

# Inheritance in C# and .NET

07/05/2018 • 26 minutes to read • Contributors  all

## In this article

[Prerequisites](#)

[Running the examples](#)

[Background: What is inheritance?](#)

[Implicit inheritance](#)

[Inheritance and an "is a" relationship](#)

[Designing the base class and derived classes](#)

[Designing abstract base classes and their derived classes](#)

[See also](#)

This tutorial introduces you to inheritance in C#. Inheritance is a feature of object-oriented programming languages that allows you to define a base class that provides specific functionality (data and behavior) and to define derived classes that either inherit or override that functionality.

## Prerequisites

This tutorial assumes that you've installed .NET Core. For installation instructions, see [.NET Core installation guide](#). You also need a code editor. This tutorial uses [Visual Studio Code](#), although you can use any code editor of your choice.

## Running the examples

To create and run the examples in this tutorial, you use the [dotnet](#) utility from the command line. Follow these steps for each example:

1. Create a directory to store the example.
2. Enter the `dotnet new console` command at a command prompt to create a new .NET Core project.
3. Copy and paste the code from the example into your code editor.
4. Enter the `dotnet restore` command from the command line to load or restore the project's dependencies.

#### ⓘ Note

Starting with .NET Core 2.0 SDK, you don't have to run `dotnet restore` because it's run implicitly by all commands that require a restore to occur, such as `dotnet new`, `dotnet build` and `dotnet run`. It's still a valid command in certain scenarios where doing an explicit restore makes sense, such as [continuous integration builds in Azure DevOps Services](#) or in build systems that need to explicitly control the time at which the restore occurs.

1. Enter the `dotnet run` command to compile and execute the example.

## Background: What is inheritance?

*Inheritance* is one of the fundamental attributes of object-oriented programming. It allows you to define a child class that reuses (inherits), extends, or modifies the behavior of a parent class. The class whose members are inherited is called the *base class*. The class that inherits the members of the base class is called the *derived class*.

C# and .NET support *single inheritance* only. That is, a class can only inherit from a single class. However, inheritance is transitive, which allows you to define an inheritance hierarchy for a set of types. In other words, type `D` can inherit from type `C`, which inherits from type `B`, which inherits from the base class type `A`. Because inheritance is transitive, the members of type `A` are available to type `D`.

Not all members of a base class are inherited by derived classes. The following members are not inherited:

- [Static constructors](#), which initialize the static data of a class.

- [Instance constructors](#), which you call to create a new instance of the class. Each class must define its own constructors.
- [Finalizers](#), which are called by the runtime's garbage collector to destroy instances of a class.

While all other members of a base class are inherited by derived classes, whether they are visible or not depends on their accessibility. A member's accessibility affects its visibility for derived classes as follows:

- [Private](#) members are visible only in derived classes that are nested in their base class. Otherwise, they are not visible in derived classes. In the following example, `A.B` is a nested class that derives from `A`, and `C` derives from `A`. The private `A.value` field is visible in `A.B`. However, if you remove the comments from the `C.GetValue` method and attempt to compile the example, it produces compiler error CS0122: "'A.value' is inaccessible due to its protection level."

C#

 Copy

```
using System;

public class A
{
    private int value = 10;

    public class B : A
    {
        public int GetValue()
        {
            return this.value;
        }
    }
}

public class C : A
{
    // public int GetValue()
    // {
    //     return this.value;
    // }
}
```

```
public class Example
{
    public static void Main(string[] args)
    {
        var b = new A.B();
        Console.WriteLine(b.GetValue());
    }
}
// The example displays the following output:
//      10
```

- [Protected](#) members are visible only in derived classes.
- [Internal](#) members are visible only in derived classes that are located in the same assembly as the base class. They are not visible in derived classes located in a different assembly from the base class.
- [Public](#) members are visible in derived classes and are part of the derived class' public interface. Public inherited members can be called just as if they are defined in the derived class. In the following example, class `A` defines a method named `Method1`, and class `B` inherits from class `A`. The example then calls `Method1` as if it were an instance method on `B`.

C#

 Copy

```
public class A
{
    public void Method1()
    {
        // Method implementation.
    }
}

public class B : A
{ }
```

```
public class Example
{
    public static void Main()
    {
        B b = new B();
        b.Method1();
    }
}
```

Derived classes can also *override* inherited members by providing an alternate implementation. In order to be able to override a member, the member in the base class must be marked with the [virtual](#) keyword. By default, base class members are not marked as `virtual` and cannot be overridden. Attempting to override a non-virtual member, as the following example does, generates compiler error CS0506: "<member> cannot override inherited member <member> because it is not marked virtual, abstract, or override."

C#

 Copy

```
public class A
{
    public void Method1()
    {
        // Do something.
    }
}

public class B : A
{
    public override void Method1() // Generates CS0506.
    {
        // Do something else.
    }
}
```

In some cases, a derived class *must* override the base class implementation. Base class members marked with the [abstract](#) keyword require that derived classes override them. Attempting to compile the following example generates compiler error CS0534, "<class> does not implement inherited abstract member <member>", because class `B` provides no implementation for `A.Method1`.

C#

 Copy

```
public abstract class A
{
    public abstract void Method1();
}

public class B : A // Generates CS0534.
{
    public void Method3()
    {
        // Do something.
    }
}
```

Inheritance applies only to classes and interfaces. Other type categories (structs, delegates, and enums) do not support inheritance. Because of these rules, attempting to compile code like the following example produces compiler error CS0527: "Type 'ValueType' in interface list is not an interface." The error message indicates that, although you can define the interfaces that a struct implements, inheritance is not supported.

C#

 Copy

```
using System;

public struct ValueStructure : ValueType // Generates CS0527.
{
}
```

# Implicit inheritance

Besides any types that they may inherit from through single inheritance, all types in the .NET type system implicitly inherit from [Object](#) or a type derived from it. The common functionality of [Object](#) is available to any type.

To see what implicit inheritance means, let's define a new class, `SimpleClass`, that is simply an empty class definition:

C#

 Copy

```
public class SimpleClass
{ }
```

You can then use reflection (which lets you inspect a type's metadata to get information about that type) to get a list of the members that belong to the `SimpleClass` type. Although you haven't defined any members in your `SimpleClass` class, output from the example indicates that it actually has nine members. One of these members is a parameterless (or default) constructor that is automatically supplied for the `SimpleClass` type by the C# compiler. The remaining eight are members of [Object](#), the type from which all classes and interfaces in the .NET type system ultimately implicitly inherit.

C#

 Copy

```
using System;
using System.Reflection;

public class Example
{
    public static void Main()
    {
        Type t = typeof(SimpleClass);
        BindingFlags flags = BindingFlags.Instance | BindingFlags.Static | BindingFlags.Public |
                               BindingFlags.NonPublic | BindingFlags.FlattenHierarchy;
        MemberInfo[] members = t.GetMembers(flags);
        Console.WriteLine($"Type {t.Name} has {members.Length} members: ");
        foreach (var member in members)
```

```

{
    string access = "";
    string stat = "";
    var method = member as MethodBase;
    if (method != null)
    {
        if (method.IsPublic)
            access = " Public";
        else if (method.IsPrivate)
            access = " Private";
        else if (method.IsFamily)
            access = " Protected";
        else if (method.IsAssembly)
            access = " Internal";
        else if (method.IsFamilyOrAssembly)
            access = " Protected Internal ";
        if (method.IsStatic)
            stat = " Static";
    }
    var output = $"{member.Name} ({member.MemberType}): {access}{stat}, Declared by
{member.DeclaringType}";
    Console.WriteLine(output);
}
}
}

// The example displays the following output:
// Type SimpleClass has 9 members:
// ToString (Method): Public, Declared by System.Object
// Equals (Method): Public, Declared by System.Object
// Equals (Method): Public Static, Declared by System.Object
// ReferenceEquals (Method): Public Static, Declared by System.Object
// GetHashCode (Method): Public, Declared by System.Object
// GetType (Method): Public, Declared by System.Object
// Finalize (Method): Internal, Declared by System.Object
// MemberwiseClone (Method): Internal, Declared by System.Object
// .ctor (Constructor): Public, Declared by SimpleClass

```



Implicit inheritance from the [Object](#) class makes these methods available to the `SimpleClass` class:

- The public `ToString` method, which converts a `SimpleClass` object to its string representation, returns the fully qualified type name. In this case, the `ToString` method returns the string "SimpleClass".
- Three methods that test for equality of two objects: the public instance `Equals(Object)` method, the public static `Equals(Object, Object)` method, and the public static `ReferenceEquals(Object, Object)` method. By default, these methods test for reference equality; that is, to be equal, two object variables must refer to the same object.
- The public `GetHashCode` method, which computes a value that allows an instance of the type to be used in hashed collections.
- The public `GetType` method, which returns a [Type](#) object that represents the `SimpleClass` type.
- The protected [Finalize](#) method, which is designed to release unmanaged resources before an object's memory is reclaimed by the garbage collector.
- The protected [MemberwiseClone](#) method, which creates a shallow clone of the current object.

Because of implicit inheritance, you can call any inherited member from a `SimpleClass` object just as if it was actually a member defined in the `SimpleClass` class. For instance, the following example calls the `SimpleClass.ToString` method, which `SimpleClass` inherits from [Object](#).

C#



```
using System;

public class SimpleClass
{
}

public class Example
{
    public static void Main()
```

```
{
    SimpleClass sc = new SimpleClass();
    Console.WriteLine(sc.ToString());
}
// The example displays the following output:
//      SimpleClass
```

The following table lists the categories of types that you can create in C# and the types from which they implicitly inherit. Each base type makes a different set of members available through inheritance to implicitly derived types.

Type category	Implicitly inherits from
class	<a href="#">Object</a>
struct	<a href="#">ValueType</a> , <a href="#">Object</a>
enum	<a href="#">Enum</a> , <a href="#">ValueType</a> , <a href="#">Object</a>
delegate	<a href="#">MulticastDelegate</a> , <a href="#">Delegate</a> , <a href="#">Object</a>

## Inheritance and an "is a" relationship

Ordinarily, inheritance is used to express an "is a" relationship between a base class and one or more derived classes, where the derived classes are specialized versions of the base class; the derived class is a type of the base class. For example, the `Publication` class represents a publication of any kind, and the `Book` and `Magazine` classes represent specific types of publications.

### Note

A class or struct can implement one or more interfaces. While interface implementation is often presented as a workaround for single inheritance or as a way of using inheritance with structs, it is intended to express a different relationship (a "can do" relationship) between an interface and its implementing type than inheritance. An interface defines a subset of functionality (such as the ability to test for equality, to compare or sort objects, or to support culture-sensitive parsing and formatting) that the interface makes available to its implementing types.

Note that "is a" also expresses the relationship between a type and a specific instantiation of that type. In the following example, `Automobile` is a class that has three unique read-only properties: `Make`, the manufacturer of the automobile; `Model`, the kind of automobile; and `Year`, its year of manufacture. Your `Automobile` class also has a constructor whose arguments are assigned to the property values, and it overrides the [Object.ToString](#) method to produce a string that uniquely identifies the `Automobile` instance rather than the `Automobile` class.

C#



```
using System;

public class Automobile
{
    public Automobile(string make, string model, int year)
    {
        if (make == null)
            throw new ArgumentNullException("The make cannot be null.");
        else if (String.IsNullOrEmpty(make))
            throw new ArgumentException("make cannot be an empty string or have space characters only.");
        Make = make;

        if (model == null)
            throw new ArgumentNullException("The model cannot be null.");
        else if (String.IsNullOrEmpty(model))
            throw new ArgumentException("model cannot be an empty string or have space characters only.");
        Model = model;

        if (year < 1857 || year > DateTime.Now.Year + 2)
            throw new ArgumentException("The year is out of range.");
    }
}
```

```
        Year = year;
    }

    public string Make { get; }

    public string Model { get; }

    public int Year { get; }

    public override string ToString() => $"{Year} {Make} {Model}";
}
```

In this case, you shouldn't rely on inheritance to represent specific car makes and models. For example, you don't need to define a `Packard` type to represent automobiles manufactured by the Packard Motor Car Company. Instead, you can represent them by creating an `Automobile` object with the appropriate values passed to its class constructor, as the following example does.

C#

 Copy

```
using System;

public class Example
{
    public static void Main()
    {
        var packard = new Automobile("Packard", "Custom Eight", 1948);
        Console.WriteLine(packard);
    }
}

// The example displays the following output:
//      1948 Packard Custom Eight
```

An is-a relationship based on inheritance is best applied to a base class and to derived classes that add additional members to the base class or that require additional functionality not present in the base class.

# Designing the base class and derived classes

Let's look at the process of designing a base class and its derived classes. In this section, you'll define a base class, `Publication`, which represents a publication of any kind, such as a book, a magazine, a newspaper, a journal, an article, etc. You'll also define a `Book` class that derives from `Publication`. You could easily extend the example to define other derived classes, such as `Magazine`, `Journal`, `Newspaper`, and `Article`.

## The base `Publication` class

In designing your `Publication` class, you need to make several design decisions:

- What members to include in your base `Publication` class, and whether the `Publication` members provide method implementations or whether `Publication` is an abstract base class that serves as a template for its derived classes.

In this case, the `Publication` class will provide method implementations. The [Designing abstract base classes and their derived classes](#) section contains an example that uses an abstract base class to define the methods that derived classes must override. Derived classes are free to provide any implementation that is suitable for the derived type.

The ability to reuse code (that is, multiple derived classes share the declaration and implementation of base class methods and do not need to override them) is an advantage of non-abstract base classes. Therefore, you should add members to `Publication` if their code is likely to be shared by some or most specialized `Publication` types. If you fail to provide base class implementations efficiently, you'll end up having to provide largely identical member implementations in derived classes rather a single implementation in the base class. The need to maintain duplicated code in multiple locations is a potential source of bugs.

Both to maximize code reuse and to create a logical and intuitive inheritance hierarchy, you want to be sure that you include in the `Publication` class only the data and functionality that is common to all or to most publications. Derived classes then implement members that are unique to the particular kinds of publication that they represent.

- How far to extend your class hierarchy. Do you want to develop a hierarchy of three or more classes, rather than simply a base class and one or more derived classes? For example, `Publication` could be a base class of `Periodical`, which in turn is a base class of `Magazine`, `Journal` and `Newspaper`.

For your example, you'll use the small hierarchy of a `Publication` class and a single derived class, `Book`. You could easily extend the example to create a number of additional classes that derive from `Publication`, such as `Magazine` and `Article`.

- Whether it makes sense to instantiate the base class. If it does not, you should apply the [abstract](#) keyword to the class. Otherwise, your `Publication` class can be instantiated by calling its class constructor. If an attempt is made to instantiate a class marked with the `abstract` keyword by a direct call to its class constructor, the C# compiler generates error CS0144, "Cannot create an instance of the abstract class or interface." If an attempt is made to instantiate the class by using reflection, the reflection method throws a [MemberAccessException](#).

By default, a base class can be instantiated by calling its class constructor. You do not have to explicitly define a class constructor. If one is not present in the base class' source code, the C# compiler automatically provides a default (parameterless) constructor.

For your example, you'll mark the `Publication` class as [abstract](#) so that it cannot be instantiated. An `abstract` class without any `abstract` methods indicates that this class represents an abstract concept that is shared among several concrete classes (like a `Book`, `Journal`).

- Whether derived classes must inherit the base class implementation of particular members, whether they have the option to override the base class implementation, or whether they must provide an implementation. You use the [abstract](#) keyword to force derived classes to provide an implementation. You use the [virtual](#) keyword to allow derived classes to override a base class method. By default, methods defined in the base class are *not* overridable.

The `Publication` class does not have any `abstract` methods, but the class itself is `abstract`.

- Whether a derived class represents the final class in the inheritance hierarchy and cannot itself be used as a base class for additional derived classes. By default, any class can serve as a base class. You can apply the [sealed](#) keyword to indicate that a class cannot serve as a base class for any additional classes. Attempting to derive from a sealed class generated compiler error CS0509, "cannot derive from sealed type <typeName>".

For your example, you'll mark your derived class as `sealed`.

The following example shows the source code for the `Publication` class, as well as a `PublicationType` enumeration that is returned by the `Publication.PublicationType` property. In addition to the members that it inherits from [Object](#), the `Publication` class defines the following unique members and member overrides:

C#

 Copy

```
using System;

public enum PublicationType { Misc, Book, Magazine, Article };

public abstract class Publication
{
    private bool published = false;
    private DateTime datePublished;
    private int totalPages;

    public Publication(string title, string publisher, PublicationType type)
    {
        if (publisher == null)
            throw new ArgumentNullException("The publisher cannot be null.");
        else if (String.IsNullOrEmpty(publisher))
            throw new ArgumentException("The publisher cannot consist only of white space.");
        Publisher = publisher;

        if (title == null)
            throw new ArgumentNullException("The title cannot be null.");
        else if (String.IsNullOrEmpty(title))
            throw new ArgumentException("The title cannot consist only of white space.");
    }
}
```

```
Title = title;

Type = type;
}

public string Publisher { get; }

public string Title { get; }

public PublicationType Type { get; }

public string CopyrightName { get; private set; }

public int CopyrightDate { get; private set; }

public int Pages
{
    get { return totalPages; }
    set
    {
        if (value <= 0)
            throw new ArgumentOutOfRangeException("The number of pages cannot be zero or negative.");
        totalPages = value;
    }
}

public string GetPublicationDate()
{
    if (!published)
        return "NYP";
    else
        return datePublished.ToString("d");
}

public void Publish(DateTime datePublished)
{
    published = true;
    this.datePublished = datePublished;
}
```



```
}


public void Copyright(string copyrightName, int copyrightDate)
{
    if (copyrightName == null)
        throw new ArgumentNullException("The name of the copyright holder cannot be null.");
    else if (String.IsNullOrEmpty(copyrightName))
        throw new ArgumentException("The name of the copyright holder cannot consist only of white space.");
    CopyrightName = copyrightName;

    int currentYear = DateTime.Now.Year;
    if (copyrightDate < currentYear - 10 || copyrightDate > currentYear + 2)
        throw new ArgumentOutOfRangeException($"The copyright year must be between {currentYear - 10} and {currentYear + 1}");
    CopyrightDate = copyrightDate;
}

public override string ToString() => Title;
}
```

- A constructor

Because the `Publication` class is `abstract`, it cannot be instantiated directly from code like the following example:

C#	
<pre>var publication = new Publication("Tiddlywinks for Experts", "Fun and Games",                                 PublicationType.Book);</pre>	

However, its instance constructor can be called directly from derived class constructors, as the source code for the `Book` class shows.

- Two publication-related properties

`Title` is a read-only `String` property whose value is supplied by calling the `Publication` constructor.

`Pages` is a read-write [Int32](#) property that indicates how many total pages the publication has. The value is stored in a private field named `totalPages`. It must be a positive number or an [ArgumentOutOfRangeException](#) is thrown.

- Publisher-related members

Two read-only properties, `Publisher` and `Type`. The values are originally supplied by the call to the `Publication` class constructor.

- Publishing-related members

Two methods, `Publish` and `GetPublicationDate`, set and return the publication date. The `Publish` method sets a private `published` flag to `true` when it is called and assigns the date passed to it as an argument to the private `datePublished` field. The `GetPublicationDate` method returns the string "NYP" if the `published` flag is `false`, and the value of the `datePublished` field if it is `true`.

- Copyright-related members

The `Copyright` method takes the name of the copyright holder and the year of the copyright as arguments and assigns them to the `CopyrightName` and `CopyrightDate` properties.

- An override of the `ToString` method

If a type does not override the [Object.ToString](#) method, it returns the fully qualified name of the type, which is of little use in differentiating one instance from another. The `Publication` class overrides [Object.ToString](#) to return the value of the `Title` property.

The following figure illustrates the relationship between your base `Publication` class and its implicitly inherited [Object](#) class.

## Object

Equals(Object)
Equals(Object, Object)
Finalize()
GetHashCode()
GetType()
MemberwiseClone()
ReferenceEquals()
ToString()
#ctor()

## Publication

Equals(Object)
Equals(Object, Object)
Finalize()
GetHashCode()
GetType()
MemberwiseClone()
ReferenceEquals()
ToString()
#ctor(String, String, PublicationType)
PublicationType
Publisher
Title
CopyrightDate
CopyrightName
Pages
Copyright()
GetPublicationDate()
Publish()

### Key

Unique member	
Inherited member	
Overridden member	

The **Book** class

The `Book` class represents a book as a specialized type of publication. The following example shows the source code for the `Book` class.

C#



```
using System;

public sealed class Book : Publication
{
    public Book(string title, string author, string publisher) :
        this(title, String.Empty, author, publisher)
    { }

    public Book(string title, string isbn, string author, string publisher) : base(title, publisher,
PublicationType.Book)
    {
        // isbn argument must be a 10- or 13-character numeric string without "-" characters.
        // We could also determine whether the ISBN is valid by comparing its checksum digit
        // with a computed checksum.
        //
        if (! String.IsNullOrEmpty(isbn)) {
            // Determine if ISBN length is correct.
            if (! (isbn.Length == 10 | isbn.Length == 13))
                throw new ArgumentException("The ISBN must be a 10- or 13-character numeric string.");
            ulong nISBN = 0;
            if (! UInt64.TryParse(isbn, out nISBN))
                throw new ArgumentException("The ISBN can consist of numeric characters only.");
        }
        ISBN = isbn;

        Author = author;
    }

    public string ISBN { get; }

    public string Author { get; }
```

```
public Decimal Price { get; private set; }

// A three-digit ISO currency symbol.
public string Currency { get; private set; }

// Returns the old price, and sets a new price.
public Decimal SetPrice(Decimal price, string currency)
{
    if (price < 0)
        throw new ArgumentOutOfRangeException("The price cannot be negative.");
    Decimal oldValue = Price;
    Price = price;

    if (currency.Length != 3)
        throw new ArgumentException("The ISO currency symbol is a 3-character string.");
    Currency = currency;

    return oldValue;
}

public override bool Equals(object obj)
{
    Book book = obj as Book;
    if (book == null)
        return false;
    else
        return ISBN == book.ISBN;
}

public override int GetHashCode() => ISBN.GetHashCode();

public override string ToString() => $"{(String.IsNullOrEmpty(Author) ? "" : Author + ", ")}{Title}";
}
```

In addition to the members that it inherits from `Publication`, the `Book` class defines the following unique members and member overrides:

- Two constructors

The two `Book` constructors share three common parameters. Two, *title* and *publisher*, correspond to parameters of the `Publication` constructor. The third is *author*, which is stored to a public immutable `Author` property. One constructor includes an *isbn* parameter, which is stored in the `ISBN` auto-property.

The first constructor uses the [this](#) keyword to call the other constructor. Constructor chaining is a common pattern in defining constructors. Constructors with fewer parameters provide default values when calling the constructor with the greatest number of parameters.

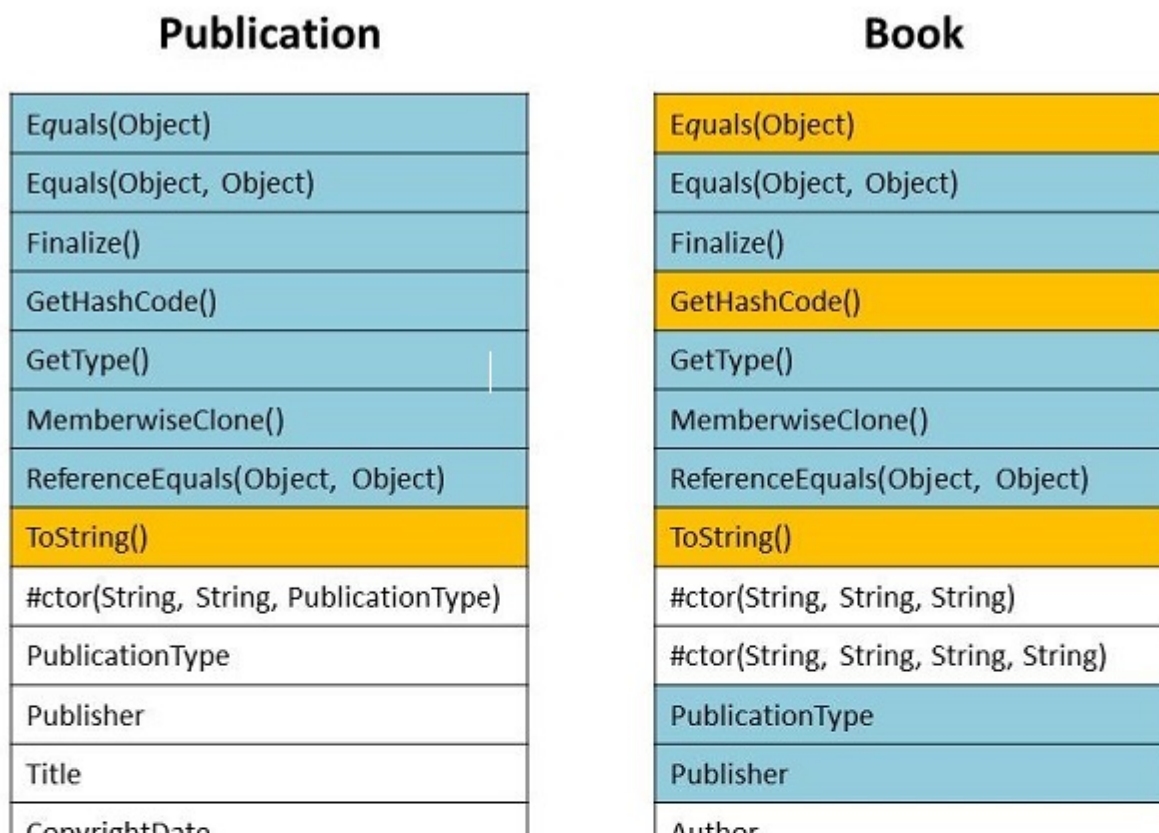
The second constructor uses the [base](#) keyword to pass the title and publisher name to the base class constructor. If you don't make an explicit call to a base class constructor in your source code, the C# compiler automatically supplies a call to the base class' default or parameterless constructor.

- A read-only `ISBN` property, which returns the `Book` object's International Standard Book Number, a unique 10- or 13-digit number. The ISBN is supplied as an argument to one of the `Book` constructors. The ISBN is stored in a private backing field, which is auto-generated by the compiler.
- A read-only `Author` property. The author name is supplied as an argument to both `Book` constructors and is stored in the property.
- Two read-only price-related properties, `Price` and `Currency`. Their values are provided as arguments in a `SetPrice` method call. The `Currency` property is the three-digit ISO currency symbol (for example, USD for the U.S. dollar). ISO currency symbols can be retrieved from the [ISOCurrencySymbol](#) property. Both of these properties are externally read-only, but both can be set by code in the `Book` class.
- A `SetPrice` method, which sets the values of the `Price` and `Currency` properties. Those values are returned by those same properties.
- Overrides to the `ToString` method (inherited from `Publication`) and the [Object.Equals\(Object\)](#) and [GetHashCode](#) methods (inherited from [Object](#)).

Unless it is overridden, the [Object.Equals\(Object\)](#) method tests for reference equality. That is, two object variables are considered to be equal if they refer to the same object. In the `Book` class, on the other hand, two `Book` objects should be equal if they have the same ISBN.

When you override the [Object.Equals\(Object\)](#) method, you must also override the [GetHashCode](#) method, which returns a value that the runtime uses to store items in hashed collections for efficient retrieval. The hash code should return a value that's consistent with the test for equality. Since you've overridden [Object.Equals\(Object\)](#) to return `true` if the ISBN properties of two `Book` objects are equal, you return the hash code computed by calling the [GetHashCode](#) method of the string returned by the `ISBN` property.

The following figure illustrates the relationship between the `Book` class and `Publication`, its base class.



CopyrightDate
CopyrightName
Pages
Copyright()
GetPublicationDate()
Publish()

**Key**

Unique member	
Inherited member	
Overridden member	

Author
Title
CopyrightDate
CopyrightName
ISBN
Pages
Price
Currency
Copyright()
GetPublicationDate()
Publish()
SetPrice()

You can now instantiate a `Book` object, invoke both its unique and inherited members, and pass it as an argument to a method that expects a parameter of type `Publication` or of type `Book`, as the following example shows.

C#

Copy

```
using System;
using static System.Console;

public class Example
{
    public static void Main()
    {
        var book = new Book("The Tempest", "0971655819", "Shakespeare, William",
                            "Public Domain Press");
    }
}
```



```
ShowPublicationInfo(book);
book.Publish(new DateTime(2016, 8, 18));
ShowPublicationInfo(book);

var book2 = new Book("The Tempest", "Classic Works Press", "Shakespeare, William");
Write($"{book.Title} and {book2.Title} are the same publication: " +
    $"{((Publication) book).Equals(book2)}");
}

public static void ShowPublicationInfo(Publication pub)
{
    string pubDate = pub.GetPublicationDate();
    WriteLine($"{pub.Title}, " +
        $"{(pubDate == "NYP" ? "Not Yet Published" : "published on " + pubDate):d} by
{pub.Publisher}");
}
}

// The example displays the following output:
//      The Tempest, Not Yet Published by Public Domain Press
//      The Tempest, published on 8/18/2016 by Public Domain Press
//      The Tempest and The Tempest are the same publication: False
```

## Designing abstract base classes and their derived classes

In the previous example, you defined a base class that provided an implementation for a number of methods to allow derived classes to share code. In many cases, however, the base class is not expected to provide an implementation. Instead, the base class is an *abstract class* that declares *abstract methods*; it serves as a template that defines the members that each derived class must implement. Typically in an abstract base class, the implementation of each derived type is unique to that type. You marked the class with the `abstract` keyword because it made no sense to instantiate a `Publication` object, although the class did provide implementations of functionality common to publications.

For example, each closed two-dimensional geometric shape includes two properties: area, the inner extent of the shape; and perimeter, or the distance along the edges of the shape. The way in which these properties are calculated, however, depends

completely on the specific shape. The formula for calculating the perimeter (or circumference) of a circle, for example, is different from that of a triangle. The `Shape` class is an `abstract` class with `abstract` methods. That indicates derived classes share the same functionality, but those derived classes implement that functionality differently.

The following example defines an abstract base class named `Shape` that defines two properties: `Area` and `Perimeter`. In addition to marking the class with the `abstract` keyword, each instance member is also marked with the `abstract` keyword. In this case, `Shape` also overrides the `Object.ToString` method to return the name of the type, rather than its fully qualified name. And it defines two static members, `GetArea` and `GetPerimeter`, that allow callers to easily retrieve the area and perimeter of an instance of any derived class. When you pass an instance of a derived class to either of these methods, the runtime calls the method override of the derived class.

C#

 Copy

```
using System;

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

    public abstract double Perimeter { get; }

    public override string ToString() => GetType().Name;

    public static double GetArea(Shape shape) => shape.Area;

    public static double GetPerimeter(Shape shape) => shape.Perimeter;
}
```

You can then derive some classes from `Shape` that represent specific shapes. The following example defines three classes, `Triangle`, `Rectangle`, and `Circle`. Each uses a formula unique for that particular shape to compute the area and perimeter. Some of the derived classes also define properties, such as `Rectangle.Diagonal` and `Circle.Diameter`, that are unique to the shape that they represent.

C#

 Copy

```
using System;

public class Square : Shape
{
    public Square(double length)
    {
        Side = length;
    }

    public double Side { get; }

    public override double Area => Math.Pow(Side, 2);

    public override double Perimeter => Side * 4;

    public double Diagonal => Math.Round(Math.Sqrt(2) * Side, 2);
}

public class Rectangle : Shape
{
    public Rectangle(double length, double width)
    {
        Length = length;
        Width = width;
    }

    public double Length { get; }

    public double Width { get; }

    public override double Area => Length * Width;

    public override double Perimeter => 2 * Length + 2 * Width;

    public bool IsSquare() => Length == Width;
```

```
    public double Diagonal => Math.Round(Math.Sqrt(Math.Pow(Length, 2) + Math.Pow(Width, 2)), 2);
}

public class Circle : Shape
{
    public Circle(double radius)
    {
        Radius = radius;
    }

    public override double Area => Math.Round(Math.PI * Math.Pow(Radius, 2), 2);

    public override double Perimeter => Math.Round(Math.PI * 2 * Radius, 2);

    // Define a circumference, since it's the more familiar term.
    public double Circumference => Perimeter;

    public double Radius { get; }

    public double Diameter => Radius * 2;
}
```

The following example uses objects derived from `Shape`. It instantiates an array of objects derived from `Shape` and calls the static methods of the `Shape` class, which wraps return `Shape` property values. The runtime retrieves values from the overridden properties of the derived types. The example also casts each `Shape` object in the array to its derived type and, if the cast succeeds, retrieves properties of that particular subclass of `Shape`.

C#

 Copy

```
using System;

public class Example
{
    public static void Main()
    {
```

```
Shape[] shapes = { new Rectangle(10, 12), new Square(5),  
                  new Circle(3) };  
foreach (var shape in shapes) {  
    Console.WriteLine($"{shape}: area, {Shape.GetArea(shape)}; " +  
                      $"perimeter, {Shape.GetPerimeter(shape)}");  
    var rect = shape as Rectangle;  
    if (rect != null) {  
        Console.WriteLine($"    Is Square: {rect.IsSquare()}, Diagonal: {rect.Diagonal}");  
        continue;  
    }  
    var sq = shape as Square;  
    if (sq != null) {  
        Console.WriteLine($"    Diagonal: {sq.Diagonal}");  
        continue;  
    }  
}  
}
```

// The example displays the following output:  
// Rectangle: area, 120; perimeter, 44  
// Is Square: False, Diagonal: 15.62  
// Square: area, 25; perimeter, 20  
// Diagonal: 7.07  
// Circle: area, 28.27; perimeter, 18.85

## See also

- [Classes and objects](#)
- [Inheritance \(C# Programming Guide\)](#)