int number;
public int Number
{
    get
    {
        return number++;   // Don't do this
    }
}
```

The `get` accessor can be used to return the field value or to compute it and return it. For example:

```
class Employee
{
    private string name;
    public string Name
    {
        get
        {
            return name != null ? name : "NA";
        }
    }
}
```

In the previous code segment, if you do not assign a value to the `Name` property, it will return the value NA.

The set Accessor

The `set` accessor resembles a method whose return type is void. It uses an implicit parameter called `value`, whose type is the type of the property. In the following example, a `set` accessor is added to the `Name` property:

```
class Person
{
    private string name;  // the name field
    public string Name    // the Name property
    {
        get
        {
            return name;
        }
        set
        {
            name = value;
        }
    }
}
```

When you assign a value to the property, the `set` accessor is invoked by using an argument that provides the new value. For example:

```
Person person = new Person();
person.Name = "Joe";  // the set accessor is invoked here

System.Console.Write(person.Name);  // the get accessor is invoked here
```

It is an error to use the implicit parameter name, `value`, for a local variable declaration in a `set` accessor.

Remarks

Properties can be marked as `public`, `private`, `protected`, `internal`, `protected internal` or `private protected`. These access modifiers define how users of the class can access the property. The `get` and `set` accessors for the same property may have different access modifiers. For example, the `get` may be `public` to allow read-only access from outside the type, and the `set` may be `private` or `protected`. For more information, see Access Modifiers.

A property may be declared as a static property by using the `static` keyword. This makes the property available to callers at any time, even if no instance of the class exists. For more information, see Static Classes and Static Class Members.

A property may be marked as a virtual property by using the virtual keyword. This enables derived classes to override the property behavior by using the override keyword. For more information about these options, see Inheritance.

A property overriding a virtual property can also be sealed, specifying that for derived classes it is no longer virtual. Lastly, a property can be declared abstract. This means that there is no implementation in the class, and derived classes must write their own implementation. For more information about these options, see Abstract and Sealed Classes and Class Members.

> **NOTE**
> It is an error to use a virtual, abstract, or override modifier on an accessor of a static property.

Example

This example demonstrates instance, static, and read-only properties. It accepts the name of the employee from the keyboard, increments `NumberOfEmployees` by 1, and displays the Employee name and number.

```
public class Employee
{
    public static int NumberOfEmployees;
    private static int counter;
    private string name;

    // A read-write instance property:
    public string Name
    {
        get { return name; }
        set { name = value; }
    }

    // A read-only static property:
    public static int Counter
    {
        get { return counter; }
    }

    // A Constructor:
    public Employee()
    {
        // Calculate the employee's number:
        counter = ++NumberOfEmployees;
    }
}

class TestEmployee
{
    static void Main()
    {
        Employee.NumberOfEmployees = 107;
        Employee e1 = new Employee();
        e1.Name = "Claude Vige";

        System.Console.WriteLine("Employee number: {0}", Employee.Counter);
        System.Console.WriteLine("Employee name: {0}", e1.Name);
    }
}
/* Output:
    Employee number: 108
    Employee name: Claude Vige
*/
```

Example

This example demonstrates how to access a property in a base class that is hidden by another property that has the same name in a derived class.

```
public class Employee
{
    private string name;
    public string Name
    {
        get { return name; }
        set { name = value; }
    }
}

public class Manager : Employee
{
    private string name;

    // Notice the use of the new modifier:
    public new string Name
    {
        get { return name; }
        set { name = value + ", Manager"; }
    }
}

class TestHiding
{
    static void Main()
    {
        Manager m1 = new Manager();

        // Derived class property.
        m1.Name = "John";

        // Base class property.
        ((Employee)m1).Name = "Mary";

        System.Console.WriteLine("Name in the derived class is: {0}", m1.Name);
        System.Console.WriteLine("Name in the base class is: {0}", ((Employee)m1).Name);
    }
}
/* Output:
    Name in the derived class is: John, Manager
    Name in the base class is: Mary
*/
```

The following are important points in the previous example:

- The property `Name` in the derived class hides the property `Name` in the base class. In such a case, the `new` modifier is used in the declaration of the property in the derived class:

  ```
  public new string Name
  ```

- The cast `(Employee)` is used to access the hidden property in the base class:

  ```
  ((Employee)m1).Name = "Mary";
  ```

  For more information about hiding members, see the new Modifier.

Example

In this example, two classes, `Cube` and `Square`, implement an abstract class, `Shape`, and override its abstract `Area` property. Note the use of the override modifier on the properties. The program accepts the side as an input and calculates the areas for the square and cube. It also accepts the area as an input and calculates the

corresponding side for the square and cube.

```
abstract class Shape
{
    public abstract double Area
    {
        get;
        set;
    }
}

class Square : Shape
{
    public double side;

    public Square(double s)  //constructor
    {
        side = s;
    }

    public override double Area
    {
        get
        {
            return side * side;
        }
        set
        {
            side = System.Math.Sqrt(value);
        }
    }
}

class Cube : Shape
{
    public double side;

    public Cube(double s)
    {
        side = s;
    }

    public override double Area
    {
        get
        {
            return 6 * side * side;
        }
        set
        {
            side = System.Math.Sqrt(value / 6);
        }
    }
}

class TestShapes
{
    static void Main()
    {
        // Input the side:
        System.Console.Write("Enter the side: ");
        double side = double.Parse(System.Console.ReadLine());

        // Compute the areas:
        Square s = new Square(side);
        Cube c = new Cube(side);

        // Display the results:
```

```
        System.Console.WriteLine("Area of the square = {0:F2}", s.Area);
        System.Console.WriteLine("Area of the cube = {0:F2}", c.Area);
        System.Console.WriteLine();

        // Input the area:
        System.Console.Write("Enter the area: ");
        double area = double.Parse(System.Console.ReadLine());

        // Compute the sides:
        s.Area = area;
        c.Area = area;

        // Display the results:
        System.Console.WriteLine("Side of the square = {0:F2}", s.side);
        System.Console.WriteLine("Side of the cube = {0:F2}", c.side);
    }
}
/* Example Output:
    Enter the side: 4
    Area of the square = 16.00
    Area of the cube = 96.00

    Enter the area: 24
    Side of the square = 4.90
    Side of the cube = 2.00
*/
```

See also

- C# Programming Guide
- Properties
- Interface Properties
- Auto-Implemented Properties

Interface Properties (C# Programming Guide)

8/19/2019 • 2 minutes to read • Edit Online

Properties can be declared on an interface. The following is an example of an interface property accessor:

```
public interface ISampleInterface
{
    // Property declaration:
    string Name
    {
        get;
        set;
    }
}
```

The accessor of an interface property does not have a body. Thus, the purpose of the accessors is to indicate whether the property is read-write, read-only, or write-only.

Example

In this example, the interface `IEmployee` has a read-write property, `Name`, and a read-only property, `Counter`. The class `Employee` implements the `IEmployee` interface and uses these two properties. The program reads the name of a new employee and the current number of employees and displays the employee name and the computed employee number.

You could use the fully qualified name of the property, which references the interface in which the member is declared. For example:

```
string IEmployee.Name
{
    get { return "Employee Name"; }
    set { }
}
```

This is called Explicit Interface Implementation. For example, if the class `Employee` is implementing two interfaces `ICitizen` and `IEmployee` and both interfaces have the `Name` property, the explicit interface member implementation will be necessary. That is, the following property declaration:

```
string IEmployee.Name
{
    get { return "Employee Name"; }
    set { }
}
```

implements the `Name` property on the `IEmployee` interface, while the following declaration:

```
string ICitizen.Name
{
    get { return "Citizen Name"; }
    set { }
}
```

implements the `Name` property on the `ICitizen` interface.

```
interface IEmployee
{
    string Name
    {
        get;
        set;
    }

    int Counter
    {
        get;
    }
}

public class Employee : IEmployee
{
    public static int numberOfEmployees;

    private string name;
    public string Name  // read-write instance property
    {
        get
        {
            return name;
        }
        set
        {
            name = value;
        }
    }

    private int counter;
    public int Counter  // read-only instance property
    {
        get
        {
            return counter;
        }
    }

    public Employee()  // constructor
    {
        counter = ++numberOfEmployees;
    }
}

class TestEmployee
{
    static void Main()
    {
        System.Console.Write("Enter number of employees: ");
        Employee.numberOfEmployees = int.Parse(System.Console.ReadLine());

        Employee e1 = new Employee();
        System.Console.Write("Enter the name of the new employee: ");
        e1.Name = System.Console.ReadLine();

        System.Console.WriteLine("The employee information:");
        System.Console.WriteLine("Employee number: {0}", e1.Counter);
        System.Console.WriteLine("Employee name: {0}", e1.Name);
    }
}
```

Sample Output

```
Enter number of employees: 210
```

```
Enter the name of the new employee: Hazem Abolrous
```

```
The employee information:
```

```
Employee number: 211
```

```
Employee name: Hazem Abolrous
```

See also

- C# Programming Guide
- Properties
- Using Properties
- Comparison Between Properties and Indexers
- Indexers
- Interfaces

Restricting Accessor Accessibility (C# Programming Guide)

8/19/2019 • 4 minutes to read • Edit Online

The get and set portions of a property or indexer are called *accessors*. By default these accessors have the same visibility or access level of the property or indexer to which they belong. For more information, see accessibility levels. However, it is sometimes useful to restrict access to one of these accessors. Typically, this involves restricting the accessibility of the `set` accessor, while keeping the `get` accessor publicly accessible. For example:

```
private string name = "Hello";

public string Name
{
    get
    {
        return name;
    }
    protected set
    {
        name = value;
    }
}
```

In this example, a property called `Name` defines a `get` and `set` accessor. The `get` accessor receives the accessibility level of the property itself, `public` in this case, while the `set` accessor is explicitly restricted by applying the protected access modifier to the accessor itself.

Restrictions on Access Modifiers on Accessors

Using the accessor modifiers on properties or indexers is subject to these conditions:

- You cannot use accessor modifiers on an interface or an explicit interface member implementation.
- You can use accessor modifiers only if the property or indexer has both `set` and `get` accessors. In this case, the modifier is permitted on only one of the two accessors.
- If the property or indexer has an override modifier, the accessor modifier must match the accessor of the overridden accessor, if any.
- The accessibility level on the accessor must be more restrictive than the accessibility level on the property or indexer itself.

Access Modifiers on Overriding Accessors

When you override a property or indexer, the overridden accessors must be accessible to the overriding code. Also, the accessibility of both the property/indexer and its accessors must match the corresponding overridden property/indexer and its accessors. For example:

```
public class Parent
{
    public virtual int TestProperty
    {
        // Notice the accessor accessibility level.
        protected set { }

        // No access modifier is used here.
        get { return 0; }
    }
}
public class Kid : Parent
{
    public override int TestProperty
    {
        // Use the same accessibility level as in the overridden accessor.
        protected set { }

        // Cannot use access modifier here.
        get { return 0; }
    }
}
```

Implementing Interfaces

When you use an accessor to implement an interface, the accessor may not have an access modifier. However, if you implement the interface using one accessor, such as `get`, the other accessor can have an access modifier, as in the following example:

```
public interface ISomeInterface
{
    int TestProperty
    {
        // No access modifier allowed here
        // because this is an interface.
        get;
    }
}

public class TestClass : ISomeInterface
{
    public int TestProperty
    {
        // Cannot use access modifier here because
        // this is an interface implementation.
        get { return 10; }

        // Interface property does not have set accessor,
        // so access modifier is allowed.
        protected set { }
    }
}
```

Accessor Accessibility Domain

If you use an access modifier on the accessor, the accessibility domain of the accessor is determined by this modifier.

If you did not use an access modifier on the accessor, the accessibility domain of the accessor is determined by the accessibility level of the property or indexer.

Example

The following example contains three classes, `BaseClass`, `DerivedClass`, and `MainClass`. There are two properties on the `BaseClass`, `Name` and `Id` on both classes. The example demonstrates how the property `Id` on `DerivedClass` can be hidden by the property `Id` on `BaseClass` when you use a restrictive access modifier such as protected or private. Therefore, when you assign values to this property, the property on the `BaseClass` class is called instead. Replacing the access modifier by public will make the property accessible.

The example also demonstrates that a restrictive access modifier, such as `private` or `protected`, on the `set` accessor of the `Name` property in `DerivedClass` prevents access to the accessor and generates an error when you assign to it.

```
public class BaseClass
{
    private string name = "Name-BaseClass";
    private string id = "ID-BaseClass";

    public string Name
    {
        get { return name; }
        set { }
    }

    public string Id
    {
        get { return id; }
        set { }
    }
}

public class DerivedClass : BaseClass
{
    private string name = "Name-DerivedClass";
    private string id = "ID-DerivedClass";

    new public string Name
    {
        get
        {
            return name;
        }

        // Using "protected" would make the set accessor not accessible.
        set
        {
            name = value;
        }
    }

    // Using private on the following property hides it in the Main Class.
    // Any assignment to the property will use Id in BaseClass.
    new private string Id
    {
        get
        {
            return id;
        }
        set
        {
            id = value;
        }
    }
}

class MainClass
```

```
{
    static void Main()
    {
        BaseClass b1 = new BaseClass();
        DerivedClass d1 = new DerivedClass();

        b1.Name = "Mary";
        d1.Name = "John";

        b1.Id = "Mary123";
        d1.Id = "John123";  // The BaseClass.Id property is called.

        System.Console.WriteLine("Base: {0}, {1}", b1.Name, b1.Id);
        System.Console.WriteLine("Derived: {0}, {1}", d1.Name, d1.Id);

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
/* Output:
    Base: Name-BaseClass, ID-BaseClass
    Derived: John, ID-BaseClass
*/
```

Comments

Notice that if you replace the declaration `new private string Id` by `new public string Id`, you get the output:

```
Name and ID in the base class: Name-BaseClass, ID-BaseClass
```

```
Name and ID in the derived class: John, John123
```

See also

- C# Programming Guide
- Properties
- Indexers
- Access Modifiers

How to: Declare and Use Read Write Properties (C# Programming Guide)

8/19/2019 • 2 minutes to read • Edit Online

Properties provide the convenience of public data members without the risks that come with unprotected, uncontrolled, and unverified access to an object's data. This is accomplished through *accessors*: special methods that assign and retrieve values from the underlying data member. The set accessor enables data members to be assigned, and the get accessor retrieves data member values.

This sample shows a `Person` class that has two properties: `Name` (string) and `Age` (int). Both properties provide `get` and `set` accessors, so they are considered read/write properties.

Example

```
class Person
{
    private string name = "N/A";
    private int age = 0;

    // Declare a Name property of type string:
    public string Name
    {
        get
        {
            return name;
        }
        set
        {
            name = value;
        }
    }

    // Declare an Age property of type int:
    public int Age
    {
        get
        {
            return age;
        }

        set
        {
            age = value;
        }
    }

    public override string ToString()
    {
        return "Name = " + Name + ", Age = " + Age;
    }
}

class TestPerson
{
    static void Main()
    {
        // Create a new Person object:
        Person person = new Person();
```

```
        // Print out the name and the age associated with the person:
        Console.WriteLine("Person details - {0}", person);

        // Set some values on the person object:
        person.Name = "Joe";
        person.Age = 99;
        Console.WriteLine("Person details - {0}", person);

        // Increment the Age property:
        person.Age += 1;
        Console.WriteLine("Person details - {0}", person);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
    Person details - Name = N/A, Age = 0
    Person details - Name = Joe, Age = 99
    Person details - Name = Joe, Age = 100
*/
```

Robust Programming

In the previous example, the `Name` and `Age` properties are public and include both a `get` and a `set` accessor. This allows any object to read and write these properties. It is sometimes desirable, however, to exclude one of the accessors. Omitting the `set` accessor, for example, makes the property read-only:

```
public string Name
{
    get
    {
        return name;
    }
}
```

Alternatively, you can expose one accessor publicly but make the other private or protected. For more information, see Asymmetric Accessor Accessibility.

Once the properties are declared, they can be used as if they were fields of the class. This allows for a very natural syntax when both getting and setting the value of a property, as in the following statements:

```
person.Name = "Joe";
person.Age = 99;
```

Note that in a property `set` method a special `value` variable is available. This variable contains the value that the user specified, for example:

```
name = value;
```

Notice the clean syntax for incrementing the `Age` property on a `Person` object:

```
person.Age += 1;
```

If separate `set` and `get` methods were used to model properties, the equivalent code might look like this:

```
person.SetAge(person.GetAge() + 1);
```

The `ToString` method is overridden in this example:

```
public override string ToString()
{
    return "Name = " + Name + ", Age = " + Age;
}
```

Notice that `ToString` is not explicitly used in the program. It is invoked by default by the `WriteLine` calls.

See also

- C# Programming Guide
- Properties
- Classes and Structs

Auto-Implemented Properties (C# Programming Guide)

8/19/2019 • 2 minutes to read • Edit Online

In C# 3.0 and later, auto-implemented properties make property-declaration more concise when no additional logic is required in the property accessors. They also enable client code to create objects. When you declare a property as shown in the following example, the compiler creates a private, anonymous backing field that can only be accessed through the property's `get` and `set` accessors.

Example

The following example shows a simple class that has some auto-implemented properties:

```
// This class is mutable. Its data can be modified from
// outside the class.
class Customer
{
    // Auto-implemented properties for trivial get and set
    public double TotalPurchases { get; set; }
    public string Name { get; set; }
    public int CustomerID { get; set; }

    // Constructor
    public Customer(double purchases, string name, int ID)
    {
        TotalPurchases = purchases;
        Name = name;
        CustomerID = ID;
    }

    // Methods
    public string GetContactInfo() { return "ContactInfo"; }
    public string GetTransactionHistory() { return "History"; }

    // .. Additional methods, events, etc.
}

class Program
{
    static void Main()
    {
        // Intialize a new object.
        Customer cust1 = new Customer(4987.63, "Northwind", 90108);

        // Modify a property.
        cust1.TotalPurchases += 499.99;
    }
}
```

In C# 6 and later, you can initialize auto-implemented properties similarly to fields:

```
public string FirstName { get; set; } = "Jane";
```

The class that is shown in the previous example is mutable. Client code can change the values in objects after they are created. In complex classes that contain significant behavior (methods) as well as data, it is often necessary to

have public properties. However, for small classes or structs that just encapsulate a set of values (data) and have little or no behaviors, you should either make the objects immutable by declaring the set accessor as private (immutable to consumers) or by declaring only a get accessor (immutable everywhere except the constructor). For more information, see How to: Implement a Lightweight Class with Auto-Implemented Properties.

See also

- Properties
- Modifiers

How to: Implement a Lightweight Class with Auto-Implemented Properties (C# Programming Guide)

8/19/2019 • 2 minutes to read • Edit Online

This example shows how to create an immutable lightweight class that serves only to encapsulate a set of auto-implemented properties. Use this kind of construct instead of a struct when you must use reference type semantics.

You can make an immutable property in two ways:

- You can declare the set accessor to be private. The property is only settable within the type, but it is immutable to consumers.

  When you declare a private `set` accessor, you cannot use an object initializer to initialize the property. You must use a constructor or a factory method.

- You can declare only the get accessor, which makes the property immutable everywhere except in the type's constructor.

Example

The following example shows two ways to implement an immutable class that has auto-implemented properties. Each way declares one of the properties with a private `set` and one of the properties with a `get` only. The first class uses a constructor only to initialize the properties, and the second class uses a static factory method that calls a constructor.

```
// This class is immutable. After an object is created,
// it cannot be modified from outside the class. It uses a
// constructor to initialize its properties.
class Contact
{
    // Read-only properties.
    public string Name { get; }
    public string Address { get; private set; }

    // Public constructor.
    public Contact(string contactName, string contactAddress)
    {
        Name = contactName;
        Address = contactAddress;
    }
}

// This class is immutable. After an object is created,
// it cannot be modified from outside the class. It uses a
// static method and private constructor to initialize its properties.
public class Contact2
{
    // Read-only properties.
    public string Name { get; private set; }
    public string Address { get; }

    // Private constructor.
    private Contact2(string contactName, string contactAddress)
    {
        Name = contactName;
        Address = contactAddress;
```

```
    }

    // Public factory method.
    public static Contact2 CreateContact(string name, string address)
    {
        return new Contact2(name, address);
    }
}

public class Program
{
    static void Main()
    {
        // Some simple data sources.
        string[] names = {"Terry Adams","Fadi Fakhouri", "Hanying Feng",
                          "Cesar Garcia", "Debra Garcia"};
        string[] addresses = {"123 Main St.", "345 Cypress Ave.", "678 1st Ave",
                              "12 108th St.", "89 E. 42nd St."};

        // Simple query to demonstrate object creation in select clause.
        // Create Contact objects by using a constructor.
        var query1 = from i in Enumerable.Range(0, 5)
                     select new Contact(names[i], addresses[i]);

        // List elements cannot be modified by client code.
        var list = query1.ToList();
        foreach (var contact in list)
        {
            Console.WriteLine("{0}, {1}", contact.Name, contact.Address);
        }

        // Create Contact2 objects by using a static factory method.
        var query2 = from i in Enumerable.Range(0, 5)
                     select Contact2.CreateContact(names[i], addresses[i]);

        // Console output is identical to query1.
        var list2 = query2.ToList();

        // List elements cannot be modified by client code.
        // CS0272:
        // list2[0].Name = "Eugene Zabokritski";

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

/* Output:
    Terry Adams, 123 Main St.
    Fadi Fakhouri, 345 Cypress Ave.
    Hanying Feng, 678 1st Ave
    Cesar Garcia, 12 108th St.
    Debra Garcia, 89 E. 42nd St.
*/
```

The compiler creates backing fields for each auto-implemented property. The fields are not accessible directly from source code.

See also

- Properties
- struct
- Object and Collection Initializers

Methods (C# Programming Guide)

8/30/2019 • 10 minutes to read • Edit Online

A method is a code block that contains a series of statements. A program causes the statements to be executed by calling the method and specifying any required method arguments. In C#, every executed instruction is performed in the context of a method. The Main method is the entry point for every C# application and it is called by the common language runtime (CLR) when the program is started.

> **NOTE**
> This topic discusses named methods. For information about anonymous functions, see Anonymous Functions.

Method Signatures

Methods are declared in a class or struct by specifying the access level such as `public` or `private`, optional modifiers such as `abstract` or `sealed`, the return value, the name of the method, and any method parameters. These parts together are the signature of the method.

> **NOTE**
> A return type of a method is not part of the signature of the method for the purposes of method overloading. However, it is part of the signature of the method when determining the compatibility between a delegate and the method that it points to.

Method parameters are enclosed in parentheses and are separated by commas. Empty parentheses indicate that the method requires no parameters. This class contains four methods:

```
abstract class Motorcycle
{
    // Anyone can call this.
    public void StartEngine() {/* Method statements here */ }

    // Only derived classes can call this.
    protected void AddGas(int gallons) { /* Method statements here */ }

    // Derived classes can override the base class implementation.
    public virtual int Drive(int miles, int speed) { /* Method statements here */ return 1; }

    // Derived classes must implement this.
    public abstract double GetTopSpeed();
}
```

Method Access

Calling a method on an object is like accessing a field. After the object name, add a period, the name of the method, and parentheses. Arguments are listed within the parentheses, and are separated by commas. The methods of the `Motorcycle` class can therefore be called as in the following example:

```
class TestMotorcycle : Motorcycle
{

    public override double GetTopSpeed()
    {
        return 108.4;
    }

    static void Main()
    {

        TestMotorcycle moto = new TestMotorcycle();

        moto.StartEngine();
        moto.AddGas(15);
        moto.Drive(5, 20);
        double speed = moto.GetTopSpeed();
        Console.WriteLine("My top speed is {0}", speed);
    }
}
```

Method Parameters vs. Arguments

The method definition specifies the names and types of any parameters that are required. When calling code calls the method, it provides concrete values called arguments for each parameter. The arguments must be compatible with the parameter type but the argument name (if any) used in the calling code does not have to be the same as the parameter named defined in the method. For example:

```
public void Caller()
{
    int numA = 4;
    // Call with an int variable.
    int productA = Square(numA);

    int numB = 32;
    // Call with another int variable.
    int productB = Square(numB);

    // Call with an integer literal.
    int productC = Square(12);

    // Call with an expression that evaulates to int.
    productC = Square(productA * 3);
}

int Square(int i)
{
    // Store input argument in a local variable.
    int input = i;
    return input * input;
}
```

Passing by Reference vs. Passing by Value

By default, when a value type is passed to a method, a copy is passed instead of the object itself. Therefore, changes to the argument have no effect on the original copy in the calling method. You can pass a value-type by reference by using the ref keyword. For more information, see Passing Value-Type Parameters. For a list of built-in value types, see Value Types Table.

When an object of a reference type is passed to a method, a reference to the object is passed. That is, the method

receives not the object itself but an argument that indicates the location of the object. If you change a member of the object by using this reference, the change is reflected in the argument in the calling method, even if you pass the object by value.

You create a reference type by using the `class` keyword, as the following example shows.

```
public class SampleRefType
{
    public int value;
}
```

Now, if you pass an object that is based on this type to a method, a reference to the object is passed. The following example passes an object of type `SampleRefType` to method `ModifyObject`.

```
public static void TestRefType()
{
    SampleRefType rt = new SampleRefType();
    rt.value = 44;
    ModifyObject(rt);
    Console.WriteLine(rt.value);
}
static void ModifyObject(SampleRefType obj)
{
    obj.value = 33;
}
```

The example does essentially the same thing as the previous example in that it passes an argument by value to a method. But, because a reference type is used, the result is different. The modification that is made in `ModifyObject` to the `value` field of the parameter, `obj`, also changes the `value` field of the argument, `rt`, in the `TestRefType` method. The `TestRefType` method displays 33 as the output.

For more information about how to pass reference types by reference and by value, see Passing Reference-Type Parameters and Reference Types.

Return Values

Methods can return a value to the caller. If the return type, the type listed before the method name, is not `void`, the method can return the value by using the `return` keyword. A statement with the `return` keyword followed by a value that matches the return type will return that value to the method caller.

The value can be returned to the caller by value or, starting with C# 7.0, by reference. Values are returned to the caller by reference if the `ref` keyword is used in the method signature and it follows each `return` keyword. For example, the following method signature and return statement indicate that the method returns a variable names `estDistance` by reference to the caller.

```
public ref double GetEstimatedDistance()
{
   return ref estDistance;
}
```

The `return` keyword also stops the execution of the method. If the return type is `void`, a `return` statement without a value is still useful to stop the execution of the method. Without the `return` keyword, the method will stop executing when it reaches the end of the code block. Methods with a non-void return type are required to use the `return` keyword to return a value. For example, these two methods use the `return` keyword to return integers:

```
class SimpleMath
{
    public int AddTwoNumbers(int number1, int number2)
    {
        return number1 + number2;
    }

    public int SquareANumber(int number)
    {
        return number * number;
    }
}
```

To use a value returned from a method, the calling method can use the method call itself anywhere a value of the same type would be sufficient. You can also assign the return value to a variable. For example, the following two code examples accomplish the same goal:

```
int result = obj.AddTwoNumbers(1, 2);
result = obj.SquareANumber(result);
// The result is 9.
Console.WriteLine(result);
```

```
result = obj.SquareANumber(obj.AddTwoNumbers(1, 2));
// The result is 9.
Console.WriteLine(result);
```

Using a local variable, in this case, `result`, to store a value is optional. It may help the readability of the code, or it may be necessary if you need to store the original value of the argument for the entire scope of the method.

To use a value returned by reference from a method, you must declare a ref local variable if you intend to modify its value. For example, if the `Planet.GetEstimatedDistance` method returns a Double value by reference, you can define it as a ref local variable with code like the following:

```
ref int distance = plant
```

Returning a multi-dimensional array from a method, `M`, that modifies the array's contents is not necessary if the calling function passed the array into `M`. You may return the resulting array from `M` for good style or functional flow of values, but it is not necessary because C# passes all reference types by value, and the value of an array reference is the pointer to the array. In the method `M`, any changes to the array's contents are observable by any code that has a reference to the array, as shown in the following example.

```
static void Main(string[] args)
{
    int[,] matrix = new int[2, 2];
    FillMatrix(matrix);
    // matrix is now full of -1
}

public static void FillMatrix(int[,] matrix)
{
    for (int i = 0; i < matrix.GetLength(0); i++)
    {
        for (int j = 0; j < matrix.GetLength(1); j++)
        {
            matrix[i, j] = -1;
        }
    }
}
```

For more information, see return.

Async Methods

By using the async feature, you can invoke asynchronous methods without using explicit callbacks or manually splitting your code across multiple methods or lambda expressions.

If you mark a method with the async modifier, you can use the await operator in the method. When control reaches an await expression in the async method, control returns to the caller, and progress in the method is suspended until the awaited task completes. When the task is complete, execution can resume in the method.

> **NOTE**
> An async method returns to the caller when either it encounters the first awaited object that's not yet complete or it gets to the end of the async method, whichever occurs first.

An async method can have a return type of Task<TResult>, Task, or void. The void return type is used primarily to define event handlers, where a void return type is required. An async method that returns void can't be awaited, and the caller of a void-returning method can't catch exceptions that the method throws.

In the following example, `DelayAsync` is an async method that has a return type of Task<TResult>. `DelayAsync` has a `return` statement that returns an integer. Therefore the method declaration of `DelayAsync` must have a return type of `Task<int>`. Because the return type is `Task<int>`, the evaluation of the `await` expression in `DoSomethingAsync` produces an integer as the following statement demonstrates: `int result = await delayTask`.

The `startButton_Click` method is an example of an async method that has a return type of void. Because `DoSomethingAsync` is an async method, the task for the call to `DoSomethingAsync` must be awaited, as the following statement shows: `await DoSomethingAsync();`. The `startButton_Click` method must be defined with the `async` modifier because the method has an `await` expression.

```
// using System.Diagnostics;
// using System.Threading.Tasks;

// This Click event is marked with the async modifier.
private async void startButton_Click(object sender, RoutedEventArgs e)
{
    await DoSomethingAsync();
}

private async Task DoSomethingAsync()
{
    Task<int> delayTask = DelayAsync();
    int result = await delayTask;

    // The previous two statements may be combined into
    // the following statement.
    //int result = await DelayAsync();

    Debug.WriteLine("Result: " + result);
}

private async Task<int> DelayAsync()
{
    await Task.Delay(100);
    return 5;
}

// Output:
//  Result: 5
```

An async method can't declare any ref or out parameters, but it can call methods that have such parameters.

For more information about async methods, see Asynchronous Programming with async and await, Control Flow in Async Programs, and Async Return Types.

Expression Body Definitions

It is common to have method definitions that simply return immediately with the result of an expression, or that have a single statement as the body of the method. There is a syntax shortcut for defining such methods using `=>`:

```
public Point Move(int dx, int dy) => new Point(x + dx, y + dy);
public void Print() => Console.WriteLine(First + " " + Last);
// Works with operators, properties, and indexers too.
public static Complex operator +(Complex a, Complex b) => a.Add(b);
public string Name => First + " " + Last;
public Customer this[long id] => store.LookupCustomer(id);
```

If the method returns `void` or is an async method, then the body of the method must be a statement expression (same as with lambdas). For properties and indexers, they must be read only, and you don't use the `get` accessor keyword.

Iterators

An iterator performs a custom iteration over a collection, such as a list or an array. An iterator uses the yield return statement to return each element one at a time. When a yield return statement is reached, the current location in code is remembered. Execution is restarted from that location when the iterator is called the next time.

You call an iterator from client code by using a foreach statement.

The return type of an iterator can be IEnumerable, IEnumerable<T>, IEnumerator, or IEnumerator<T>.

For more information, see Iterators.

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
- Classes and Structs
- Access Modifiers
- Static Classes and Static Class Members
- Inheritance
- Abstract and Sealed Classes and Class Members
- params
- return
- out
- ref
- Passing Parameters

Local functions (C# Programming Guide)

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

Starting with C# 7.0, C# supports *local functions*. Local functions are private methods of a type that are nested in another member. They can only be called from their containing member. Local functions can be declared in and called from:

- Methods, especially iterator methods and async methods
- Constructors
- Property accessors
- Event accessors
- Anonymous methods
- Lambda expressions
- Finalizers
- Other local functions

However, local functions can't be declared inside an expression-bodied member.

> **NOTE**
> In some cases, you can use a lambda expression to implement functionality also supported by a local function. For a comparison, see Local functions compared to Lambda expressions.

Local functions make the intent of your code clear. Anyone reading you code can see that the method is not callable except by the containing method. For team projects, they also make it impossible for another developer to mistakenly call the method directly from elsewhere in the class or struct.

Local function syntax

A local function is defined as a nested method inside a containing member. Its definition has the following syntax:

```
<modifiers: async | unsafe> <return-type> <method-name> <parameter-list>
```

Local functions can use the async and unsafe modifiers.

Note that all local variables that are defined in the containing member, including its method parameters, are accessible in the local function.

Unlike a method definition, a local function definition cannot include the following elements:

- The member access modifier. Because all local functions are private, including an access modifier, such as the `private` keyword, generates compiler error CS0106, "The modifier 'private' is not valid for this item."
- The static keyword. Including the `static` keyword generates compiler error CS0106, "The modifier 'static' is not valid for this item."

In addition, attributes can't be applied to the local function or to its parameters and type parameters.

The following example defines a local function named `AppendPathSeparator` that is private to a method named `GetText`:

```
using System;
using System.IO;

class Example
{
    static void Main()
    {
        string contents = GetText(@"C:\temp", "example.txt");
        Console.WriteLine("Contents of the file:\n" + contents);
    }

    private static string GetText(string path, string filename)
    {
         var sr = File.OpenText(AppendPathSeparator(path) + filename);
         var text = sr.ReadToEnd();
         return text;

         // Declare a local function.
         string AppendPathSeparator(string filepath)
         {
            if (! filepath.EndsWith(@"\"))
               filepath += @"\";

            return filepath;
         }
    }
}
```

Local functions and exceptions

One of the useful features of local functions is that they can allow exceptions to surface immediately. For method iterators, exceptions are surfaced only when the returned sequence is enumerated, and not when the iterator is retrieved. For async methods, any exceptions thrown in an async method are observed when the returned task is awaited.

The following example defines an `OddSequence` method that enumerates odd numbers between a specified range. Because it passes a number greater than 100 to the `OddSequence` enumerator method, the method throws an ArgumentOutOfRangeException. As the output from the example shows, the exception surfaces only when you iterate the numbers, and not when you retrieve the enumerator.

```
using System;
using System.Collections.Generic;

class Example
{
   static void Main()
   {
      IEnumerable<int> ienum = OddSequence(50, 110);
      Console.WriteLine("Retrieved enumerator...");

      foreach (var i in ienum)
      {
         Console.Write($"{i} ");
      }
   }

   public static IEnumerable<int> OddSequence(int start, int end)
   {
      if (start < 0 || start > 99)
         throw new ArgumentOutOfRangeException("start must be between 0 and 99.");
      if (end > 100)
         throw new ArgumentOutOfRangeException("end must be less than or equal to 100.");
      if (start >= end)
         throw new ArgumentException("start must be less than end.");

      for (int i = start; i <= end; i++)
      {
         if (i % 2 == 1)
            yield return i;
      }
   }
}
// The example displays the following output:
//    Retrieved enumerator...
//
//    Unhandled Exception: System.ArgumentOutOfRangeException: Specified argument was out of the range of valid
values.
//    Parameter name: end must be less than or equal to 100.
//       at Sequence.<GetNumericRange>d__1.MoveNext() in Program.cs:line 23
//       at Example.Main() in Program.cs:line 43
```

Instead, you can throw an exception when performing validation and before retrieving the iterator by returning the iterator from a local function, as the following example shows.

```
using System;
using System.Collections.Generic;

class Example
{
   static void Main()
   {
      IEnumerable<int> ienum = OddSequence(50, 110);
      Console.WriteLine("Retrieved enumerator...");

      foreach (var i in ienum)
      {
         Console.Write($"{i} ");
      }
   }

   public static IEnumerable<int> OddSequence(int start, int end)
   {
      if (start < 0 || start > 99)
         throw new ArgumentOutOfRangeException("start must be between 0 and 99.");
      if (end > 100)
         throw new ArgumentOutOfRangeException("end must be less than or equal to 100.");
      if (start >= end)
         throw new ArgumentException("start must be less than end.");

      return GetOddSequenceEnumerator();

      IEnumerable<int> GetOddSequenceEnumerator()
      {
         for (int i = start; i <= end; i++)
         {
            if (i % 2 == 1)
               yield return i;
         }
      }
   }
}
// The example displays the following output:
//    Unhandled Exception: System.ArgumentOutOfRangeException: Specified argument was out of the range of valid
values.
//    Parameter name: end must be less than or equal to 100.
//       at Sequence.<GetNumericRange>d__1.MoveNext() in Program.cs:line 23
//       at Example.Main() in Program.cs:line 43
```

Local functions can be used in a similar way to handle exceptions outside of the asynchronous operation. Ordinarily, exceptions thrown in async method require that you examine the inner exceptions of an AggregateException. Local functions allow your code to fail fast and allow your exception to be both thrown and observed synchronously.

The following example uses an asynchronous method named `GetMultipleAsync` to pause for a specified number of seconds and return a value that is a random multiple of that number of seconds. The maximum delay is 5 seconds; an ArgumentOutOfRangeException results if the value is greater than 5. As the following example shows, the exception that is thrown when a value of 6 is passed to the `GetMultipleAsync` method is wrapped in an AggregateException after the `GetMultipleAsync` method begins execution.

```
using System;
using System.Threading.Tasks;

class Example
{
   static void Main()
   {
      int result = GetMultipleAsync(6).Result;
      Console.WriteLine($"The returned value is {result:N0}");
   }

   static async Task<int> GetMultipleAsync(int secondsDelay)
   {
      Console.WriteLine("Executing GetMultipleAsync...");
      if (secondsDelay < 0 || secondsDelay > 5)
         throw new ArgumentOutOfRangeException("secondsDelay cannot exceed 5.");

      await Task.Delay(secondsDelay * 1000);
      return secondsDelay * new Random().Next(2,10);
   }
}
// The example displays the following output:
//    Executing GetMultipleAsync...
//
//    Unhandled Exception: System.AggregateException:
//         One or more errors occurred. (Specified argument was out of the range of valid values.
//    Parameter name: secondsDelay cannot exceed 5.) --->
//         System.ArgumentOutOfRangeException: Specified argument was out of the range of valid values.
//    Parameter name: secondsDelay cannot exceed 5.
//       at Example.<GetMultiple>d__1.MoveNext() in Program.cs:line 17
//       --- End of inner exception stack trace ---
//       at System.Threading.Tasks.Task.ThrowIfExceptional(Boolean includeTaskCanceledExceptions)
//       at System.Threading.Tasks.Task`1.GetResultCore(Boolean waitCompletionNotification)
//       at Example.Main() in C:\Users\ronpet\Documents\Visual Studio 2017\Projects\local-
functions\async1\Program.cs:line 8
```

As we did with the method iterator, we can refactor the code from this example to perform the validation before calling the asynchronous method. As the output from the following example shows, the ArgumentOutOfRangeException is not wrapped in a AggregateException.

```
using System;
using System.Threading.Tasks;

class Example
{
   static void Main()
   {
      int result = GetMultiple(6).Result;
      Console.WriteLine($"The returned value is {result:N0}");
   }

   static Task<int> GetMultiple(int secondsDelay)
   {
      if (secondsDelay < 0 || secondsDelay > 5)
         throw new ArgumentOutOfRangeException("secondsDelay cannot exceed 5.");

      return GetValueAsync();

      async Task<int> GetValueAsync()
      {
         Console.WriteLine("Executing GetValueAsync...");
         await Task.Delay(secondsDelay * 1000);
         return secondsDelay * new Random().Next(2,10);
      }
   }
}
// The example displays the following output:
//    Unhandled Exception: System.ArgumentOutOfRangeException:
//       Specified argument was out of the range of valid values.
//    Parameter name: secondsDelay cannot exceed 5.
//       at Example.GetMultiple(Int32 secondsDelay) in Program.cs:line 17
//       at Example.Main() in Program.cs:line 8
```

See also

- Methods

Ref returns and ref locals

4/8/2019 • 6 minutes to read • Edit Online

Starting with C# 7.0, C# supports reference return values (ref returns). A reference return value allows a method to return a reference to a variable, rather than a value, back to a caller. The caller can then choose to treat the returned variable as if it were returned by value or by reference. The caller can create a new variable that is itself a reference to the returned value, called a ref local.

What is a reference return value?

Most developers are familiar with passing an argument to a called method *by reference*. A called method's argument list includes a variable passed by reference. Any changes made to its value by the called method are observed by the caller. A *reference return value* means that a method returns a *reference* (or an alias) to some variable. That variable's scope must include the method. That variable's lifetime must extend beyond the return of the method. Modifications to the method's return value by the caller are made to the variable that is returned by the method.

Declaring that a method returns a *reference return value* indicates that the method returns an alias to a variable. The design intent is often that the calling code should have access to that variable through the alias, including to modify it. It follows that methods returning by reference can't have the return type `void`.

There are some restrictions on the expression that a method can return as a reference return value. Restrictions include:

- The return value must have a lifetime that extends beyond the execution of the method. In other words, it cannot be a local variable in the method that returns it. It can be an instance or static field of a class, or it can be an argument passed to the method. Attempting to return a local variable generates compiler error CS8168, "Cannot return local 'obj' by reference because it is not a ref local."
- The return value cannot be the literal `null`. Returning `null` generates compiler error CS8156, "An expression cannot be used in this context because it may not be returned by reference."

  A method with a ref return can return an alias to a variable whose value is currently the null (uninstantiated) value or a nullable type for a value type.
- The return value cannot be a constant, an enumeration member, the by-value return value from a property, or a method of a `class` or `struct`. Violating this rule generates compiler error CS8156, "An expression cannot be used in this context because it may not be returned by reference."

In addition, reference return values are not allowed on async methods. An asynchronous method may return before it has finished execution, while its return value is still unknown.

Defining a ref return value

A method that returns a *reference return value* must satisfy the following two conditions:

- The method signature includes the ref keyword in front of the return type.
- Each return statement in the method body includes the ref keyword in front of the name of the returned instance.

The following example shows a method that satisfies those conditions and returns a reference to a `Person` object named `p`:

```
public ref Person GetContactInformation(string fname, string lname)
{
    // ...method implementation...
    return ref p;
}
```

Consuming a ref return value

The ref return value is an alias to another variable in the called method's scope. You can interpret any use of the ref return as using the variable it aliases:

- When you assign its value, you are assigning a value to the variable it aliases.
- When you read its value, you are reading the value of the variable it aliases.
- If you return it *by reference*, you are returning an alias to that same variable.
- If you pass it to another method *by reference*, you are passing a reference to the variable it aliases.
- When you make a ref local alias, you make a new alias to the same variable.

Ref locals

Assume the `GetContactInformation` method is declared as a ref return:

```
public ref Person GetContactInformation(string fname, string lname)
```

A by-value assignment reads the value of a variable and assigns it to a new variable:

```
Person p = contacts.GetContactInformation("Brandie", "Best");
```

The preceding assignment declares `p` as a local variable. Its initial value is copied from reading the value returned by `GetContactInformation`. Any future assignments to `p` will not change the value of the variable returned by `GetContactInformation`. The variable `p` is no longer an alias to the variable returned.

You declare a *ref local* variable to copy the alias to the original value. In the following assignment, `p` is an alias to the variable returned from `GetContactInformation`.

```
ref Person p = ref contacts.GetContactInformation("Brandie", "Best");
```

Subsequent usage of `p` is the same as using the variable returned by `GetContactInformation` because `p` is an alias for that variable. Changes to `p` also change the variable returned from `GetContactInformation`.

The `ref` keyword is used both before the local variable declaration *and* before the method call.

You can access a value by reference in the same way. In some cases, accessing a value by reference increases performance by avoiding a potentially expensive copy operation. For example, the following statement shows how one can define a ref local value that is used to reference a value.

```
ref VeryLargeStruct reflocal = ref veryLargeStruct;
```

The `ref` keyword is used both before the local variable declaration *and* before the value in the second example. Failure to include both `ref` keywords in the variable declaration and assignment in both examples results in compiler error CS8172, "Cannot initialize a by-reference variable with a value."

Prior to C# 7.3, ref local variables couldn't be reassigned to refer to different storage after being initialized. That

restriction has been removed. The following example shows a reassignment:

```
ref VeryLargeStruct reflocal = ref veryLargeStruct; // initialization
refLocal = ref anotherVeryLargeStruct; // reassigned, refLocal refers to different storage.
```

Ref local variables must still be initialized when they are declared.

Ref returns and ref locals: an example

The following example defines a `NumberStore` class that stores an array of integer values. The `FindNumber` method returns by reference the first number that is greater than or equal to the number passed as an argument. If no number is greater than or equal to the argument, the method returns the number in index 0.

```
using System;

class NumberStore
{
    int[] numbers = { 1, 3, 7, 15, 31, 63, 127, 255, 511, 1023 };

    public ref int FindNumber(int target)
    {
        for (int ctr = 0; ctr < numbers.Length; ctr++)
        {
            if (numbers[ctr] >= target)
                return ref numbers[ctr];
        }
        return ref numbers[0];
    }

    public override string ToString() => string.Join(" ", numbers);
}
```

The following example calls the `NumberStore.FindNumber` method to retrieve the first value that is greater than or equal to 16. The caller then doubles the value returned by the method. The output from the example shows the change reflected in the value of the array elements of the `NumberStore` instance.

```
var store = new NumberStore();
Console.WriteLine($"Original sequence: {store.ToString()}");
int number = 16;
ref var value = ref store.FindNumber(number);
value *= 2;
Console.WriteLine($"New sequence:      {store.ToString()}");
// The example displays the following output:
//       Original sequence: 1 3 7 15 31 63 127 255 511 1023
//       New sequence:      1 3 7 15 62 63 127 255 511 1023
```

Without support for reference return values, such an operation is performed by returning the index of the array element along with its value. The caller can then use this index to modify the value in a separate method call. However, the caller can also modify the index to access and possibly modify other array values.

The following example shows how the `FindNumber` method could be rewritten after C# 7.3 to use ref local reassignment:

```
using System;

class NumberStore
{
    int[] numbers = { 1, 3, 7, 15, 31, 63, 127, 255, 511, 1023 };

    public ref int FindNumber(int target)
    {
        ref int returnVal = ref numbers[0];
        var ctr = numbers.Length - 1;
        while ((ctr > 0) && numbers[ctr] >= target)
        {
            returnVal = ref numbers[ctr];
            ctr--;
        }
        return ref returnVal;
    }

    public override string ToString() => string.Join(" ", numbers);
}
```

This second version is more efficient with longer sequences in scenarios where the number sought is closer to the end of the array.

See also

- ref keyword
- Write safe efficient code

Passing Parameters (C# Programming Guide)

8/19/2019 • 2 minutes to read • Edit Online

In C#, arguments can be passed to parameters either by value or by reference. Passing by reference enables function members, methods, properties, indexers, operators, and constructors to change the value of the parameters and have that change persist in the calling environment. To pass a parameter by reference with the intent of changing the value, use the `ref`, or `out` keyword. To pass by reference with the intent of avoiding copying but not changing the value, use the `in` modifier. For simplicity, only the `ref` keyword is used in the examples in this topic. For more information about the difference between `in`, `ref`, and `out`, see in, ref, and out.

The following example illustrates the difference between value and reference parameters.

```
class Program
{
    static void Main(string[] args)
    {
        int arg;

        // Passing by value.
        // The value of arg in Main is not changed.
        arg = 4;
        squareVal(arg);
        Console.WriteLine(arg);
        // Output: 4

        // Passing by reference.
        // The value of arg in Main is changed.
        arg = 4;
        squareRef(ref arg);
        Console.WriteLine(arg);
        // Output: 16
    }

    static void squareVal(int valParameter)
    {
        valParameter *= valParameter;
    }

    // Passing by reference
    static void squareRef(ref int refParameter)
    {
        refParameter *= refParameter;
    }
}
```

For more information, see the following topics:

- Passing Value-Type Parameters
- Passing Reference-Type Parameters

C# Language Specification

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

See also

- C# Programming Guide
- Methods

Passing Value-Type Parameters (C# Programming Guide)

8/19/2019 • 3 minutes to read • Edit Online

A value-type variable contains its data directly as opposed to a reference-type variable, which contains a reference to its data. Passing a value-type variable to a method by value means passing a copy of the variable to the method. Any changes to the parameter that take place inside the method have no affect on the original data stored in the argument variable. If you want the called method to change the value of the argument, you must pass it by reference, using the ref or out keyword. You may also use the in keyword to pass a value parameter by reference to avoid the copy while guaranteeing that the value will not be changed. For simplicity, the following examples use `ref`.

Passing Value Types by Value

The following example demonstrates passing value-type parameters by value. The variable `n` is passed by value to the method `SquareIt`. Any changes that take place inside the method have no affect on the original value of the variable.

```
class PassingValByVal
{
    static void SquareIt(int x)
    // The parameter x is passed by value.
    // Changes to x will not affect the original value of x.
    {
        x *= x;
        System.Console.WriteLine("The value inside the method: {0}", x);
    }
    static void Main()
    {
        int n = 5;
        System.Console.WriteLine("The value before calling the method: {0}", n);

        SquareIt(n);  // Passing the variable by value.
        System.Console.WriteLine("The value after calling the method: {0}", n);

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
/* Output:
    The value before calling the method: 5
    The value inside the method: 25
    The value after calling the method: 5
*/
```

The variable `n` is a value type. It contains its data, the value `5`. When `SquareIt` is invoked, the contents of `n` are copied into the parameter `x`, which is squared inside the method. In `Main`, however, the value of `n` is the same after calling the `SquareIt` method as it was before. The change that takes place inside the method only affects the local variable `x`.

Passing Value Types by Reference

The following example is the same as the previous example, except that the argument is passed as a `ref` parameter. The value of the underlying argument, `n`, is changed when `x` is changed in the method.

```
class PassingValByRef
{
    static void SquareIt(ref int x)
    // The parameter x is passed by reference.
    // Changes to x will affect the original value of x.
    {
        x *= x;
        System.Console.WriteLine("The value inside the method: {0}", x);
    }
    static void Main()
    {
        int n = 5;
        System.Console.WriteLine("The value before calling the method: {0}", n);

        SquareIt(ref n);  // Passing the variable by reference.
        System.Console.WriteLine("The value after calling the method: {0}", n);

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
/* Output:
    The value before calling the method: 5
    The value inside the method: 25
    The value after calling the method: 25
*/
```

In this example, it is not the value of `n` that is passed; rather, a reference to `n` is passed. The parameter `x` is not an int; it is a reference to an `int`, in this case, a reference to `n`. Therefore, when `x` is squared inside the method, what actually is squared is what `x` refers to, `n`.

Swapping Value Types

A common example of changing the values of arguments is a swap method, where you pass two variables to the method, and the method swaps their contents. You must pass the arguments to the swap method by reference. Otherwise, you swap local copies of the parameters inside the method, and no change occurs in the calling method. The following example swaps integer values.

```
static void SwapByRef(ref int x, ref int y)
{
    int temp = x;
    x = y;
    y = temp;
}
```

When you call the `SwapByRef` method, use the `ref` keyword in the call, as shown in the following example.

```
static void Main()
{
    int i = 2, j = 3;
    System.Console.WriteLine("i = {0}  j = {1}" , i, j);

    SwapByRef (ref i, ref j);

    System.Console.WriteLine("i = {0}  j = {1}" , i, j);

    // Keep the console window open in debug mode.
    System.Console.WriteLine("Press any key to exit.");
    System.Console.ReadKey();
}
/* Output:
    i = 2  j = 3
    i = 3  j = 2
*/
```

See also

- C# Programming Guide
- Passing Parameters
- Passing Reference-Type Parameters

Passing Reference-Type Parameters (C# Programming Guide)

8/19/2019 • 4 minutes to read • Edit Online

A variable of a reference type does not contain its data directly; it contains a reference to its data. When you pass a reference-type parameter by value, it is possible to change the data belonging to the referenced object, such as the value of a class member. However, you cannot change the value of the reference itself; for example, you cannot use the same reference to allocate memory for a new object and have it persist outside the method. To do that, pass the parameter using the ref or out keyword. For simplicity, the following examples use `ref`.

Passing Reference Types by Value

The following example demonstrates passing a reference-type parameter, `arr`, by value, to a method, `Change`. Because the parameter is a reference to `arr`, it is possible to change the values of the array elements. However, the attempt to reassign the parameter to a different memory location only works inside the method and does not affect the original variable, `arr`.

```
class PassingRefByVal
{
    static void Change(int[] pArray)
    {
        pArray[0] = 888;  // This change affects the original element.
        pArray = new int[5] {-3, -1, -2, -3, -4};   // This change is local.
        System.Console.WriteLine("Inside the method, the first element is: {0}", pArray[0]);
    }

    static void Main()
    {
        int[] arr = {1, 4, 5};
        System.Console.WriteLine("Inside Main, before calling the method, the first element is: {0}", arr
[0]);

        Change(arr);
        System.Console.WriteLine("Inside Main, after calling the method, the first element is: {0}", arr [0]);
    }
}
/* Output:
    Inside Main, before calling the method, the first element is: 1
    Inside the method, the first element is: -3
    Inside Main, after calling the method, the first element is: 888
*/
```

In the preceding example, the array, `arr`, which is a reference type, is passed to the method without the `ref` parameter. In such a case, a copy of the reference, which points to `arr`, is passed to the method. The output shows that it is possible for the method to change the contents of an array element, in this case from `1` to `888`. However, allocating a new portion of memory by using the new operator inside the `Change` method makes the variable `pArray` reference a new array. Thus, any changes after that will not affect the original array, `arr`, which is created inside `Main`. In fact, two arrays are created in this example, one inside `Main` and one inside the `Change` method.

Passing Reference Types by Reference

The following example is the same as the previous example, except that the `ref` keyword is added to the method

header and call. Any changes that take place in the method affect the original variable in the calling program.

```
class PassingRefByRef
{
    static void Change(ref int[] pArray)
    {
        // Both of the following changes will affect the original variables:
        pArray[0] = 888;
        pArray = new int[5] {-3, -1, -2, -3, -4};
        System.Console.WriteLine("Inside the method, the first element is: {0}", pArray[0]);
    }

    static void Main()
    {
        int[] arr = {1, 4, 5};
        System.Console.WriteLine("Inside Main, before calling the method, the first element is: {0}", arr[0]);

        Change(ref arr);
        System.Console.WriteLine("Inside Main, after calling the method, the first element is: {0}", arr[0]);
    }
}
/* Output:
    Inside Main, before calling the method, the first element is: 1
    Inside the method, the first element is: -3
    Inside Main, after calling the method, the first element is: -3
*/
```

All of the changes that take place inside the method affect the original array in `Main`. In fact, the original array is reallocated using the `new` operator. Thus, after calling the `Change` method, any reference to `arr` points to the five-element array, which is created in the `Change` method.

Swapping Two Strings

Swapping strings is a good example of passing reference-type parameters by reference. In the example, two strings, `str1` and `str2`, are initialized in `Main` and passed to the `SwapStrings` method as parameters modified by the `ref` keyword. The two strings are swapped inside the method and inside `Main` as well.

```
class SwappingStrings
{
    static void SwapStrings(ref string s1, ref string s2)
    // The string parameter is passed by reference.
    // Any changes on parameters will affect the original variables.
    {
        string temp = s1;
        s1 = s2;
        s2 = temp;
        System.Console.WriteLine("Inside the method: {0} {1}", s1, s2);
    }

    static void Main()
    {
        string str1 = "John";
        string str2 = "Smith";
        System.Console.WriteLine("Inside Main, before swapping: {0} {1}", str1, str2);

        SwapStrings(ref str1, ref str2);   // Passing strings by reference
        System.Console.WriteLine("Inside Main, after swapping: {0} {1}", str1, str2);
    }
}
/* Output:
    Inside Main, before swapping: John Smith
    Inside the method: Smith John
    Inside Main, after swapping: Smith John
*/
```

In this example, the parameters need to be passed by reference to affect the variables in the calling program. If you remove the `ref` keyword from both the method header and the method call, no changes will take place in the calling program.

For more information about strings, see string.

See also

- C# Programming Guide
- Passing Parameters
- ref
- in
- out
- Reference Types

How to: Know the Difference Between Passing a Struct and Passing a Class Reference to a Method (C# Programming Guide)

8/19/2019 • 2 minutes to read • Edit Online

The following example demonstrates how passing a struct to a method differs from passing a class instance to a method. In the example, both of the arguments (struct and class instance) are passed by value, and both methods change the value of one field of the argument. However, the results of the two methods are not the same because what is passed when you pass a struct differs from what is passed when you pass an instance of a class.

Because a struct is a value type, when you pass a struct by value to a method, the method receives and operates on a copy of the struct argument. The method has no access to the original struct in the calling method and therefore can't change it in any way. The method can change only the copy.

A class instance is a reference type, not a value type. When a reference type is passed by value to a method, the method receives a copy of the reference to the class instance. That is, the method receives a copy of the address of the instance, not a copy of the instance itself. The class instance in the calling method has an address, the parameter in the called method has a copy of the address, and both addresses refer to the same object. Because the parameter contains only a copy of the address, the called method cannot change the address of the class instance in the calling method. However, the called method can use the address to access the class members that both the original address and the copy reference. If the called method changes a class member, the original class instance in the calling method also changes.

The output of the following example illustrates the difference. The value of the `willIChange` field of the class instance is changed by the call to method `ClassTaker` because the method uses the address in the parameter to find the specified field of the class instance. The `willIChange` field of the struct in the calling method is not changed by the call to method `StructTaker` because the value of the argument is a copy of the struct itself, not a copy of its address. `StructTaker` changes the copy, and the copy is lost when the call to `StructTaker` is completed.

Example

```
class TheClass
{
    public string willIChange;
}

struct TheStruct
{
    public string willIChange;
}

class TestClassAndStruct
{
    static void ClassTaker(TheClass c)
    {
        c.willIChange = "Changed";
    }

    static void StructTaker(TheStruct s)
    {
        s.willIChange = "Changed";
    }

    static void Main()
    {
        TheClass testClass = new TheClass();
        TheStruct testStruct = new TheStruct();

        testClass.willIChange = "Not Changed";
        testStruct.willIChange = "Not Changed";

        ClassTaker(testClass);
        StructTaker(testStruct);

        Console.WriteLine("Class field = {0}", testClass.willIChange);
        Console.WriteLine("Struct field = {0}", testStruct.willIChange);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
    Class field = Changed
    Struct field = Not Changed
*/
```

See also

- C# Programming Guide
- Classes
- Structs
- Passing Parameters

Implicitly typed local variables (C# Programming Guide)

5/15/2019 • 5 minutes to read • Edit Online

Local variables can be declared without giving an explicit type. The `var` keyword instructs the compiler to infer the type of the variable from the expression on the right side of the initialization statement. The inferred type may be a built-in type, an anonymous type, a user-defined type, or a type defined in the .NET Framework class library. For more information about how to initialize arrays with `var`, see Implicitly Typed Arrays.

The following examples show various ways in which local variables can be declared with `var`:

```
// i is compiled as an int
var i = 5;

// s is compiled as a string
var s = "Hello";

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

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

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

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

It is important to understand that the `var` keyword does not mean "variant" and does not indicate that the variable is loosely typed, or late-bound. It just means that the compiler determines and assigns the most appropriate type.

The `var` keyword may be used in the following contexts:

- On local variables (variables declared at method scope) as shown in the previous example.
- In a for initialization statement.

  ```
  for(var x = 1; x < 10; x++)
  ```

- In a foreach initialization statement.

  ```
  foreach(var item in list){...}
  ```

- In a using statement.

  ```
  using (var file = new StreamReader("C:\\myfile.txt")) {...}
  ```

For more information, see How to: Use Implicitly Typed Local Variables and Arrays in a Query Expression.

var and anonymous types

In many cases the use of `var` is optional and is just a syntactic convenience. However, when a variable is initialized with an anonymous type you must declare the variable as `var` if you need to access the properties of the object at a later point. This is a common scenario in LINQ query expressions. For more information, see Anonymous Types.

From the perspective of your source code, an anonymous type has no name. Therefore, if a query variable has been initialized with `var`, then the only way to access the properties in the returned sequence of objects is to use `var` as the type of the iteration variable in the `foreach` statement.

```
class ImplicitlyTypedLocals2
{
    static void Main()
    {
        string[] words = { "aPPLE", "BlUeBeRrY", "cHeRry" };

        // If a query produces a sequence of anonymous types,
        // then use var in the foreach statement to access the properties.
        var upperLowerWords =
             from w in words
             select new { Upper = w.ToUpper(), Lower = w.ToLower() };

        // Execute the query
        foreach (var ul in upperLowerWords)
        {
            Console.WriteLine("Uppercase: {0}, Lowercase: {1}", ul.Upper, ul.Lower);
        }
    }
}
/* Outputs:
    Uppercase: APPLE, Lowercase: apple
    Uppercase: BLUEBERRY, Lowercase: blueberry
    Uppercase: CHERRY, Lowercase: cherry
 */
```

Remarks

The following restrictions apply to implicitly-typed variable declarations:

- `var` can only be used when a local variable is declared and initialized in the same statement; the variable cannot be initialized to null, or to a method group or an anonymous function.
- `var` cannot be used on fields at class scope.
- Variables declared by using `var` cannot be used in the initialization expression. In other words, this expression is legal `: int i = (i = 20);` but this expression produces a compile-time error: `var i = (i = 20);`
- Multiple implicitly-typed variables cannot be initialized in the same statement.
- If a type named `var` is in scope, then the `var` keyword will resolve to that type name and will not be treated as part of an implicitly typed local variable declaration.

Implicit typing with the `var` keyword can only be applied to variables at local method scope. Implicit typing is not available for class fields as the C# compiler would encounter a logical paradox as it processed the code: the compiler needs to know the type of the field, but it cannot determine the type until the assignment expression is

analyzed, and the expression cannot be evaluated without knowing the type. Consider the following code:

```
private var bookTitles;
```

`bookTitles` is a class field given the type `var`. As the field has no expression to evaluate, it is impossible for the compiler to infer what type `bookTitles` is supposed to be. In addition, adding an expression to the field (like you would for a local variable) is also insufficient:

```
private var bookTitles = new List<string>();
```

When the compiler encounters fields during code compilation, it records each field's type before processing any expressions associated with it. The compiler encounters the same paradox trying to parse `bookTitles`: it needs to know the type of the field, but the compiler would normally determine `var`'s type by analyzing the expression, which isn't possible without knowing the type beforehand.

You may find that `var` can also be useful with query expressions in which the exact constructed type of the query variable is difficult to determine. This can occur with grouping and ordering operations.

The `var` keyword can also be useful when the specific type of the variable is tedious to type on the keyboard, or is obvious, or does not add to the readability of the code. One example where `var` is helpful in this manner is with nested generic types such as those used with group operations. In the following query, the type of the query variable is `IEnumerable<IGrouping<string, Student>>`. As long as you and others who must maintain your code understand this, there is no problem with using implicit typing for convenience and brevity.

```
// Same as previous example except we use the entire last name as a key.
// Query variable is an IEnumerable<IGrouping<string, Student>>
var studentQuery3 =
    from student in students
    group student by student.Last;
```

However, the use of `var` does have at least the potential to make your code more difficult to understand for other developers. For that reason, the C# documentation generally uses `var` only when it is required.

See also

- C# Reference
- Implicitly Typed Arrays
- How to: Use Implicitly Typed Local Variables and Arrays in a Query Expression
- Anonymous Types
- Object and Collection Initializers
- var
- LINQ Query Expressions
- LINQ (Language-Integrated Query)
- for
- foreach, in
- using Statement

How to: Use Implicitly Typed Local Variables and Arrays in a Query Expression (C# Programming Guide)

8/19/2019 • 2 minutes to read • Edit Online

You can use implicitly typed local variables whenever you want the compiler to determine the type of a local variable. You must use implicitly typed local variables to store anonymous types, which are often used in query expressions. The following examples illustrate both optional and required uses of implicitly typed local variables in queries.

Implicitly typed local variables are declared by using the var contextual keyword. For more information, see Implicitly Typed Local Variables and Implicitly Typed Arrays.

Example

The following example shows a common scenario in which the `var` keyword is required: a query expression that produces a sequence of anonymous types. In this scenario, both the query variable and the iteration variable in the `foreach` statement must be implicitly typed by using `var` because you do not have access to a type name for the anonymous type. For more information about anonymous types, see Anonymous Types.

```
private static void QueryNames(char firstLetter)
{
    // Create the query. Use of var is required because
    // the query produces a sequence of anonymous types:
    // System.Collections.Generic.IEnumerable<????>.
    var studentQuery =
        from student in students
        where student.FirstName[0] == firstLetter
        select new { student.FirstName, student.LastName };

    // Execute the query and display the results.
    foreach (var anonType in studentQuery)
    {
        Console.WriteLine("First = {0}, Last = {1}", anonType.FirstName, anonType.LastName);
    }
}
```

Example

The following example uses the `var` keyword in a situation that is similar, but in which the use of `var` is optional. Because `student.LastName` is a string, execution of the query returns a sequence of strings. Therefore, the type of `queryID` could be declared as `System.Collections.Generic.IEnumerable<string>` instead of `var`. Keyword `var` is used for convenience. In the example, the iteration variable in the `foreach` statement is explicitly typed as a string, but it could instead be declared by using `var`. Because the type of the iteration variable is not an anonymous type, the use of `var` is an option, not a requirement. Remember, `var` itself is not a type, but an instruction to the compiler to infer and assign the type.

```
// Variable queryID could be declared by using
// System.Collections.Generic.IEnumerable<string>
// instead of var.
var queryID =
    from student in students
    where student.ID > 111
    select student.LastName;

// Variable str could be declared by using var instead of string.
foreach (string str in queryID)
{
    Console.WriteLine("Last name: {0}", str);
}
```

See also

- C# Programming Guide
- Extension Methods
- LINQ (Language-Integrated Query)
- var
- LINQ Query Expressions

Extension Methods (C# Programming Guide)

8/19/2019 • 7 minutes to read • Edit Online

Extension methods enable you to "add" methods to existing types without creating a new derived type, recompiling, or otherwise modifying the original type. Extension methods are a special kind of static method, but they are called as if they were instance methods on the extended type. For client code written in C#, F# and Visual Basic, there is no apparent difference between calling an extension method and the methods that are actually defined in a type.

The most common extension methods are the LINQ standard query operators that add query functionality to the existing System.Collections.IEnumerable and System.Collections.Generic.IEnumerable<T> types. To use the standard query operators, first bring them into scope with a `using System.Linq` directive. Then any type that implements IEnumerable<T> appears to have instance methods such as GroupBy, OrderBy, Average, and so on. You can see these additional methods in IntelliSense statement completion when you type "dot" after an instance of an IEnumerable<T> type such as List<T> or Array.

The following example shows how to call the standard query operator `OrderBy` method on an array of integers. The expression in parentheses is a lambda expression. Many standard query operators take lambda expressions as parameters, but this is not a requirement for extension methods. For more information, see Lambda Expressions.

```
class ExtensionMethods2
{

    static void Main()
    {
        int[] ints = { 10, 45, 15, 39, 21, 26 };
        var result = ints.OrderBy(g => g);
        foreach (var i in result)
        {
            System.Console.Write(i + " ");
        }
    }
}
//Output: 10 15 21 26 39 45
```

Extension methods are defined as static methods but are called by using instance method syntax. Their first parameter specifies which type the method operates on, and the parameter is preceded by the this modifier. Extension methods are only in scope when you explicitly import the namespace into your source code with a `using` directive.

The following example shows an extension method defined for the System.String class. Note that it is defined inside a non-nested, non-generic static class:

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

The `WordCount` extension method can be brought into scope with this `using` directive:

```
using ExtensionMethods;
```

And it can be called from an application by using this syntax:

```
string s = "Hello Extension Methods";
int i = s.WordCount();
```

In your code you invoke the extension method with instance method syntax. However, the intermediate language (IL) generated by the compiler translates your code into a call on the static method. Therefore, the principle of encapsulation is not really being violated. In fact, extension methods cannot access private variables in the type they are extending.

For more information, see How to: Implement and Call a Custom Extension Method.

In general, you will probably be calling extension methods far more often than implementing your own. Because extension methods are called by using instance method syntax, no special knowledge is required to use them from client code. To enable extension methods for a particular type, just add a `using` directive for the namespace in which the methods are defined. For example, to use the standard query operators, add this `using` directive to your code:

```
using System.Linq;
```

(You may also have to add a reference to System.Core.dll.) You will notice that the standard query operators now appear in IntelliSense as additional methods available for most IEnumerable<T> types.

Binding Extension Methods at Compile Time

You can use extension methods to extend a class or interface, but not to override them. An extension method with the same name and signature as an interface or class method will never be called. At compile time, extension methods always have lower priority than instance methods defined in the type itself. In other words, if a type has a method named `Process(int i)`, and you have an extension method with the same signature, the compiler will always bind to the instance method. When the compiler encounters a method invocation, it first looks for a match in the type's instance methods. If no match is found, it will search for any extension methods that are defined for the type, and bind to the first extension method that it finds. The following example demonstrates how the compiler determines which extension method or instance method to bind to.

Example

The following example demonstrates the rules that the C# compiler follows in determining whether to bind a method call to an instance method on the type, or to an extension method. The static class `Extensions` contains extension methods defined for any type that implements `IMyInterface`. Classes `A`, `B`, and `C` all implement the interface.

The `MethodB` extension method is never called because its name and signature exactly match methods already implemented by the classes.

When the compiler cannot find an instance method with a matching signature, it will bind to a matching extension method if one exists.

```
// Define an interface named IMyInterface.
namespace DefineIMyInterface
{
```

```
{
    using System;

    public interface IMyInterface
    {
        // Any class that implements IMyInterface must define a method
        // that matches the following signature.
        void MethodB();
    }
}


// Define extension methods for IMyInterface.
namespace Extensions
{
    using System;
    using DefineIMyInterface;

    // The following extension methods can be accessed by instances of any
    // class that implements IMyInterface.
    public static class Extension
    {
        public static void MethodA(this IMyInterface myInterface, int i)
        {
            Console.WriteLine
                ("Extension.MethodA(this IMyInterface myInterface, int i)");
        }

        public static void MethodA(this IMyInterface myInterface, string s)
        {
            Console.WriteLine
                ("Extension.MethodA(this IMyInterface myInterface, string s)");
        }

        // This method is never called in ExtensionMethodsDemo1, because each
        // of the three classes A, B, and C implements a method named MethodB
        // that has a matching signature.
        public static void MethodB(this IMyInterface myInterface)
        {
            Console.WriteLine
                ("Extension.MethodB(this IMyInterface myInterface)");
        }
    }
}


// Define three classes that implement IMyInterface, and then use them to test
// the extension methods.
namespace ExtensionMethodsDemo1
{
    using System;
    using Extensions;
    using DefineIMyInterface;

    class A : IMyInterface
    {
        public void MethodB() { Console.WriteLine("A.MethodB()"); }
    }

    class B : IMyInterface
    {
        public void MethodB() { Console.WriteLine("B.MethodB()"); }
        public void MethodA(int i) { Console.WriteLine("B.MethodA(int i)"); }
    }

    class C : IMyInterface
    {
        public void MethodB() { Console.WriteLine("C.MethodB()"); }
        public void MethodA(object obj)
```

```
        {
            Console.WriteLine("C.MethodA(object obj)");
        }
    }

    class ExtMethodDemo
    {
        static void Main(string[] args)
        {
            // Declare an instance of class A, class B, and class C.
            A a = new A();
            B b = new B();
            C c = new C();

            // For a, b, and c, call the following methods:
            //      -- MethodA with an int argument
            //      -- MethodA with a string argument
            //      -- MethodB with no argument.

            // A contains no MethodA, so each call to MethodA resolves to
            // the extension method that has a matching signature.
            a.MethodA(1);           // Extension.MethodA(IMyInterface, int)
            a.MethodA("hello");     // Extension.MethodA(IMyInterface, string)

            // A has a method that matches the signature of the following call
            // to MethodB.
            a.MethodB();            // A.MethodB()

            // B has methods that match the signatures of the following
            // method calls.
            b.MethodA(1);           // B.MethodA(int)
            b.MethodB();            // B.MethodB()

            // B has no matching method for the following call, but
            // class Extension does.
            b.MethodA("hello");     // Extension.MethodA(IMyInterface, string)

            // C contains an instance method that matches each of the following
            // method calls.
            c.MethodA(1);           // C.MethodA(object)
            c.MethodA("hello");     // C.MethodA(object)
            c.MethodB();            // C.MethodB()
        }
    }
}
/* Output:
    Extension.MethodA(this IMyInterface myInterface, int i)
    Extension.MethodA(this IMyInterface myInterface, string s)
    A.MethodB()
    B.MethodA(int i)
    B.MethodB()
    Extension.MethodA(this IMyInterface myInterface, string s)
    C.MethodA(object obj)
    C.MethodA(object obj)
    C.MethodB()
 */
```

General Guidelines

In general, we recommend that you implement extension methods sparingly and only when you have to. Whenever possible, client code that must extend an existing type should do so by creating a new type derived from the existing type. For more information, see Inheritance.

When using an extension method to extend a type whose source code you cannot change, you run the risk that a change in the implementation of the type will cause your extension method to break.

If you do implement extension methods for a given type, remember the following points:

- An extension method will never be called if it has the same signature as a method defined in the type.
- Extension methods are brought into scope at the namespace level. For example, if you have multiple static classes that contain extension methods in a single namespace named `Extensions`, they will all be brought into scope by the `using Extensions;` directive.

For a class library that you implemented, you shouldn't use extension methods to avoid incrementing the version number of an assembly. If you want to add significant functionality to a library for which you own the source code, you should follow the standard .NET Framework guidelines for assembly versioning. For more information, see Assembly Versioning.

See also

- C# Programming Guide
- Parallel Programming Samples (these include many example extension methods)
- Lambda Expressions
- Standard Query Operators Overview
- Conversion rules for Instance parameters and their impact
- Extension methods Interoperability between languages
- Extension methods and Curried Delegates
- Extension method Binding and Error reporting

How to: Implement and Call a Custom Extension Method (C# Programming Guide)

8/19/2019 • 2 minutes to read • Edit Online

This topic shows how to implement your own extension methods for any .NET type. Client code can use your extension methods by adding a reference to the DLL that contains them, and adding a using directive that specifies the namespace in which the extension methods are defined.

To define and call the extension method

1. Define a static class to contain the extension method.

   The class must be visible to client code. For more information about accessibility rules, see Access Modifiers.

2. Implement the extension method as a static method with at least the same visibility as the containing class.

3. The first parameter of the method specifies the type that the method operates on; it must be preceded with the this modifier.

4. In the calling code, add a `using` directive to specify the namespace that contains the extension method class.

5. Call the methods as if they were instance methods on the type.

   Note that the first parameter is not specified by calling code because it represents the type on which the operator is being applied, and the compiler already knows the type of your object. You only have to provide arguments for parameters 2 through `n`.

Example

The following example implements an extension method named `WordCount` in the `CustomExtensions.StringExtension` class. The method operates on the String class, which is specified as the first method parameter. The `CustomExtensions` namespace is imported into the application namespace, and the method is called inside the `Main` method.

```
using System.Linq;
using System.Text;
using System;

namespace CustomExtensions
{
    // Extension methods must be defined in a static class.
    public static class StringExtension
    {
        // This is the extension method.
        // The first parameter takes the "this" modifier
        // and specifies the type for which the method is defined.
        public static int WordCount(this String str)
        {
            return str.Split(new char[] {' ', '.','?'}, StringSplitOptions.RemoveEmptyEntries).Length;
        }
    }
}
namespace Extension_Methods_Simple
{
    // Import the extension method namespace.
    using CustomExtensions;
    class Program
    {
        static void Main(string[] args)
        {
            string s = "The quick brown fox jumped over the lazy dog.";
            // Call the method as if it were an
            // instance method on the type. Note that the first
            // parameter is not specified by the calling code.
            int i = s.WordCount();
            System.Console.WriteLine("Word count of s is {0}", i);
        }
    }
}
```

.NET Framework Security

Extension methods present no specific security vulnerabilities. They can never be used to impersonate existing methods on a type, because all name collisions are resolved in favor of the instance or static method defined by the type itself. Extension methods cannot access any private data in the extended class.

See also

- C# Programming Guide
- Extension Methods
- LINQ (Language-Integrated Query)
- Static Classes and Static Class Members
- protected
- internal
- public
- this
- namespace

How to: Create a New Method for an Enumeration (C# Programming Guide)

8/19/2019 • 2 minutes to read • Edit Online

You can use extension methods to add functionality specific to a particular enum type.

Example

In the following example, the `Grades` enumeration represents the possible letter grades that a student may receive in a class. An extension method named `Passing` is added to the `Grades` type so that each instance of that type now "knows" whether it represents a passing grade or not.

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Linq;

namespace EnumExtension
{
    // Define an extension method in a non-nested static class.
    public static class Extensions
    {
        public static Grades minPassing = Grades.D;
        public static bool Passing(this Grades grade)
        {
            return grade >= minPassing;
        }
    }

    public enum Grades { F = 0, D=1, C=2, B=3, A=4 };
    class Program
    {
        static void Main(string[] args)
        {
            Grades g1 = Grades.D;
            Grades g2 = Grades.F;
            Console.WriteLine("First {0} a passing grade.", g1.Passing() ? "is" : "is not");
            Console.WriteLine("Second {0} a passing grade.", g2.Passing() ? "is" : "is not");

            Extensions.minPassing = Grades.C;
            Console.WriteLine("\r\nRaising the bar!\r\n");
            Console.WriteLine("First {0} a passing grade.", g1.Passing() ? "is" : "is not");
            Console.WriteLine("Second {0} a passing grade.", g2.Passing() ? "is" : "is not");
        }
    }
  }
/* Output:
    First is a passing grade.
    Second is not a passing grade.

    Raising the bar!

    First is not a passing grade.
    Second is not a passing grade.
 */
```

Note that the `Extensions` class also contains a static variable that is updated dynamically and that the return value

of the extension method reflects the current value of that variable. This demonstrates that, behind the scenes, extension methods are invoked directly on the static class in which they are defined.

See also

- C# Programming Guide
- Extension Methods

Named and Optional Arguments (C# Programming Guide)

8/22/2019 • 8 minutes to read • Edit Online

C# 4 introduces named and optional arguments. *Named arguments* enable you to specify an argument for a particular parameter by associating the argument with the parameter's name rather than with the parameter's position in the parameter list. *Optional arguments* enable you to omit arguments for some parameters. Both techniques can be used with methods, indexers, constructors, and delegates.

When you use named and optional arguments, the arguments are evaluated in the order in which they appear in the argument list, not the parameter list.

Named and optional parameters, when used together, enable you to supply arguments for only a few parameters from a list of optional parameters. This capability greatly facilitates calls to COM interfaces such as the Microsoft Office Automation APIs.

Named Arguments

Named arguments free you from the need to remember or to look up the order of parameters in the parameter lists of called methods. The parameter for each argument can be specified by parameter name. For example, a function that prints order details (such as, seller name, order number & product name) can be called in the standard way by sending arguments by position, in the order defined by the function.

```
PrintOrderDetails("Gift Shop", 31, "Red Mug");
```

If you do not remember the order of the parameters but know their names, you can send the arguments in any order.

```
PrintOrderDetails(orderNum: 31, productName: "Red Mug", sellerName: "Gift Shop");
```

```
PrintOrderDetails(productName: "Red Mug", sellerName: "Gift Shop", orderNum: 31);
```

Named arguments also improve the readability of your code by identifying what each argument represents. In the example method below, the `sellerName` cannot be null or white space. As both `sellerName` and `productName` are string types, instead of sending arguments by position, it makes sense to use named arguments to disambiguate the two and reduce confusion for anyone reading the code.

Named arguments, when used with positional arguments, are valid as long as

- they're not followed by any positional arguments, or

```
PrintOrderDetails("Gift Shop", 31, productName: "Red Mug");
```

- *starting with C# 7.2*, they're used in the correct position. In the example below, the parameter `orderNum` is in the correct position but isn't explicitly named.

```
PrintOrderDetails(sellerName: "Gift Shop", 31, productName: "Red Mug");
```

However, out-of-order named arguments are invalid if they're followed by positional arguments.

```
// This generates CS1738: Named argument specifications must appear after all fixed arguments have been
specified.
PrintOrderDetails(productName: "Red Mug", 31, "Gift Shop");
```

Example

The following code implements the examples from this section along with some additional ones.

```
class NamedExample
{
    static void Main(string[] args)
    {
        // The method can be called in the normal way, by using positional arguments.
        PrintOrderDetails("Gift Shop", 31, "Red Mug");

        // Named arguments can be supplied for the parameters in any order.
        PrintOrderDetails(orderNum: 31, productName: "Red Mug", sellerName: "Gift Shop");
        PrintOrderDetails(productName: "Red Mug", sellerName: "Gift Shop", orderNum: 31);

        // Named arguments mixed with positional arguments are valid
        // as long as they are used in their correct position.
        PrintOrderDetails("Gift Shop", 31, productName: "Red Mug");
        PrintOrderDetails(sellerName: "Gift Shop", 31, productName: "Red Mug");    // C# 7.2 onwards
        PrintOrderDetails("Gift Shop", orderNum: 31, "Red Mug");                   // C# 7.2 onwards

        // However, mixed arguments are invalid if used out-of-order.
        // The following statements will cause a compiler error.
        // PrintOrderDetails(productName: "Red Mug", 31, "Gift Shop");
        // PrintOrderDetails(31, sellerName: "Gift Shop", "Red Mug");
        // PrintOrderDetails(31, "Red Mug", sellerName: "Gift Shop");
    }

    static void PrintOrderDetails(string sellerName, int orderNum, string productName)
    {
        if (string.IsNullOrWhiteSpace(sellerName))
        {
            throw new ArgumentException(message: "Seller name cannot be null or empty.", paramName:
nameof(sellerName));
        }

        Console.WriteLine($"Seller: {sellerName}, Order #: {orderNum}, Product: {productName}");
    }
}
```

Optional Arguments

The definition of a method, constructor, indexer, or delegate can specify that its parameters are required or that they are optional. Any call must provide arguments for all required parameters, but can omit arguments for optional parameters.

Each optional parameter has a default value as part of its definition. If no argument is sent for that parameter, the default value is used. A default value must be one of the following types of expressions:

- a constant expression;
- an expression of the form `new ValType()`, where `ValType` is a value type, such as an enum or a struct;
- an expression of the form default(ValType), where `ValType` is a value type.

Optional parameters are defined at the end of the parameter list, after any required parameters. If the caller provides an argument for any one of a succession of optional parameters, it must provide arguments for all preceding optional parameters. Comma-separated gaps in the argument list are not supported. For example, in the following code, instance method `ExampleMethod` is defined with one required and two optional parameters.

```
public void ExampleMethod(int required, string optionalstr = "default string",
    int optionalint = 10)
```

The following call to `ExampleMethod` causes a compiler error, because an argument is provided for the third parameter but not for the second.

```
//anExample.ExampleMethod(3, ,4);
```

However, if you know the name of the third parameter, you can use a named argument to accomplish the task.

```
anExample.ExampleMethod(3, optionalint: 4);
```

IntelliSense uses brackets to indicate optional parameters, as shown in the following illustration:

> **NOTE**
>
> You can also declare optional parameters by using the .NET OptionalAttribute class. `OptionalAttribute` parameters do not require a default value.

Example

In the following example, the constructor for `ExampleClass` has one parameter, which is optional. Instance method `ExampleMethod` has one required parameter, `required`, and two optional parameters, `optionalstr` and `optionalint`. The code in `Main` shows the different ways in which the constructor and method can be invoked.

```
namespace OptionalNamespace
{
    class OptionalExample
    {
        static void Main(string[] args)
        {
            // Instance anExample does not send an argument for the constructor's
            // optional parameter.
            ExampleClass anExample = new ExampleClass();
            anExample.ExampleMethod(1, "One", 1);
            anExample.ExampleMethod(2, "Two");
            anExample.ExampleMethod(3);

            // Instance anotherExample sends an argument for the constructor's
            // optional parameter.
            ExampleClass anotherExample = new ExampleClass("Provided name");
            anotherExample.ExampleMethod(1, "One", 1);
            anotherExample.ExampleMethod(2, "Two");
            anotherExample.ExampleMethod(3);

            // The following statements produce compiler errors.

            // An argument must be supplied for the first parameter, and it
            // must be an integer.
            //anExample.ExampleMethod("One", 1);
            //anExample.ExampleMethod();

            // You cannot leave a gap in the provided arguments.
            //anExample.ExampleMethod(3, ,4);
            //anExample.ExampleMethod(3, 4);
```

```
            // You can use a named parameter to make the previous
            // statement work.
            anExample.ExampleMethod(3, optionalint: 4);
        }
    }

    class ExampleClass
    {
        private string _name;

        // Because the parameter for the constructor, name, has a default
        // value assigned to it, it is optional.
        public ExampleClass(string name = "Default name")
        {
            _name = name;
        }

        // The first parameter, required, has no default value assigned
        // to it. Therefore, it is not optional. Both optionalstr and
        // optionalint have default values assigned to them. They are optional.
        public void ExampleMethod(int required, string optionalstr = "default string",
            int optionalint = 10)
        {
            Console.WriteLine("{0}: {1}, {2}, and {3}.", _name, required, optionalstr,
                optionalint);
        }
    }

    // The output from this example is the following:
    // Default name: 1, One, and 1.
    // Default name: 2, Two, and 10.
    // Default name: 3, default string, and 10.
    // Provided name: 1, One, and 1.
    // Provided name: 2, Two, and 10.
    // Provided name: 3, default string, and 10.
    // Default name: 3, default string, and 4.

}
```

COM Interfaces

Named and optional arguments, along with support for dynamic objects and other enhancements, greatly improve interoperability with COM APIs, such as Office Automation APIs.

For example, the AutoFormat method in the Microsoft Office Excel Range interface has seven parameters, all of which are optional. These parameters are shown in the following illustration:

```
excelApp.get_Range("A1", "B4").AutoFormat(
```

```
dynamic Range.AutoFormat([Excel.XlRangeAutoFormat Format = 1],
[object Number = Type.Missing], [object Font = Type.Missing],
[object Alignment = Type.Missing], [object Border = Type.Missing],
[object Pattern = Type.Missing], [object Width = Type.Missing])
```

In C# 3.0 and earlier versions, an argument is required for each parameter, as shown in the following example.

```
// In C# 3.0 and earlier versions, you need to supply an argument for
// every parameter. The following call specifies a value for the first
// parameter, and sends a placeholder value for the other six. The
// default values are used for those parameters.
var excelApp = new Microsoft.Office.Interop.Excel.Application();
excelApp.Workbooks.Add();
excelApp.Visible = true;

var myFormat =
    Microsoft.Office.Interop.Excel.XlRangeAutoFormat.xlRangeAutoFormatAccounting1;

excelApp.get_Range("A1", "B4").AutoFormat(myFormat, Type.Missing,
    Type.Missing, Type.Missing, Type.Missing, Type.Missing, Type.Missing);
```

However, you can greatly simplify the call to `AutoFormat` by using named and optional arguments, introduced in C# 4.0. Named and optional arguments enable you to omit the argument for an optional parameter if you do not want to change the parameter's default value. In the following call, a value is specified for only one of the seven parameters.

```
// The following code shows the same call to AutoFormat in C# 4.0. Only
// the argument for which you want to provide a specific value is listed.
excelApp.Range["A1", "B4"].AutoFormat( Format: myFormat );
```

For more information and examples, see How to: Use Named and Optional Arguments in Office Programming and How to: Access Office Interop Objects by Using Visual C# Features.

Overload Resolution

Use of named and optional arguments affects overload resolution in the following ways:

- A method, indexer, or constructor is a candidate for execution if each of its parameters either is optional or corresponds, by name or by position, to a single argument in the calling statement, and that argument can be converted to the type of the parameter.
- If more than one candidate is found, overload resolution rules for preferred conversions are applied to the arguments that are explicitly specified. Omitted arguments for optional parameters are ignored.
- If two candidates are judged to be equally good, preference goes to a candidate that does not have optional parameters for which arguments were omitted in the call. This is a consequence of a general preference in overload resolution for candidates that have fewer parameters.

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

- How to: Use Named and Optional Arguments in Office Programming
- Using Type dynamic
- Using Constructors
- Using Indexers

How to: Use Named and Optional Arguments in Office Programming (C# Programming Guide)

8/19/2019 • 5 minutes to read • Edit Online

Named arguments and optional arguments, introduced in C# 4, enhance convenience, flexibility, and readability in C# programming. In addition, these features greatly facilitate access to COM interfaces such as the Microsoft Office automation APIs.

In the following example, method ConvertToTable has sixteen parameters that represent characteristics of a table, such as number of columns and rows, formatting, borders, fonts, and colors. All sixteen parameters are optional, because most of the time you do not want to specify particular values for all of them. However, without named and optional arguments, a value or a placeholder value has to be provided for each parameter. With named and optional arguments, you specify values only for the parameters that are required for your project.

You must have Microsoft Office Word installed on your computer to complete these procedures.

> **NOTE**
>
> Your computer might show different names or locations for some of the Visual Studio user interface elements in the following instructions. The Visual Studio edition that you have and the settings that you use determine these elements. For more information, see Personalizing the IDE.

To create a new console application

1. Start Visual Studio.
2. On the **File** menu, point to **New**, and then click **Project**.
3. In the **Templates Categories** pane, expand **Visual C#**, and then click **Windows**.
4. Look in the top of the **Templates** pane to make sure that **.NET Framework 4** appears in the **Target Framework** box.
5. In the **Templates** pane, click **Console Application**.
6. Type a name for your project in the **Name** field.
7. Click **OK**.

   The new project appears in **Solution Explorer**.

To add a reference

1. In **Solution Explorer**, right-click your project's name and then click **Add Reference**. The **Add Reference** dialog box appears.
2. On the **.NET** page, select **Microsoft.Office.Interop.Word** in the **Component Name** list.
3. Click **OK**.

To add necessary using directives

1. In **Solution Explorer**, right-click the **Program.cs** file and then click **View Code**.
2. Add the following `using` directives to the top of the code file.

```
using Word = Microsoft.Office.Interop.Word;
```

To display text in a Word document

1. In the `Program` class in Program.cs, add the following method to create a Word application and a Word document. The Add method has four optional parameters. This example uses their default values. Therefore, no arguments are necessary in the calling statement.

```
static void DisplayInWord()
{
    var wordApp = new Word.Application();
    wordApp.Visible = true;
    // docs is a collection of all the Document objects currently
    // open in Word.
    Word.Documents docs = wordApp.Documents;

    // Add a document to the collection and name it doc.
    Word.Document doc = docs.Add();
}
```

2. Add the following code at the end of the method to define where to display text in the document, and what text to display.

```
// Define a range, a contiguous area in the document, by specifying
// a starting and ending character position. Currently, the document
// is empty.
Word.Range range = doc.Range(0, 0);

// Use the InsertAfter method to insert a string at the end of the
// current range.
range.InsertAfter("Testing, testing, testing. . .");
```

To run the application

1. Add the following statement to Main.

```
DisplayInWord();
```

2. Press CTRL+F5 to run the project. A Word document appears that contains the specified text.

To change the text to a table

1. Use the `ConvertToTable` method to enclose the text in a table. The method has sixteen optional parameters. IntelliSense encloses optional parameters in brackets, as shown in the following illustration.

```
range.ConvertToTable(
```

```
Word.Table Range.ConvertToTable([ref object Separator = Type.Missing], [ref object NumRows =
Type.Missing], [ref object NumColumns = Type.Missing], [ref object InitialColumnWidth =
Type.Missing], [ref object Format = Type.Missing], [ref object ApplyBorders = Type.Missing],
[ref object ApplyShading = Type.Missing], [ref object ApplyFont = Type.Missing], [ref object
ApplyColor = Type.Missing], [ref object ApplyHeadingsRows = Type.Missing], [ref object
ApplyLastRow = Type.Missing], [ref object ApplyFirstColumn = Type.Missing], [ref object
ApplyLastColumn = Type.Missing], [ref object AutoFit = Type.Missing], [ref object
AutoFitBehavior = Type.Missing], [ref object DefaultTableBehavior = Type.Missing])
```

Named and optional arguments enable you to specify values for only the parameters that you want to change. Add the following code to the end of method `DisplayInWord` to create a simple table. The argument specifies that the commas in the text string in `range` separate the cells of the table.

```
// Convert to a simple table. The table will have a single row with
// three columns.
range.ConvertToTable(Separator: ",");
```

In earlier versions of C#, the call to `ConvertToTable` requires a reference argument for each parameter, as shown in the following code.

```
// Call to ConvertToTable in Visual C# 2008 or earlier. This code
// is not part of the solution.
var missing = Type.Missing;
object separator = ",";
range.ConvertToTable(ref separator, ref missing, ref missing,
    ref missing, ref missing, ref missing, ref missing,
    ref missing, ref missing, ref missing, ref missing,
    ref missing, ref missing, ref missing, ref missing,
    ref missing);
```

2. Press CTRL+F5 to run the project.

To experiment with other parameters

1. To change the table so that it has one column and three rows, replace the last line in `DisplayInWord` with the following statement and then type CTRL+F5.

```
range.ConvertToTable(Separator: ",", AutoFit: true, NumColumns: 1);
```

2. To specify a predefined format for the table, replace the last line in `DisplayInWord` with the following statement and then type CTRL+F5. The format can be any of the WdTableFormat constants.

```
range.ConvertToTable(Separator: ",", AutoFit: true, NumColumns: 1,
    Format: Word.WdTableFormat.wdTableFormatElegant);
```

Example

The following code includes the full example.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Word = Microsoft.Office.Interop.Word;


namespace OfficeHowTo
{
    class WordProgram
    {
        static void Main(string[] args)
        {
            DisplayInWord();
        }

        static void DisplayInWord()
        {
            var wordApp = new Word.Application();
            wordApp.Visible = true;
            // docs is a collection of all the Document objects currently
            // open in Word.
            Word.Documents docs = wordApp.Documents;

            // Add a document to the collection and name it doc.
            Word.Document doc = docs.Add();

            // Define a range, a contiguous area in the document, by specifying
            // a starting and ending character position. Currently, the document
            // is empty.
            Word.Range range = doc.Range(0, 0);

            // Use the InsertAfter method to insert a string at the end of the
            // current range.
            range.InsertAfter("Testing, testing, testing. . .");

            // You can comment out any or all of the following statements to
            // see the effect of each one in the Word document.

            // Next, use the ConvertToTable method to put the text into a table.
            // The method has 16 optional parameters. You only have to specify
            // values for those you want to change.

            // Convert to a simple table. The table will have a single row with
            // three columns.
            range.ConvertToTable(Separator: ",");

            // Change to a single column with three rows..
            range.ConvertToTable(Separator: ",", AutoFit: true, NumColumns: 1);

            // Format the table.
            range.ConvertToTable(Separator: ",", AutoFit: true, NumColumns: 1,
                Format: Word.WdTableFormat.wdTableFormatElegant);
        }
    }
}
```

See also

- Named and Optional Arguments

Constructors (C# Programming Guide)

8/19/2019 • 2 minutes to read • Edit Online

Whenever a class or struct is created, its constructor is called. A class or struct may have multiple constructors that take different arguments. Constructors enable the programmer to set default values, limit instantiation, and write code that is flexible and easy to read. For more information and examples, see Using Constructors and Instance Constructors.

Parameterless constructors

If you don't provide a constructor for your class, C# creates one by default that instantiates the object and sets member variables to the default values as listed in the Default Values Table. If you don't provide a constructor for your struct, C# relies on an *implicit parameterless constructor* to automatically initialize each field of a value type to its default value as listed in the Default Values Table. For more information and examples, see Instance Constructors.

Constructor syntax

A constructor is a method whose name is the same as the name of its type. Its method signature includes only the method name and its parameter list; it does not include a return type. The following example shows the constructor for a class named `Person`.

```
public class Person
{
   private string last;
   private string first;

   public Person(string lastName, string firstName)
   {
      last = lastName;
      first = firstName;
   }

   // Remaining implementation of Person class.
}
```

If a constructor can be implemented as a single statement, you can use an expression body definition. The following example defines a `Location` class whose constructor has a single string parameter named *name*. The expression body definition assigns the argument to the `locationName` field.

```
public class Location
{
   private string locationName;

   public Location(string name) => Name = name;

   public string Name
   {
      get => locationName;
      set => locationName = value;
   }
}
```

Static constructors

The previous examples have all shown instance constructors, which create a new object. A class or struct can also have a static constructor, which initializes static members of the type. Static constructors are parameterless. If you don't provide a static constructor to initialize static fields, the C# compiler initializes static fields to their default value as listed in the Default Values Table.

The following example uses a static constructor to initialize a static field.

```
public class Adult : Person
{
   private static int minimumAge;

   public Adult(string lastName, string firstName) : base(lastName, firstName)
   { }

   static Adult()
   {
      minimumAge = 18;
   }

   // Remaining implementation of Adult class.
}
```

You can also define a static constructor with an expression body definition, as the following example shows.

```
public class Child : Person
{
   private static int maximumAge;

   public Child(string lastName, string firstName) : base(lastName, firstName)
   { }

   static Child() => maximumAge = 18;

   // Remaining implementation of Child class.
}
```

For more information and examples, see Static Constructors.

In This Section

Using Constructors

Instance Constructors

Private Constructors

Static Constructors

How to: Write a Copy Constructor

See also

- C# Programming Guide
- Classes and Structs
- Finalizers
- static
- Why Do Initializers Run In The Opposite Order As Constructors? Part One

Using Constructors (C# Programming Guide)

8/31/2019 • 4 minutes to read • Edit Online

When a class or struct is created, its constructor is called. Constructors have the same name as the class or struct, and they usually initialize the data members of the new object.

In the following example, a class named `Taxi` is defined by using a simple constructor. This class is then instantiated with the new operator. The `Taxi` constructor is invoked by the `new` operator immediately after memory is allocated for the new object.

```
public class Taxi
{
    public bool IsInitialized;
    public Taxi()
    {
        IsInitialized = true;
    }
}

class TestTaxi
{
    static void Main()
    {
        Taxi t = new Taxi();
        Console.WriteLine(t.IsInitialized);
    }
}
```

A constructor that takes no parameters is called a *parameterless constructor*. Parameterless constructors are invoked whenever an object is instantiated by using the `new` operator and no arguments are provided to `new`. For more information, see Instance Constructors.

Unless the class is static, classes without constructors are given a public parameterless constructor by the C# compiler in order to enable class instantiation. For more information, see Static Classes and Static Class Members.

You can prevent a class from being instantiated by making the constructor private, as follows:

```
class NLog
{
    // Private Constructor:
    private NLog() { }

    public static double e = Math.E;  //2.71828...
}
```

For more information, see Private Constructors.

Constructors for struct types resemble class constructors, but `structs` cannot contain an explicit parameterless constructor because one is provided automatically by the compiler. This constructor initializes each field in the `struct` to the default values. For more information, see Default Values Table. However, this parameterless constructor is only invoked if the `struct` is instantiated with `new`. For example, this code uses the parameterless constructor for Int32, so that you are assured that the integer is initialized:

```
int i = new int();
Console.WriteLine(i);
```

The following code, however, causes a compiler error because it does not use `new`, and because it tries to use an object that has not been initialized:

```
int i;
Console.WriteLine(i);
```

Alternatively, objects based on `structs` (including all built-in numeric types) can be initialized or assigned and then used as in the following example:

```
int a = 44;  // Initialize the value type...
int b;
b = 33;      // Or assign it before using it.
Console.WriteLine("{0}, {1}", a, b);
```

So calling the parameterless constructor for a value type is not required.

Both classes and `structs` can define constructors that take parameters. Constructors that take parameters must be called through a `new` statement or a base statement. Classes and `structs` can also define multiple constructors, and neither is required to define a parameterless constructor. For example:

```
public class Employee
{
    public int Salary;

    public Employee(int annualSalary)
    {
        Salary = annualSalary;
    }

    public Employee(int weeklySalary, int numberOfWeeks)
    {
        Salary = weeklySalary * numberOfWeeks;
    }
}
```

This class can be created by using either of the following statements:

```
Employee e1 = new Employee(30000);
Employee e2 = new Employee(500, 52);
```

A constructor can use the `base` keyword to call the constructor of a base class. For example:

```
public class Manager : Employee
{
    public Manager(int annualSalary)
        : base(annualSalary)
    {
        //Add further instructions here.
    }
}
```

In this example, the constructor for the base class is called before the block for the constructor is executed. The

`base` keyword can be used with or without parameters. Any parameters to the constructor can be used as parameters to `base`, or as part of an expression. For more information, see base.

In a derived class, if a base-class constructor is not called explicitly by using the `base` keyword, the parameterless constructor, if there is one, is called implicitly. This means that the following constructor declarations are effectively the same:

```
public Manager(int initialData)
{
    //Add further instructions here.
}
```

```
public Manager(int initialData)
    : base()
{
    //Add further instructions here.
}
```

If a base class does not offer a parameterless constructor, the derived class must make an explicit call to a base constructor by using `base`.

A constructor can invoke another constructor in the same object by using the this keyword. Like `base`, `this` can be used with or without parameters, and any parameters in the constructor are available as parameters to `this`, or as part of an expression. For example, the second constructor in the previous example can be rewritten using `this`:

```
public Employee(int weeklySalary, int numberOfWeeks)
    : this(weeklySalary * numberOfWeeks)
{
}
```

The use of the `this` keyword in the previous example causes this constructor to be called:

```
public Employee(int annualSalary)
{
    Salary = annualSalary;
}
```

Constructors can be marked as public, private, protected, internal, protected internal or private protected. These access modifiers define how users of the class can construct the class. For more information, see Access Modifiers.

A constructor can be declared static by using the static keyword. Static constructors are called automatically, immediately before any static fields are accessed, and are generally used to initialize static class members. For more information, see Static Constructors.

C# Language Specification

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

See also

- C# Programming Guide

- Classes and Structs
- Constructors
- Finalizers

Instance Constructors (C# Programming Guide)

8/22/2019 • 4 minutes to read • Edit Online

Instance constructors are used to create and initialize any instance member variables when you use the new expression to create an object of a class. To initialize a static class, or static variables in a non-static class, you define a static constructor. For more information, see Static Constructors.

The following example shows an instance constructor:

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

    // constructor
    public Coords()
    {
        x = 0;
        y = 0;
    }
}
```

NOTE

For clarity, this class contains public fields. The use of public fields is not a recommended programming practice because it allows any method anywhere in a program unrestricted and unverified access to an object's inner workings. Data members should generally be private, and should be accessed only through class methods and properties.

This instance constructor is called whenever an object based on the `Coords` class is created. A constructor like this one, which takes no arguments, is called a *parameterless constructor*. However, it is often useful to provide additional constructors. For example, we can add a constructor to the `Coords` class that allows us to specify the initial values for the data members:

```
// A constructor with two arguments.
public Coords(int x, int y)
{
    this.x = x;
    this.y = y;
}
```

This allows `Coords` objects to be created with default or specific initial values, like this:

```
var p1 = new Coords();
var p2 = new Coords(5, 3);
```

If a class does not have a constructor, a parameterless constructor is automatically generated and default values are used to initialize the object fields. For example, an int is initialized to 0. For more information on default values, see Default Values Table. Therefore, because the `Coords` class parameterless constructor initializes all data members to zero, it can be removed altogether without changing how the class works. A complete example using multiple constructors is provided in Example 1 later in this topic, and an example of an automatically generated constructor is provided in Example 2.

Instance constructors can also be used to call the instance constructors of base classes. The class constructor can invoke the constructor of the base class through the initializer, as follows:

```
class Circle : Shape
{
    public Circle(double radius)
        : base(radius, 0)
    {
    }
}
```

In this example, the `Circle` class passes values representing radius and height to the constructor provided by `Shape` from which `Circle` is derived. A complete example using `Shape` and `Circle` appears in this topic as Example 3.

Example 1

The following example demonstrates a class with two class constructors, one without arguments and one with two arguments.

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

    // Default constructor.
    public Coords()
    {
        x = 0;
        y = 0;
    }

    // A constructor with two arguments.
    public Coords(int x, int y)
    {
        this.x = x;
        this.y = y;
    }

    // Override the ToString method.
    public override string ToString()
    {
        return $"({x},{y})";
    }
}

class MainClass
{
    static void Main()
    {
        var p1 = new Coords();
        var p2 = new Coords(5, 3);

        // Display the results using the overriden ToString method.
        Console.WriteLine($"Coords #1 at {p1}");
        Console.WriteLine($"Coords #2 at {p2}");
        Console.ReadKey();
    }
}
/* Output:
 Coords #1 at (0,0)
 Coords #2 at (5,3)
*/
```

Example 2

In this example, the class `Person` does not have any constructors, in which case, a parameterless constructor is automatically provided and the fields are initialized to their default values.

```
public class Person
{
    public int age;
    public string name;
}

class TestPerson
{
    static void Main()
    {
        var person = new Person();

        Console.WriteLine("Name: {person.name}, Age: {person.age}");
        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
// Output:  Name: , Age: 0
```

Notice that the default value of `age` is `0` and the default value of `name` is `null`. For more information on default values, see Default Values Table.

Example 3

The following example demonstrates using the base class initializer. The `Circle` class is derived from the general class `Shape`, and the `Cylinder` class is derived from the `Circle` class. The constructor on each derived class is using its base class initializer.

```
abstract class Shape
{
    public const double pi = Math.PI;
    protected double x, y;

    public Shape(double x, double y)
    {
        this.x = x;
        this.y = y;
    }

    public abstract double Area();
}

class Circle : Shape
{
    public Circle(double radius)
        : base(radius, 0)
    {
    }
    public override double Area()
    {
        return pi * x * x;
    }
}

class Cylinder : Circle
{
    public Cylinder(double radius, double height)
        : base(radius)
    {
        y = height;
    }

    public override double Area()
    {
        return (2 * base.Area()) + (2 * pi * x * y);
    }
}

class TestShapes
{
    static void Main()
    {
        double radius = 2.5;
        double height = 3.0;

        Circle ring = new Circle(radius);
        Cylinder tube = new Cylinder(radius, height);

        Console.WriteLine("Area of the circle = {0:F2}", ring.Area());
        Console.WriteLine("Area of the cylinder = {0:F2}", tube.Area());

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
    Area of the circle = 19.63
    Area of the cylinder = 86.39
*/
```

For more examples on invoking the base class constructors, see virtual, override, and base.

See also

- C# Programming Guide
- Classes and Structs
- Constructors
- Finalizers
- static

Private Constructors (C# Programming Guide)

8/19/2019 • 2 minutes to read • Edit Online

A private constructor is a special instance constructor. It is generally used in classes that contain static members only. If a class has one or more private constructors and no public constructors, other classes (except nested classes) cannot create instances of this class. For example:

```
class NLog
{
    // Private Constructor:
    private NLog() { }

    public static double e = Math.E;  //2.71828...
}
```

The declaration of the empty constructor prevents the automatic generation of a parameterless constructor. Note that if you do not use an access modifier with the constructor it will still be private by default. However, the private modifier is usually used explicitly to make it clear that the class cannot be instantiated.

Private constructors are used to prevent creating instances of a class when there are no instance fields or methods, such as the Math class, or when a method is called to obtain an instance of a class. If all the methods in the class are static, consider making the complete class static. For more information see Static Classes and Static Class Members.

Example

The following is an example of a class using a private constructor.

```
public class Counter
{
    private Counter() { }
    public static int currentCount;
    public static int IncrementCount()
    {
        return ++currentCount;
    }
}

class TestCounter
{
    static void Main()
    {
        // If you uncomment the following statement, it will generate
        // an error because the constructor is inaccessible:
        // Counter aCounter = new Counter();   // Error

        Counter.currentCount = 100;
        Counter.IncrementCount();
        Console.WriteLine("New count: {0}", Counter.currentCount);

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
// Output: New count: 101
```

Notice that if you uncomment the following statement from the example, it will generate an error because the constructor is inaccessible because of its protection level:

```
// Counter aCounter = new Counter();   // Error
```

C# Language Specification

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

See also

- C# Programming Guide
- Classes and Structs
- Constructors
- Finalizers
- private
- public

Static Constructors (C# Programming Guide)

8/19/2019 • 4 minutes to read • Edit Online

A static constructor is used to initialize any static data, or to perform a particular action that needs to be performed once only. It is called automatically before the first instance is created or any static members are referenced.

```
class SimpleClass
{
    // Static variable that must be initialized at run time.
    static readonly long baseline;

    // Static constructor is called at most one time, before any
    // instance constructor is invoked or member is accessed.
    static SimpleClass()
    {
        baseline = DateTime.Now.Ticks;
    }
}
```

Remarks

Static constructors have the following properties:

- A static constructor does not take access modifiers or have parameters.
- A class or struct can only have one static constructor.
- Static constructors cannot be inherited or overloaded.
- A static constructor cannot be called directly and is only meant to be called by the common language runtime (CLR). It is invoked automatically.
- The user has no control on when the static constructor is executed in the program.
- A static constructor is called automatically to initialize the class before the first instance is created or any static members are referenced. A static constructor will run before an instance constructor. Note that a type's static constructor is called when a static method assigned to an event or a delegate is invoked and not when it is assigned. If static field variable initializers are present in the class of the static constructor, they will be executed in the textual order in which they appear in the class declaration immediately prior to the execution of the static constructor.
- If you don't provide a static constructor to initialize static fields, all static fields are initialized to their default value as listed in the Default Values Table.
- If a static constructor throws an exception, the runtime will not invoke it a second time, and the type will remain uninitialized for the lifetime of the application domain in which your program is running. Most commonly, a TypeInitializationException exception is thrown when a static constructor is unable to instantiate a type or for an unhandled exception occurring within a static constructor. For implicit static constructors that are not explicitly defined in source code, troubleshooting may require inspection of the intermediate language (IL) code.
- The presence of a static constructor prevents the addition of the BeforeFieldInit type attribute. This limits runtime optimization.

- A field declared as `static readonly` may only be assigned as part of its declaration or in a static constructor. When an explicit static constructor is not required, initialize static fields at declaration, rather than through a static constructor for better runtime optimization.

> **NOTE**
> Though not directly accessible, the presence of an explicit static constructor should be documented to assist with troubleshooting initialization exceptions.

Usage

- A typical use of static constructors is when the class is using a log file and the constructor is used to write entries to this file.
- Static constructors are also useful when creating wrapper classes for unmanaged code, when the constructor can call the `LoadLibrary` method.
- Static constructors are also a convenient place to enforce run-time checks on the type parameter that cannot be checked at compile-time via constraints (Type parameter constraints).

Example

In this example, class `Bus` has a static constructor. When the first instance of `Bus` is created (`bus1`), the static constructor is invoked to initialize the class. The sample output verifies that the static constructor runs only one time, even though two instances of `Bus` are created, and that it runs before the instance constructor runs.

```
public class Bus
{
    // Static variable used by all Bus instances.
    // Represents the time the first bus of the day starts its route.
    protected static readonly DateTime globalStartTime;

    // Property for the number of each bus.
    protected int RouteNumber { get; set; }

    // Static constructor to initialize the static variable.
    // It is invoked before the first instance constructor is run.
    static Bus()
    {
        globalStartTime = DateTime.Now;

        // The following statement produces the first line of output,
        // and the line occurs only once.
        Console.WriteLine("Static constructor sets global start time to {0}",
            globalStartTime.ToLongTimeString());
    }

    // Instance constructor.
    public Bus(int routeNum)
    {
        RouteNumber = routeNum;
        Console.WriteLine("Bus #{0} is created.", RouteNumber);
    }

    // Instance method.
    public void Drive()
    {
        TimeSpan elapsedTime = DateTime.Now - globalStartTime;

        // For demonstration purposes we treat milliseconds as minutes to simulate
        // actual bus times. Do not do this in your actual bus schedule program!
        Console.WriteLine("{0} is starting its route {1:N2} minutes after global start time {2}.",
```

```
                                this.RouteNumber,
                                elapsedTime.Milliseconds,
                                globalStartTime.ToShortTimeString());
        }
    }

    class TestBus
    {
        static void Main()
        {
            // The creation of this instance activates the static constructor.
            Bus bus1 = new Bus(71);

            // Create a second bus.
            Bus bus2 = new Bus(72);

            // Send bus1 on its way.
            bus1.Drive();

            // Wait for bus2 to warm up.
            System.Threading.Thread.Sleep(25);

            // Send bus2 on its way.
            bus2.Drive();

            // Keep the console window open in debug mode.
            System.Console.WriteLine("Press any key to exit.");
            System.Console.ReadKey();
        }
    }
    /* Sample output:
        Static constructor sets global start time to 3:57:08 PM.
        Bus #71 is created.
        Bus #72 is created.
        71 is starting its route 6.00 minutes after global start time 3:57 PM.
        72 is starting its route 31.00 minutes after global start time 3:57 PM.
    */
```

C# language specification

For more information, see the Static constructors section of the C# language specification.

See also

- C# Programming Guide
- Classes and Structs
- Constructors
- Static Classes and Static Class Members
- Finalizers
- Constructor Design Guidelines
- Security Warning - CA2121: Static constructors should be private

How to: Write a Copy Constructor (C# Programming Guide)

8/19/2019 • 2 minutes to read • Edit Online

C# doesn't provide a copy constructor for objects, but you can write one yourself.

Example

In the following example, the `Person` class defines a copy constructor that takes, as its argument, an instance of `Person`. The values of the properties of the argument are assigned to the properties of the new instance of `Person`. The code contains an alternative copy constructor that sends the `Name` and `Age` properties of the instance that you want to copy to the instance constructor of the class.

```
class Person
{
    // Copy constructor.
    public Person(Person previousPerson)
    {
        Name = previousPerson.Name;
        Age = previousPerson.Age;
    }

    //// Alternate copy constructor calls the instance constructor.
    //public Person(Person previousPerson)
    //    : this(previousPerson.Name, previousPerson.Age)
    //{
    //}

    // Instance constructor.
    public Person(string name, int age)
    {
        Name = name;
        Age = age;
    }

    public int Age { get; set; }

    public string Name { get; set; }

    public string Details()
    {
        return Name + " is " + Age.ToString();
    }
}

class TestPerson
{
    static void Main()
    {
        // Create a Person object by using the instance constructor.
        Person person1 = new Person("George", 40);

        // Create another Person object, copying person1.
        Person person2 = new Person(person1);

        // Change each person's age.
        person1.Age = 39;
        person2.Age = 41;

        // Change person2's name.
        person2.Name = "Charles";

        // Show details to verify that the name and age fields are distinct.
        Console.WriteLine(person1.Details());
        Console.WriteLine(person2.Details());

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
// Output:
// George is 39
// Charles is 41
```

See also

- ICloneable

- C# Programming Guide
- Classes and Structs
- Constructors
- Finalizers

Finalizers (C# Programming Guide)

8/22/2019 • 3 minutes to read • Edit Online

Finalizers (which are also called **destructors**) are used to perform any necessary final clean-up when a class instance is being collected by the garbage collector.

Remarks

- Finalizers cannot be defined in structs. They are only used with classes.
- A class can only have one finalizer.
- Finalizers cannot be inherited or overloaded.
- Finalizers cannot be called. They are invoked automatically.
- A finalizer does not take modifiers or have parameters.

For example, the following is a declaration of a finalizer for the `Car` class.

```
class Car
{
    ~Car()  // finalizer
    {
        // cleanup statements...
    }
}
```

A finalizer can also be implemented as an expression body definition, as the following example shows.

```
using System;

public class Destroyer
{
   public override string ToString() => GetType().Name;

   ~Destroyer() => Console.WriteLine($"The {ToString()} destructor is executing.");
}
```

The finalizer implicitly calls Finalize on the base class of the object. Therefore, a call to a finalizer is implicitly translated to the following code:

```
protected override void Finalize()
{
    try
    {
        // Cleanup statements...
    }
    finally
    {
        base.Finalize();
    }
}
```

This means that the `Finalize` method is called recursively for all instances in the inheritance chain, from the

most-derived to the least-derived.

> **NOTE**
> Empty finalizers should not be used. When a class contains a finalizer, an entry is created in the `Finalize` queue. When the finalizer is called, the garbage collector is invoked to process the queue. An empty finalizer just causes a needless loss of performance.

The programmer has no control over when the finalizer is called because this is determined by the garbage collector. The garbage collector checks for objects that are no longer being used by the application. If it considers an object eligible for finalization, it calls the finalizer (if any) and reclaims the memory used to store the object.

In .NET Framework applications (but not in .NET Core applications), finalizers are also called when the program exits.

It is possible to force garbage collection by calling Collect, but most of the time, this should be avoided because it may create performance issues.

Using finalizers to release resources

In general, C# does not require as much memory management as is needed when you develop with a language that does not target a runtime with garbage collection. This is because the .NET Framework garbage collector implicitly manages the allocation and release of memory for your objects. However, when your application encapsulates unmanaged resources such as windows, files, and network connections, you should use finalizers to free those resources. When the object is eligible for finalization, the garbage collector runs the `Finalize` method of the object.

Explicit release of resources

If your application is using an expensive external resource, we also recommend that you provide a way to explicitly release the resource before the garbage collector frees the object. You do this by implementing a `Dispose` method from the IDisposable interface that performs the necessary cleanup for the object. This can considerably improve the performance of the application. Even with this explicit control over resources, the finalizer becomes a safeguard to clean up resources if the call to the `Dispose` method failed.

For more details about cleaning up resources, see the following topics:

- Cleaning Up Unmanaged Resources
- Implementing a Dispose Method
- using Statement

Example

The following example creates three classes that make a chain of inheritance. The class `First` is the base class, `Second` is derived from `First`, and `Third` is derived from `Second`. All three have finalizers. In `Main`, an instance of the most-derived class is created. When the program runs, notice that the finalizers for the three classes are called automatically, and in order, from the most-derived to the least-derived.

```
class First
{
    ~First()
    {
        System.Diagnostics.Trace.WriteLine("First's destructor is called.");
    }
}

class Second : First
{
    ~Second()
    {
        System.Diagnostics.Trace.WriteLine("Second's destructor is called.");
    }
}

class Third : Second
{
    ~Third()
    {
        System.Diagnostics.Trace.WriteLine("Third's destructor is called.");
    }
}

class TestDestructors
{
    static void Main()
    {
        Third t = new Third();
    }

}
/* Output (to VS Output Window):
    Third's destructor is called.
    Second's destructor is called.
    First's destructor is called.
*/
```

C# language specification

For more information, see the Destructors section of the C# language specification.

See also

- IDisposable
- C# Programming Guide
- Constructors
- Garbage Collection

Object and Collection Initializers (C# Programming Guide)

8/19/2019 • 8 minutes to read • Edit Online

C# lets you instantiate an object or collection and perform member assignments in a single statement.

Object initializers

Object initializers let you assign values to any accessible fields or properties of an object at creation time without having to invoke a constructor followed by lines of assignment statements. The object initializer syntax enables you to specify arguments for a constructor or omit the arguments (and parentheses syntax). The following example shows how to use an object initializer with a named type, `Cat` and how to invoke the parameterless constructor. Note the use of auto-implemented properties in the `Cat` class. For more information, see Auto-Implemented Properties.

```
public class Cat
{
    // Auto-implemented properties.
    public int Age { get; set; }
    public string Name { get; set; }

    public Cat()
    {
    }

    public Cat(string name)
    {
        this.Name = name;
    }
}
```

```
Cat cat = new Cat { Age = 10, Name = "Fluffy" };
Cat sameCat = new Cat("Fluffy"){ Age = 10 };
```

The object initializers syntax allows you to create an instance, and after that it assigns the newly created object, with its assigned properties, to the variable in the assignment.

Starting with C# 6, object initializers can set indexers, in addition to assigning fields and properties. Consider this basic `Matrix` class:

```
public class Matrix
{
    private double[,] storage = new double[3, 3];

    public double this[int row, int column]
    {
        // The embedded array will throw out of range exceptions as appropriate.
        get { return storage[row, column]; }
        set { storage[row, column] = value; }
    }
}
```

You could initialize the identity matrix with the following code:

```
var identity = new Matrix
{
    [0, 0] = 1.0,
    [0, 1] = 0.0,
    [0, 2] = 0.0,

    [1, 0] = 0.0,
    [1, 1] = 1.0,
    [1, 2] = 0.0,

    [2, 0] = 0.0,
    [2, 1] = 0.0,
    [2, 2] = 1.0,
};
```

Any accessible indexer that contains an accessible setter can be used as one of the expressions in an object initializer, regardless of the number or types of arguments. The index arguments form the left side of the assignment, and the value is the right side of the expression. For example, these are all valid if `IndexersExample` has the appropriate indexers:

```
var thing = new IndexersExample {
    name = "object one",
    [1] = '1',
    [2] = '4',
    [3] = '9',
    Size = Math.PI,
    ['C',4] = "Middle C"
}
```

For the preceding code to compile, the `IndexersExample` type must have the following members:

```
public string name;
public double Size { set { ... }; }
public char this[int i] { set { ... }; }
public string this[char c, int i] {  set { ... }; }
```

Object Initializers with anonymous types

Although object initializers can be used in any context, they are especially useful in LINQ query expressions. Query expressions make frequent use of anonymous types, which can only be initialized by using an object initializer, as shown in the following declaration.

```
var pet = new { Age = 10, Name = "Fluffy" };
```

Anonymous types enable the `select` clause in a LINQ query expression to transform objects of the original sequence into objects whose value and shape may differ from the original. This is useful if you want to store only a part of the information from each object in a sequence. In the following example, assume that a product object (`p`) contains many fields and methods, and that you are only interested in creating a sequence of objects that contain the product name and the unit price.

```
var productInfos =
    from p in products
    select new { p.ProductName, p.UnitPrice };
```

When this query is executed, the `productInfos` variable will contain a sequence of objects that can be accessed in a `foreach` statement as shown in this example:

```
foreach(var p in productInfos){...}
```

Each object in the new anonymous type has two public properties that receive the same names as the properties or fields in the original object. You can also rename a field when you are creating an anonymous type; the following example renames the `UnitPrice` field to `Price`.

```
select new {p.ProductName, Price = p.UnitPrice};
```

Collection initializers

Collection initializers let you specify one or more element initializers when you initialize a collection type that implements IEnumerable and has `Add` with the appropriate signature as an instance method or an extension method. The element initializers can be a simple value, an expression, or an object initializer. By using a collection initializer, you do not have to specify multiple calls; the compiler adds the calls automatically.

The following example shows two simple collection initializers:

```
List<int> digits = new List<int> { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
List<int> digits2 = new List<int> { 0 + 1, 12 % 3, MakeInt() };
```

The following collection initializer uses object initializers to initialize objects of the `Cat` class defined in a previous example. Note that the individual object initializers are enclosed in braces and separated by commas.

```
List<Cat> cats = new List<Cat>
{
    new Cat{ Name = "Sylvester", Age=8 },
    new Cat{ Name = "Whiskers", Age=2 },
    new Cat{ Name = "Sasha", Age=14 }
};
```

You can specify null as an element in a collection initializer if the collection's `Add` method allows it.

```
List<Cat> moreCats = new List<Cat>
{
    new Cat{ Name = "Furrytail", Age=5 },
    new Cat{ Name = "Peaches", Age=4 },
    null
};
```

You can specify indexed elements if the collection supports read / write indexing.

```
var numbers = new Dictionary<int, string>
{
    [7] = "seven",
    [9] = "nine",
    [13] = "thirteen"
};
```

The preceding sample generates code that calls the Item[TKey] to set the values. Beginning with C# 6, you can initialize dictionaries and other associative containers using the following syntax. Notice that instead of indexer

syntax, with parentheses and an assignment, it uses an object with multiple values:

```
var moreNumbers = new Dictionary<int, string>
{
    {19, "nineteen" },
    {23, "twenty-three" },
    {42, "forty-two" }
};
```

This initializer example calls Add(TKey, TValue) to add the three items into the dictionary. These two different ways to initialize associative collections have slightly different behavior because of the method calls the compiler generates. Both variants work with the `Dictionary` class. Other types may only support one or the other based on their public API.

Examples

The following example combines the concepts of object and collection initializers.

```
public class InitializationSample
{
    public class Cat
    {
        // Auto-implemented properties.
        public int Age { get; set; }
        public string Name { get; set; }

        public Cat() { }

        public Cat(string name)
        {
            Name = name;
        }
    }

    public static void Main()
    {
        Cat cat = new Cat { Age = 10, Name = "Fluffy" };
        Cat sameCat = new Cat("Fluffy"){ Age = 10 };

        List<Cat> cats = new List<Cat>
        {
            new Cat { Name = "Sylvester", Age = 8 },
            new Cat { Name = "Whiskers", Age = 2 },
            new Cat { Name = "Sasha", Age = 14 }
        };

        List<Cat> moreCats = new List<Cat>
        {
            new Cat { Name = "Furrytail", Age = 5 },
            new Cat { Name = "Peaches", Age = 4 },
            null
        };

        // Display results.
        System.Console.WriteLine(cat.Name);

        foreach (Cat c in cats)
            System.Console.WriteLine(c.Name);

        foreach (Cat c in moreCats)
            if (c != null)
                System.Console.WriteLine(c.Name);
            else
                System.Console.WriteLine("List element has null value.");
    }
    // Output:
    //Fluffy
    //Sylvester
    //Whiskers
    //Sasha
    //Furrytail
    //Peaches
    //List element has null value.
}
```

The following example shows an object that implements IEnumerable and contains an `Add` method with multiple parameters, It uses a collection initializer with multiple elements per item in the list that correspond to the signature of the `Add` method.

```
    public class FullExample
    {
        class FormattedAddresses : IEnumerable<string>
        {
            private List<string> internalList = new List<string>();
            public IEnumerator<string> GetEnumerator() => internalList.GetEnumerator();

            System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator() =>
internalList.GetEnumerator();

            public void Add(string firstname, string lastname,
                string street, string city,
                string state, string zipcode) => internalList.Add(
                $@"{firstname} {lastname}
{street}
{city}, {state} {zipcode}"
                );
        }

        public static void Main()
        {
            FormattedAddresses addresses = new FormattedAddresses()
            {
                {"John", "Doe", "123 Street", "Topeka", "KS", "00000" },
                {"Jane", "Smith", "456 Street", "Topeka", "KS", "00000" }
            };

            Console.WriteLine("Address Entries:");

            foreach (string addressEntry in addresses)
            {
                Console.WriteLine("\r\n" + addressEntry);
            }
        }

        /*
         * Prints:

            Address Entries:

            John Doe
            123 Street
            Topeka, KS 00000

            Jane Smith
            456 Street
            Topeka, KS 00000
         */
    }
```

`Add` methods can use the `params` keyword to take a variable number of arguments, as shown in the following example. This example also demonstrates the custom implementation of an indexer to initialize a collection using indexes.

```
public class DictionaryExample
{
    class RudimentaryMultiValuedDictionary<TKey, TValue> : IEnumerable<KeyValuePair<TKey, List<TValue>>>
    {
        private Dictionary<TKey, List<TValue>> internalDictionary = new Dictionary<TKey, List<TValue>>();

        public IEnumerator<KeyValuePair<TKey, List<TValue>>> GetEnumerator() =>
internalDictionary.GetEnumerator();

        System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator() =>
internalDictionary.GetEnumerator();
```

```
        public List<TValue> this[TKey key]
        {
            get => internalDictionary[key];
            set => Add(key, value);
        }

        public void Add(TKey key, params TValue[] values) => Add(key, (IEnumerable<TValue>)values);

        public void Add(TKey key, IEnumerable<TValue> values)
        {
            if (!internalDictionary.TryGetValue(key, out List<TValue> storedValues))
                internalDictionary.Add(key, storedValues = new List<TValue>());

            storedValues.AddRange(values);
        }
    }

    public static void Main()
    {
        RudimentaryMultiValuedDictionary<string, string> rudimentaryMultiValuedDictionary1
            = new RudimentaryMultiValuedDictionary<string, string>()
            {
                {"Group1", "Bob", "John", "Mary" },
                {"Group2", "Eric", "Emily", "Debbie", "Jesse" }
            };
        RudimentaryMultiValuedDictionary<string, string> rudimentaryMultiValuedDictionary2
            = new RudimentaryMultiValuedDictionary<string, string>()
            {
                ["Group1"] = new List<string>() { "Bob", "John", "Mary" },
                ["Group2"] = new List<string>() { "Eric", "Emily", "Debbie", "Jesse" }
            };
        RudimentaryMultiValuedDictionary<string, string> rudimentaryMultiValuedDictionary3
            = new RudimentaryMultiValuedDictionary<string, string>()
            {
                {"Group1", new string []{ "Bob", "John", "Mary" } },
                { "Group2", new string[]{ "Eric", "Emily", "Debbie", "Jesse" } }
            };

        Console.WriteLine("Using first multi-valued dictionary created with a collection initializer:");

        foreach (KeyValuePair<string, List<string>> group in rudimentaryMultiValuedDictionary1)
        {
            Console.WriteLine($"\r\nMembers of group {group.Key}: ");

            foreach (string member in group.Value)
            {
                Console.WriteLine(member);
            }
        }

        Console.WriteLine("\r\nUsing second multi-valued dictionary created with a collection initializer
using indexing:");

        foreach (KeyValuePair<string, List<string>> group in rudimentaryMultiValuedDictionary2)
        {
            Console.WriteLine($"\r\nMembers of group {group.Key}: ");

            foreach (string member in group.Value)
            {
                Console.WriteLine(member);
            }
        }
        Console.WriteLine("\r\nUsing third multi-valued dictionary created with a collection initializer
using indexing:");

        foreach (KeyValuePair<string, List<string>> group in rudimentaryMultiValuedDictionary3)
        {
            Console.WriteLine($"\r\nMembers of group {group.Key}: ");
```

```
            foreach (string member in group.Value)
            {
                Console.WriteLine(member);
            }
        }
    }

    /*
     * Prints:

        Using first multi-valued dictionary created with a collection initializer:

        Members of group Group1:
        Bob
        John
        Mary

        Members of group Group2:
        Eric
        Emily
        Debbie
        Jesse

        Using second multi-valued dictionary created with a collection initializer using indexing:

        Members of group Group1:
        Bob
        John
        Mary

        Members of group Group2:
        Eric
        Emily
        Debbie
        Jesse

        Using third multi-valued dictionary created with a collection initializer using indexing:

        Members of group Group1:
        Bob
        John
        Mary

        Members of group Group2:
        Eric
        Emily
        Debbie
        Jesse
     */
}
```

See also

- C# Programming Guide
- LINQ Query Expressions
- Anonymous Types

How to: Initialize Objects by Using an Object Initializer (C# Programming Guide)

6/20/2019 • 2 minutes to read • Edit Online

You can use object initializers to initialize type objects in a declarative manner without explicitly invoking a constructor for the type.

The following examples show how to use object initializers with named objects. The compiler processes object initializers by first accessing the default instance constructor and then processing the member initializations. Therefore, if the parameterless constructor is declared as `private` in the class, object initializers that require public access will fail.

You must use an object initializer if you're defining an anonymous type. For more information, see How to: Return Subsets of Element Properties in a Query.

Example

The following example shows how to initialize a new `StudentName` type by using object initializers. This example sets properties in the `StudentName` type:

```
public class HowToObjectInitializers
{
    public static void Main()
    {
        // Declare a StudentName by using the constructor that has two parameters.
        StudentName student1 = new StudentName("Craig", "Playstead");

        // Make the same declaration by using an object initializer and sending
        // arguments for the first and last names. The default constructor is
        // invoked in processing this declaration, not the constructor that has
        // two parameters.
        StudentName student2 = new StudentName
        {
            FirstName = "Craig",
            LastName = "Playstead",
        };

        // Declare a StudentName by using an object initializer and sending
        // an argument for only the ID property. No corresponding constructor is
        // necessary. Only the default constructor is used to process object
        // initializers.
        StudentName student3 = new StudentName
        {
            ID = 183
        };

        // Declare a StudentName by using an object initializer and sending
        // arguments for all three properties. No corresponding constructor is
        // defined in the class.
        StudentName student4 = new StudentName
        {
            FirstName = "Craig",
            LastName = "Playstead",
            ID = 116
        };

        Console.WriteLine(student1.ToString());
        Console.WriteLine(student2.ToString());
```

```
        Console.WriteLine(student3.ToString());
        Console.WriteLine(student4.ToString());
    }
    // Output:
    // Craig  0
    // Craig  0
    //   183
    // Craig  116

    public class StudentName
    {
        // The default constructor has no parameters. The default constructor
        // is invoked in the processing of object initializers.
        // You can test this by changing the access modifier from public to
        // private. The declarations in Main that use object initializers will
        // fail.
        public StudentName() { }

        // The following constructor has parameters for two of the three
        // properties.
        public StudentName(string first, string last)
        {
            FirstName = first;
            LastName = last;
        }

        // Properties.
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public int ID { get; set; }

        public override string ToString() => FirstName + "  " + ID;
    }
}
```

Object initializers can be used to set indexers in an object. The following example defines a `BaseballTeam` class that uses an indexer to get and set players at different positions. The initializer can assign players, based on the abbreviation for the position, or the number used for each position baseball scorecards:

```
public class HowToIndexInitializer
{
    public class BaseballTeam
    {
        private string[] players = new string[9];
        private readonly List<string> positionAbbreviations = new List<string>
        {
            "P", "C", "1B", "2B", "3B", "SS", "LF", "CF", "RF"
        };

        public string this[int position]
        {
            // Baseball positions are 1 - 9.
            get { return players[position-1]; }
            set { players[position-1] = value; }
        }
        public string this[string position]
        {
            get { return players[positionAbbreviations.IndexOf(position)]; }
            set { players[positionAbbreviations.IndexOf(position)] = value; }
        }
    }

    public static void Main()
    {
        var team = new BaseballTeam
        {
            ["RF"] = "Mookie Betts",
            [4] = "Jose Altuve",
            ["CF"] = "Mike Trout"
        };

        Console.WriteLine(team["2B"]);
    }
}
```

See also

- C# Programming Guide
- Object and Collection Initializers

How to initialize a dictionary with a collection initializer (C# Programming Guide)

8/19/2019 • 2 minutes to read • Edit Online

A Dictionary<TKey,TValue> contains a collection of key/value pairs. Its Add method takes two parameters, one for the key and one for the value. One way to initialize a Dictionary<TKey,TValue>, or any collection whose `Add` method takes multiple parameters, is to enclose each set of parameters in braces as shown in the following example. Another option is to use an index initializer, also shown in the following example.

Example

In the following code example, a Dictionary<TKey,TValue> is initialized with instances of type `StudentName`. The first initialization uses the `Add` method with two arguments. The compiler generates a call to `Add` for each of the pairs of `int` keys and `StudentName` values. The second uses a public read / write indexer method of the `Dictionary` class:

```
public class HowToDictionaryInitializer
{
    class StudentName
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public int ID { get; set; }
    }

    public static void Main()
    {
        var students = new Dictionary<int, StudentName>()
        {
            { 111, new StudentName { FirstName="Sachin", LastName="Karnik", ID=211 } },
            { 112, new StudentName { FirstName="Dina", LastName="Salimzianova", ID=317 } },
            { 113, new StudentName { FirstName="Andy", LastName="Ruth", ID=198 } }
        };

        foreach(var index in Enumerable.Range(111, 3))
        {
            Console.WriteLine($"Student {index} is {students[index].FirstName} {students[index].LastName}");
        }
        Console.WriteLine();

        var students2 = new Dictionary<int, StudentName>()
        {
            [111] = new StudentName { FirstName="Sachin", LastName="Karnik", ID=211 },
            [112] = new StudentName { FirstName="Dina", LastName="Salimzianova", ID=317 } ,
            [113] = new StudentName { FirstName="Andy", LastName="Ruth", ID=198 }
        };

        foreach (var index in Enumerable.Range(111, 3))
        {
            Console.WriteLine($"Student {index} is {students2[index].FirstName} {students2[index].LastName}");
        }
    }
}
```

Note the two pairs of braces in each element of the collection in the first declaration. The innermost braces enclose the object initializer for the `StudentName`, and the outermost braces enclose the initializer for the key/value pair that

will be added to the `students` Dictionary<TKey,TValue>. Finally, the whole collection initializer for the dictionary is enclosed in braces. In the second initialization, the left side of the assignment is the key and the right side is the value, using an object initializer for `StudentName`.

See also

- C# Programming Guide
- Object and Collection Initializers

Nested Types (C# Programming Guide)

8/19/2019 • 2 minutes to read • Edit Online

A type defined within a class or struct is called a nested type. For example:

```
class Container
{
    class Nested
    {
        Nested() { }
    }
}
```

Regardless of whether the outer type is a class or a struct, nested types default to private; they are accessible only from their containing type. In the previous example, the `Nested` class is inaccessible to external types.

You can also specify an access modifier to define the accessibility of a nested type, as follows:

- Nested types of a **class** can be public, protected, internal, protected internal, private or private protected.

  However, defining a `protected`, `protected internal` or `private protected` nested class inside a sealed class generates compiler warning CS0628, "new protected member declared in sealed class."

- Nested types of a **struct** can be public, internal, or private.

The following example makes the `Nested` class public:

```
class Container
{
    public class Nested
    {
        Nested() { }
    }
}
```

The nested, or inner, type can access the containing, or outer, type. To access the containing type, pass it as an argument to the constructor of the nested type. For example:

```
public class Container
{
    public class Nested
    {
        private Container parent;

        public Nested()
        {
        }
        public Nested(Container parent)
        {
            this.parent = parent;
        }
    }
}
```

A nested type has access to all of the members that are accessible to its containing type. It can access private and

protected members of the containing type, including any inherited protected members.

In the previous declaration, the full name of class `Nested` is `Container.Nested`. This is the name used to create a new instance of the nested class, as follows:

```
Container.Nested nest = new Container.Nested();
```

See also

- C# Programming Guide
- Classes and Structs
- Access Modifiers
- Constructors

Partial Classes and Methods (C# Programming Guide)

8/19/2019 • 6 minutes to read • Edit Online

It is possible to split the definition of a class, a struct, an interface or a method over two or more source files. Each source file contains a section of the type or method definition, and all parts are combined when the application is compiled.

Partial Classes

There are several situations when splitting a class definition is desirable:

- When working on large projects, spreading a class over separate files enables multiple programmers to work on it at the same time.
- When working with automatically generated source, code can be added to the class without having to recreate the source file. Visual Studio uses this approach when it creates Windows Forms, Web service wrapper code, and so on. You can create code that uses these classes without having to modify the file created by Visual Studio.
- To split a class definition, use the partial keyword modifier, as shown here:

```
public partial class Employee
{
    public void DoWork()
    {
    }
}

public partial class Employee
{
    public void GoToLunch()
    {
    }
}
```

The `partial` keyword indicates that other parts of the class, struct, or interface can be defined in the namespace. All the parts must use the `partial` keyword. All the parts must be available at compile time to form the final type. All the parts must have the same accessibility, such as `public`, `private`, and so on.

If any part is declared abstract, then the whole type is considered abstract. If any part is declared sealed, then the whole type is considered sealed. If any part declares a base type, then the whole type inherits that class.

All the parts that specify a base class must agree, but parts that omit a base class still inherit the base type. Parts can specify different base interfaces, and the final type implements all the interfaces listed by all the partial declarations. Any class, struct, or interface members declared in a partial definition are available to all the other parts. The final type is the combination of all the parts at compile time.

> **NOTE**
>
> The `partial` modifier is not available on delegate or enumeration declarations.

The following example shows that nested types can be partial, even if the type they are nested within is not partial itself.

```
class Container
{
    partial class Nested
    {
        void Test() { }
    }
    partial class Nested
    {
        void Test2() { }
    }
}
```

At compile time, attributes of partial-type definitions are merged. For example, consider the following declarations:

```
[SerializableAttribute]
partial class Moon { }

[ObsoleteAttribute]
partial class Moon { }
```

They are equivalent to the following declarations:

```
[SerializableAttribute]
[ObsoleteAttribute]
class Moon { }
```

The following are merged from all the partial-type definitions:

- XML comments
- interfaces
- generic-type parameter attributes
- class attributes
- members

For example, consider the following declarations:

```
partial class Earth : Planet, IRotate { }
partial class Earth : IRevolve { }
```

They are equivalent to the following declarations:

```
class Earth : Planet, IRotate, IRevolve { }
```

Restrictions

There are several rules to follow when you are working with partial class definitions:

- All partial-type definitions meant to be parts of the same type must be modified with `partial`. For example, the following class declarations generate an error:

```
public partial class A { }
//public class A { }  // Error, must also be marked partial
```

- The `partial` modifier can only appear immediately before the keywords `class`, `struct`, or `interface`.
- Nested partial types are allowed in partial-type definitions as illustrated in the following example:

```
partial class ClassWithNestedClass
{
    partial class NestedClass { }
}

partial class ClassWithNestedClass
{
    partial class NestedClass { }
}
```

- All partial-type definitions meant to be parts of the same type must be defined in the same assembly and the same module (.exe or .dll file). Partial definitions cannot span multiple modules.
- The class name and generic-type parameters must match on all partial-type definitions. Generic types can be partial. Each partial declaration must use the same parameter names in the same order.
- The following keywords on a partial-type definition are optional, but if present on one partial-type definition, cannot conflict with the keywords specified on another partial definition for the same type:
  - public
  - private
  - protected
  - internal
  - abstract
  - sealed
  - base class
  - new modifier (nested parts)
  - generic constraints

For more information, see Constraints on Type Parameters.

Example 1

Description

In the following example, the fields and the constructor of the class, `Coords`, are declared in one partial class definition, and the member, `PrintCoords`, is declared in another partial class definition.

Code

```
public partial class Coords
{
    private int x;
    private int y;

    public Coords(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}

public partial class Coords
{
    public void PrintCoords()
    {
        Console.WriteLine("Coords: {0},{1}", x, y);
    }

}

class TestCoords
{
    static void Main()
    {
        Coords myCoords = new Coords(10, 15);
        myCoords.PrintCoords();

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
// Output: Coords: 10,15
```

Example 2

Description

The following example shows that you can also develop partial structs and interfaces.

Code

```
partial interface ITest
{
    void Interface_Test();
}

partial interface ITest
{
    void Interface_Test2();
}

partial struct S1
{
    void Struct_Test() { }
}

partial struct S1
{
    void Struct_Test2() { }
}
```

Partial Methods

A partial class or struct may contain a partial method. One part of the class contains the signature of the method. An optional implementation may be defined in the same part or another part. If the implementation is not supplied, then the method and all calls to the method are removed at compile time.

Partial methods enable the implementer of one part of a class to define a method, similar to an event. The implementer of the other part of the class can decide whether to implement the method or not. If the method is not implemented, then the compiler removes the method signature and all calls to the method. The calls to the method, including any results that would occur from evaluation of arguments in the calls, have no effect at run time. Therefore, any code in the partial class can freely use a partial method, even if the implementation is not supplied. No compile-time or run-time errors will result if the method is called but not implemented.

Partial methods are especially useful as a way to customize generated code. They allow for a method name and signature to be reserved, so that generated code can call the method but the developer can decide whether to implement the method. Much like partial classes, partial methods enable code created by a code generator and code created by a human developer to work together without run-time costs.

A partial method declaration consists of two parts: the definition, and the implementation. These may be in separate parts of a partial class, or in the same part. If there is no implementation declaration, then the compiler optimizes away both the defining declaration and all calls to the method.

```
// Definition in file1.cs
partial void onNameChanged();

// Implementation in file2.cs
partial void onNameChanged()
{
  // method body
}
```

- Partial method declarations must begin with the contextual keyword partial and the method must return void.
- Partial methods can have in or ref but not out parameters.
- Partial methods are implicitly private, and therefore they cannot be virtual.
- Partial methods cannot be extern, because the presence of the body determines whether they are defining or implementing.
- Partial methods can have static and unsafe modifiers.
- Partial methods can be generic. Constraints are put on the defining partial method declaration, and may optionally be repeated on the implementing one. Parameter and type parameter names do not have to be the same in the implementing declaration as in the defining one.
- You can make a delegate to a partial method that has been defined and implemented, but not to a partial method that has only been defined.

C# Language Specification

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

See also

- C# Programming Guide

- Classes
- Structs
- Interfaces
- partial (Type)

Anonymous Types (C# Programming Guide)

8/19/2019 • 3 minutes to read • Edit Online

Anonymous types provide a convenient way to encapsulate a set of read-only properties into a single object without having to explicitly define a type first. The type name is generated by the compiler and is not available at the source code level. The type of each property is inferred by the compiler.

You create anonymous types by using the new operator together with an object initializer. For more information about object initializers, see Object and Collection Initializers.

The following example shows an anonymous type that is initialized with two properties named `Amount` and `Message`.

```
var v = new { Amount = 108, Message = "Hello" };

// Rest the mouse pointer over v.Amount and v.Message in the following
// statement to verify that their inferred types are int and string.
Console.WriteLine(v.Amount + v.Message);
```

Anonymous types typically are used in the select clause of a query expression to return a subset of the properties from each object in the source sequence. For more information about queries, see LINQ Query Expressions.

Anonymous types contain one or more public read-only properties. No other kinds of class members, such as methods or events, are valid. The expression that is used to initialize a property cannot be `null`, an anonymous function, or a pointer type.

The most common scenario is to initialize an anonymous type with properties from another type. In the following example, assume that a class exists that is named `Product`. Class `Product` includes `Color` and `Price` properties, together with other properties that you are not interested in. Variable `products` is a collection of `Product` objects. The anonymous type declaration starts with the `new` keyword. The declaration initializes a new type that uses only two properties from `Product`. This causes a smaller amount of data to be returned in the query.

If you do not specify member names in the anonymous type, the compiler gives the anonymous type members the same name as the property being used to initialize them. You must provide a name for a property that is being initialized with an expression, as shown in the previous example. In the following example, the names of the properties of the anonymous type are `Color` and `Price`.

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

foreach (var v in productQuery)
{
    Console.WriteLine("Color={0}, Price={1}", v.Color, v.Price);
}
```

Typically, when you use an anonymous type to initialize a variable, you declare the variable as an implicitly typed local variable by using var. The type name cannot be specified in the variable declaration because only the compiler has access to the underlying name of the anonymous type. For more information about `var`, see Implicitly Typed Local Variables.

You can create an array of anonymously typed elements by combining an implicitly typed local variable and an implicitly typed array, as shown in the following example.

```
var anonArray = new[] { new { name = "apple", diam = 4 }, new { name = "grape", diam = 1 }};
```

Remarks

Anonymous types are class types that derive directly from object, and that cannot be cast to any type except object. The compiler provides a name for each anonymous type, although your application cannot access it. From the perspective of the common language runtime, an anonymous type is no different from any other reference type.

If two or more anonymous object initializers in an assembly specify a sequence of properties that are in the same order and that have the same names and types, the compiler treats the objects as instances of the same type. They share the same compiler-generated type information.

You cannot declare a field, a property, an event, or the return type of a method as having an anonymous type. Similarly, you cannot declare a formal parameter of a method, property, constructor, or indexer as having an anonymous type. To pass an anonymous type, or a collection that contains anonymous types, as an argument to a method, you can declare the parameter as type object. However, doing this defeats the purpose of strong typing. If you must store query results or pass them outside the method boundary, consider using an ordinary named struct or class instead of an anonymous type.

Because the Equals and GetHashCode methods on anonymous types are defined in terms of the `Equals` and `GetHashCode` methods of the properties, two instances of the same anonymous type are equal only if all their properties are equal.

See also

- C# Programming Guide
- Object and Collection Initializers
- Getting Started with LINQ in C#
- LINQ Query Expressions

How to: Return Subsets of Element Properties in a Query (C# Programming Guide)

8/31/2019 • 2 minutes to read • Edit Online

Use an anonymous type in a query expression when both of these conditions apply:

- You want to return only some of the properties of each source element.
- You do not have to store the query results outside the scope of the method in which the query is executed.

If you only want to return one property or field from each source element, then you can just use the dot operator in the `select` clause. For example, to return only the `ID` of each `student`, write the `select` clause as follows:

```
select student.ID;
```

Example

The following example shows how to use an anonymous type to return only a subset of the properties of each source element that matches the specified condition.

```
private static void QueryByScore()
{
    // Create the query. var is required because
    // the query produces a sequence of anonymous types.
    var queryHighScores =
        from student in students
        where student.ExamScores[0] > 95
        select new { student.FirstName, student.LastName };

    // Execute the query.
    foreach (var obj in queryHighScores)
    {
        // The anonymous type's properties were not named. Therefore
        // they have the same names as the Student properties.
        Console.WriteLine(obj.FirstName + ", " + obj.LastName);
    }
}
/* Output:
Adams, Terry
Fakhouri, Fadi
Garcia, Cesar
Omelchenko, Svetlana
Zabokritski, Eugene
*/
```

Note that the anonymous type uses the source element's names for its properties if no names are specified. To give new names to the properties in the anonymous type, write the `select` statement as follows:

```
select new { First = student.FirstName, Last = student.LastName };
```

If you try this in the previous example, then the `Console.WriteLine` statement must also change:

```
Console.WriteLine(student.First + " " + student.Last);
```

Compiling the Code

To run this code, copy and paste the class into a C# console application with a `using` directive for System.Linq.

See also

- C# Programming Guide
- Anonymous Types
- LINQ Query Expressions