Chapter 3. Creating Types in C#

In this chapter, we delve into types and type members.

Classes

A *class* is the most common kind of reference type. The simplest possible class declaration is as follows:

```
class YourClassName
{
}
```

A more complex class optionally has the following:
Preceding the keyword `class`
Attributes and *class modifiers*. The non-nested class modifiers are `public`, `internal`, `abstract`, `sealed`, `static`, `unsafe`, and `partial`.
Following `YourClassName`
Generic type parameters and *constraints*, a *base class*, and *interfaces*.
Within the braces

Class members (these are *methods*, *properties*, *indexers*, *events*, *fields*, *constructors*, *overloaded operators*, *nested types*, and a *finalizer*).

This chapter covers all of these constructs except attributes, operator functions, and the `unsafe` keyword, which are covered in Chapter 4. The following sections enumerate each of the class members.

Fields

A *field* is a variable that is a member of a class or struct. For example:

```
class Octopus
{
  string name;
  public int Age = 10;
}
```

Fields allow the following modifiers:

Static modifier
`static`

Access modifiers
`public internal private protected`

Inheritance modifier
`new`

Unsafe code modifier
`unsafe`

Read-only modifier
`readonly`

Threading modifier
`volatile`

There are two popular naming conventions for private fields: camel-cased (e.g., `firstName`) and camel-cased with an underscore (`_firstName`). The latter convention lets you instantly distinguish private fields from parameters and local variables.

The readonly modifier

The `readonly` modifier prevents a field from being modified after construction. A read-only field can be assigned only in its declaration or within the enclosing type's constructor.

Field initialization

Field initialization is optional. An uninitialized field has a default value (`0`, `\0`, `null`, `false`). Field initializers run before constructors:

```
public int Age = 10;
```

A field initializer can contain expressions and call methods:

```
static readonly string TempFolder =
System.IO.Path.GetTempPath();
```

Declaring multiple fields together

For convenience, you can declare multiple fields of the same type in a comma-separated list. This is a convenient way for all the fields to share the same attributes and field modifiers:

```
static readonly int legs = 8,
                    eyes = 2;
```

Constants

A *constant* is evaluated statically at compile time, and the compiler literally substitutes its value whenever used (rather like a macro in C++). A constant can be any of the built-in numeric types, `bool`, `char`, `string`, or an enum type.

A constant is declared with the `const` keyword and must be initialized with a value. For example:

```
public class Test
{
  public const string Message = "Hello World";
}
```

A constant can serve a similar role to a `static readonly` field, but it is much more restrictive—both in the types you can use and in field initialization semantics. A constant also differs from a `static readonly` field in that the evaluation of the constant occurs at compile time. Thus

```
public static double Circumference (double radius)
{
  return 2 * System.Math.PI * radius;
}
```

is compiled to:

```
public static double Circumference (double radius)
{
  return 6.2831853071795862 * radius;
}
```

It makes sense for `PI` to be a constant because its value is predetermined at compile time. In contrast, a `static readonly` field's value can potentially differ each time the program is run:

```
static readonly DateTime StartupTime = DateTime.Now;
```

NOTE

A `static readonly` field is also advantageous when exposing to other assemblies a value that might change in a later version. For instance, suppose that assembly X exposes a constant as follows:

```
public const decimal ProgramVersion =
2.3;
```

If assembly Y references X and uses this constant, the value `2.3` will be baked into assembly Y when compiled. This means that if X is later recompiled with the constant set to `2.4`, Y will still use the old value of `2.3` *until* Y *is recompiled*. A `static readonly` field prevents this problem.

Another way of looking at this is that any value that might change in the future is not constant by definition; thus, it should not be represented as one.

Constants can also be declared local to a method:

```
void Test()
{
  const double twoPI = 2 * System.Math.PI;
  ...
}
```

Nonlocal constants allow the following modifiers:
Access modifiers
`public internal private protected`
Inheritance modifier
`new`

Methods

A *method* performs an action in a series of statements. A method can

receive *input* data from the caller by specifying *parameters* and *output* data back to the caller by specifying a *return type*. A method can specify a `void` return type, indicating that it doesn't return any value to its caller. A method can also output data back to the caller via `ref`/`out` parameters.

A method's *signature* must be unique within the type. A method's signature comprises its name and parameter types in order (but not the parameter *names*, nor the return type).

Methods allow the following modifiers:

Static modifier
`static`
Access modifiers
`public internal private protected`
Inheritance modifiers
`new virtual abstract override sealed`
Partial method modifier
`partial`
Unmanaged code modifiers
`unsafe extern`
Asynchronous code modifier
`async`

Expression-bodied methods

A method that comprises a single expression, such as

```
int Foo (int x) { return x * 2; }
```

can be written more tersely as an *expression-bodied method*. A fat

arrow replaces the braces and `return` keyword:

```
int Foo (int x) => x * 2;
```

Expression-bodied functions can also have a void return type:

```
void Foo (int x) => Console.WriteLine (x);
```

Local methods

You can define a method within another method:

```
void WriteCubes()
{
  Console.WriteLine (Cube (3));
  Console.WriteLine (Cube (4));
  Console.WriteLine (Cube (5));

  int Cube (int value) => value * value * value;
}
```

The local method (`Cube`, in this case) is visible only to the enclosing method (`WriteCubes`). This simplifies the containing type and instantly signals to anyone looking at the code that `Cube` is used nowhere else. Another benefit of local methods is that they can access the local variables and parameters of the enclosing method. This has a number of consequences, which we describe in detail in “Capturing Outer Variables” in Chapter 4.

Local methods can appear within other function kinds, such as property accessors, constructors, and so on. You can even put local methods inside other local methods, and inside lambda expressions that use a statement block (Chapter 4). Local methods can be iterators (Chapter 4) or asynchronous (Chapter 14).

Static local methods

Adding the `static` modifier to a local method (from C# 8) prevents it from seeing the local variables and parameters of the enclosing method. This helps to reduce coupling and prevents the local method from accidentally referring to variables in the containing method.

Local methods and top-level statements (C# 9)

Any methods that you declare in top-level statements are treated as local methods. This means that (unless marked as `static`) they can access the variables in the top-level statements:

```
int x = 3;
Foo();

void Foo() => Console.WriteLine (x);
```

Overloading methods

> **NOTE**
>
> Local methods cannot be overloaded. This means that methods declared in top-level statements (which are treated as local methods) cannot be overloaded.

A type can *overload* methods (define multiple methods with the same name) as long as the signatures are different. For example, the following methods can all coexist in the same type:

```
void Foo (int x) {...}
void Foo (double x) {...}
void Foo (int x, float y) {...}
void Foo (float x, int y) {...}
```

However, the following pairs of methods cannot coexist in the same type, because the return type and the `params` modifier are not part of a method's signature:

```
void  Foo (int x) {...}
float Foo (int x) {...}           // Compile-time
error

void  Goo (int[] x) {...}
void  Goo (params int[] x) {...}  // Compile-time
error
```

Whether a parameter is pass-by-value or pass-by-reference is also part of the signature. For example, `Foo(int)` can coexist with either `Foo(ref int)` or `Foo(out int)`. However, `Foo(ref int)` and `Foo(out int)` cannot coexist:

```
void Foo (int x) {...}
void Foo (ref int x) {...}     // OK so far
void Foo (out int x) {...}     // Compile-time error
```

Instance Constructors

Constructors run initialization code on a class or struct. A constructor is defined like a method, except that the method name and return type are reduced to the name of the enclosing type:

```
Panda p = new Panda ("Petey");   // Call constructor

public class Panda
{
  string name;                   // Define field
  public Panda (string n)        // Define
constructor
  {
    name = n;                    // Initialization
code (set up field)
  }
}
```

Instance constructors allow the following modifiers:
Access modifiers
`public internal private protected`

Unmanaged code modifiers
`unsafe extern`

Single-statement constructors can also be written as expression-bodied members:

```
public Panda (string n) => name = n;
```

Overloading constructors

A class or struct may overload constructors. To prevent code duplication, one constructor can call another, using the `this` keyword:

```
using System;

public class Wine
{
  public decimal Price;
  public int Year;
  public Wine (decimal price) { Price = price; }
  public Wine (decimal price, int year) : this
(price) { Year = year; }
}
```

When one constructor calls another, the *called constructor* executes first.

You can pass an *expression* into another constructor, as follows:

```
public Wine (decimal price, DateTime year) : this
(price, year.Year) { }
```

The expression itself cannot make use of the `this` reference, for example, to call an instance method. (This is enforced because the object has not been initialized by the constructor at this stage, so any methods that you call on it are likely to fail.) It can, however, call static methods.

Implicit parameterless constructors

For classes, the C# compiler automatically generates a parameterless public constructor if and only if you do not define any constructors. However, as soon as you define at least one constructor, the parameterless constructor is no longer automatically generated.

Constructor and field initialization order

We previously saw that fields can be initialized with default values in their declaration:

```
class Player
{
  int shields = 50;   // Initialized first
  int health = 100;   // Initialized second
}
```

Field initializations occur *before* the constructor is executed, and in the declaration order of the fields.

Nonpublic constructors

Constructors do not need to be public. A common reason to have a nonpublic constructor is to control instance creation via a static method call. The static method could be used to return an object from a pool rather than creating a new object, or to return various subclasses based on input arguments:

```
public class Class1
{
  Class1() {}                                  // Private
constructor
  public static Class1 Create (...)
  {
    // Perform custom logic here to return an
instance of Class1
    ...
  }
}
```

Deconstructors

A deconstructor (also called a *deconstructing method*) acts as an approximate opposite to a constructor: whereas a constructor typically takes a set of values (as parameters) and assigns them to fields, a deconstructor does the reverse and assigns fields back to a set of variables.

A deconstruction method must be called `Deconstruct` and have one or more `out` parameters, such as in the following class:

```
class Rectangle
```

```
{
  public readonly float Width, Height;

  public Rectangle (float width, float height)
  {
    Width = width;
    Height = height;
  }

  public void Deconstruct (out float width, out float
height)
  {
    width = Width;
    height = Height;
  }
}
```

The following special syntax calls the deconstructor:

```
var rect = new Rectangle (3, 4);
(float width, float height) = rect;           //
Deconstruction
Console.WriteLine (width + " " + height);     // 3 4
```

The second line is the deconstructing call. It creates two local variables and then calls the `Deconstruct` method. Our deconstructing call is equivalent to the following:

```
float width, height;
rect.Deconstruct (out width, out height);
```

Or:

```
rect.Deconstruct (out var width, out var height);
```

Deconstructing calls allow implicit typing, so we could shorten our call to this:

```
(var width, var height) = rect;
```

Or simply this:

```
var (width, height) = rect;
```

NOTE

You can use C#'s discard symbol (_) if you're uninterested in one or more variables:

```
var (_, height) = rect;
```

This better indicates your intention than declaring a variable that you never use.

If the variables into which you're deconstructing are already defined, omit the types altogether:

```
float width, height;
(width, height) = rect;
```

This is called a *deconstructing assignment*. You can use a deconstructing assignment to simplify your class's constructor:

```
public Rectangle (float width, float height) =>
  (Width, Height) = (width, height);
```

You can offer the caller a range of deconstruction options by overloading the `Deconstruct` method.

> **NOTE**
>
> The `Deconstruct` method can be an extension method (see "Extension Methods" in Chapter 4). This is a useful trick if you want to deconstruct types that you did not author.

Object Initializers

To simplify object initialization, any accessible fields or properties of an object can be set via an *object initializer* directly after construction. For example, consider the following class:

```
public class Bunny
{
  public string Name;
  public bool LikesCarrots;
  public bool LikesHumans;

  public Bunny () {}
  public Bunny (string n) { Name = n; }
}
```

Using object initializers, you can instantiate `Bunny` objects as follows:

```
// Note parameterless constructors can omit empty
parentheses
Bunny b1 = new Bunny { Name="Bo", LikesCarrots=true,
LikesHumans=false };
Bunny b2 = new Bunny ("Bo")     { LikesCarrots=true,
LikesHumans=false };
```

The code to construct `b1` and `b2` is precisely equivalent to the following:

```
Bunny temp1 = new Bunny();    // temp1 is a compiler-
generated name
temp1.Name = "Bo";
temp1.LikesCarrots = true;
```

```
temp1.LikesHumans = false;
Bunny b1 = temp1;

Bunny temp2 = new Bunny ("Bo");
temp2.LikesCarrots = true;
temp2.LikesHumans = false;
Bunny b2 = temp2;
```

The temporary variables are to ensure that if an exception is thrown during initialization, you can't end up with a half-initialized object.

OBJECT INITIALIZERS VERSUS OPTIONAL PARAMETERS

Instead of using object initializers, we could make `Bunny`'s constructor accept optional parameters:

```
public Bunny (string name,
              bool likesCarrots = false,
              bool likesHumans = false)
{
  Name = name;
  LikesCarrots = likesCarrots;
  LikesHumans = likesHumans;
}
```

This would allow us to construct a `Bunny` as follows:

```
Bunny b1 = new Bunny (name: "Bo",
                      likesCarrots: true);
```

Historically, this approach could be advantageous in that it allowed us to make `Bunny`'s fields (or *properties*, which we'll explain shortly) read-only. Making fields or properties read-only is good practice when there's no valid reason for them to change throughout the life of the object. However, as we'll see soon in our discussion on properties, C# 9's `init` modifier lets us achieve this goal with object initializers.

Optional parameters have two drawbacks. The first is that while their use in constructors allows for read-only types, they don't (easily) allow for *nondestructive mutation*. (We'll cover nondestructive mutation—and the solution to this problem—in "Records (C# 9)" in Chapter 4.)

The second drawback of optional parameters is that when used in public libraries, they hinder backward compatibility. This is because the act of adding an optional parameter at a later date breaks the assembly's *binary compatibility* with existing consumers. (This is particularly important when a library is published on NuGet: the problem becomes intractable when a consumer references packages *A* and *B*, if *A* and *B* each depend on incompatible versions of *L*.)

The difficulty is that each optional parameter value is baked into the *calling site*. In other words, C# translates our constructor call into this:

```
Bunny b1 = new Bunny ("Bo", true, false);
```

This is problematic if we instantiate the `Bunny` class from another assembly and later modify `Bunny` by adding another optional parameter—such as `likesCats`. Unless the referencing assembly is also recompiled, it will continue to call the (now nonexistent) constructor with three parameters and fail at runtime. (A subtler problem is that if we changed the value of one of the optional parameters, callers in other assemblies would continue to use the old optional value until they were recompiled.)

The this Reference

The `this` reference refers to the instance itself. In the following example, the `Marry` method uses `this` to set the `partner`'s `mate` field:

```
public class Panda
{
  public Panda Mate;

  public void Marry (Panda partner)
  {
    Mate = partner;
    partner.Mate = this;
  }
}
```

The `this` reference also disambiguates a local variable or parameter from a field. For example:

```
public class Test
{
  string name;
  public Test (string name) { this.name = name; }
}
```

The `this` reference is valid only within nonstatic members of a class or struct.

Properties

Properties look like fields from the outside, but internally they contain logic, like methods do. For example, you can't tell by looking at the following code whether `CurrentPrice` is a field or a property:

```
Stock msft = new Stock();
msft.CurrentPrice = 30;
msft.CurrentPrice -= 3;
Console.WriteLine (msft.CurrentPrice);
```

A property is declared like a field but with a `get`/`set` block added. Here's how to implement `CurrentPrice` as a property:

```
public class Stock
{
  decimal currentPrice;            // The private
"backing" field

  public decimal CurrentPrice      // The public
property
  {
    get { return currentPrice; }
    set { currentPrice = value; }
  }
}
```

`get` and `set` denote property *accessors*. The `get` accessor runs when the property is read. It must return a value of the property's type. The `set` accessor runs when the property is assigned. It has an

implicit parameter named `value` of the property's type that you typically assign to a private field (in this case, `currentPrice`).

Although properties are accessed in the same way as fields, they differ in that they give the implementer complete control over getting and setting its value. This control enables the implementer to choose whatever internal representation is needed without exposing the internal details to the user of the property. In this example, the `set` method could throw an exception if `value` was outside a valid range of values.

> **NOTE**
>
> Throughout this book, we use public fields extensively to keep the examples free of distraction. In a real application, you would typically favor public properties over public fields in order to promote encapsulation.

Properties allow the following modifiers:

Static modifier
`static`

Access modifiers
`public internal private protected`

Inheritance modifiers
`new virtual abstract override sealed`

Unmanaged code modifiers
`unsafe extern`

Read-only and calculated properties

A property is read-only if it specifies only a `get` accessor, and it is write-only if it specifies only a `set` accessor. Write-only properties are rarely used.

A property typically has a dedicated backing field to store the underlying data. However, a property can also be computed from other data:

```
decimal currentPrice, sharesOwned;

public decimal Worth
{
  get { return currentPrice * sharesOwned; }
}
```

Expression-bodied properties

You can declare a read-only property, such as the one in the preceding example, more tersely as an *expression-bodied property*. A fat arrow replaces all the braces and the `get` and `return` keywords:

```
public decimal Worth => currentPrice * sharesOwned;
```

With a little extra syntax, `set` accessors can also be expression-bodied:

```
public decimal Worth
{
  get => currentPrice * sharesOwned;
  set => sharesOwned = value / currentPrice;
}
```

Automatic properties

The most common implementation for a property is a getter and/or setter that simply reads and writes to a private field of the same type as the property. An *automatic property* declaration instructs the compiler to provide this implementation. We can improve the first example in this section by declaring `CurrentPrice` as an automatic property:

```
public class Stock
{
  ...
  public decimal CurrentPrice { get; set; }
}
```

The compiler automatically generates a private backing field of a compiler-generated name that cannot be referred to. The `set` accessor can be marked `private` or `protected` if you want to expose the property as read-only to other types. Automatic properties were introduced in C# 3.0.

Property initializers

You can add a *property initializer* to automatic properties, just as with fields:

```
public decimal CurrentPrice { get; set; } = 123;
```

This gives `CurrentPrice` an initial value of `123`. Properties with an initializer can be read-only:

```
public int Maximum { get; } = 999;
```

Just as with read-only fields, read-only automatic properties can also be assigned in the type's constructor. This is useful in creating *immutable* (read-only) types.

get and set accessibility

The `get` and `set` accessors can have different access levels. The typical use case for this is to have a `public` property with an `internal` or `private` access modifier on the setter:

```
public class Foo
{
  private decimal x;
  public decimal X
  {
    get         { return x;  }
    private set { x = Math.Round (value, 2); }
  }
}
```

Notice that you declare the property itself with the more permissive

access level (`public`, in this case) and add the modifier to the accessor you want to be *less* accessible.

Init-only setters (C# 9)

From C# 9, you can declare a property accessor with `init` instead of `set`:

```
public class Note
{
  public int Pitch    { get; init; } = 20;   //
"Init-only" property
  public int Duration { get; init; } = 100;  //
"Init-only" property
}
```

These *init-only* properties act like read-only properties, except that they can also be set via an object initializer:

```
var note = new Note { Pitch = 50 };
```

After that, the property cannot be altered:

```
note.Pitch = 200;  // Error – init-only setter!
```

Init-only properties cannot even be set from inside their class, except via their property initializer, the constructor, or another init-only

accessor.

The alternative to init-only properties is to have read-only properties that you populate via a constructor:

```
public class Note
{
  public int Pitch    { get; }
  public int Duration { get; }

  public Note (int pitch = 20, int duration = 100)
  {
    Pitch = pitch; Duration = duration;
  }
}
```

Should the class be part of a public library, this approach makes versioning difficult, in that adding an optional parameter to the constructor at a later date breaks binary compatibility with consumers (whereas adding a new init-only property breaks nothing).

> **NOTE**
>
> Init-only properties have another significant advantage, which is that they allow for nondestructive mutation when used in conjunction with records (see “Records (C# 9)”).

Just as with ordinary `set` accessors, init-only accessors can also provide an implementation:

```
public class Note
{
  readonly int _pitch;
  public int Pitch { get => _pitch; init => _pitch =
value; }
  ...
```

Notice that the `_pitch` field is read-only: init-only setters are permitted to modify `readonly` fields in their own class. (Without this feature, `_pitch` would need to be writable, and the class would fail at being internally immutable.)

> **NOTE**
>
> Changing a property's accessor from `init` to `set` (or vice versa) is a *binary breaking change*: anyone that references your assembly will need to recompile their assembly.
>
> This should not be an issue when creating wholly immutable types, in that your type will never require properties with a (writable) `set` accessor.

CLR property implementation

C# property accessors internally compile to methods called `get_XXX` and `set_XXX`:

```
public decimal get_CurrentPrice {...}
public void set_CurrentPrice (decimal value) {...}
```

An `init` accessor is processed like a `set` accessor, but with an extra flag encoded into the `set` accessor's "modreq" metadata (see "Init-only properties" in Chapter 18).

Simple nonvirtual property accessors are *inlined* by the Just-In-Time (JIT) compiler, eliminating any performance difference between accessing a property and a field. Inlining is an optimization in which a method call is replaced with the body of that method.

Indexers

Indexers provide a natural syntax for accessing elements in a class or struct that encapsulate a list or dictionary of values. Indexers are similar to properties but are accessed via an index argument rather than a property name. The `string` class has an indexer that lets you access each of its `char` values via an `int` index:

```
string s = "hello";
Console.WriteLine (s[0]); // 'h'
Console.WriteLine (s[3]); // 'l'
```

The syntax for using indexers is like that for using arrays, except that the index argument(s) can be of any type(s).

Indexers have the same modifiers as properties (see “Properties”) and can be called null-conditionally by inserting a question mark before the square bracket (see “Null Operators” in Chapter 2):

```
string s = null;
Console.WriteLine (s?[0]);  // Writes nothing; no
error
```

Implementing an indexer

To write an indexer, define a property called `this`, specifying the arguments in square brackets:

```
class Sentence
{
  string[] words = "The quick brown fox".Split();

  public string this [int wordNum]      // indexer
  {
    get { return words [wordNum];  }
    set { words [wordNum] = value; }
  }
}
```

Here’s how we could use this indexer:

```
Sentence s = new Sentence();
Console.WriteLine (s[3]);       // fox
```

```
s[3] = "kangaroo";
Console.WriteLine (s[3]);       // kangaroo
```

A type can declare multiple indexers, each with parameters of different types. An indexer can also take more than one parameter:

```
public string this [int arg1, string arg2]
{
  get { ... }  set { ... }
}
```

If you omit the `set` accessor, an indexer becomes read-only, and you can use expression-bodied syntax to shorten its definition:

```
public string this [int wordNum] => words [wordNum];
```

CLR indexer implementation

Indexers internally compile to methods called `get_Item` and `set_Item`, as follows:

```
public string get_Item (int wordNum) {...}
public void set_Item (int wordNum, string value)
{...}
```

Using indices and ranges with indexers

You can support indices and ranges (see “Indices and Ranges”) in your own classes by defining an indexer with a parameter type of `Index` or `Range`. We could extend our previous example, by adding the following indexers to the `Sentence` class:

```
  public string this [Index index] => words [index];
  public string[] this [Range range] => words
[range];
```

This then enables the following:

```
Sentence s = new Sentence();
Console.WriteLine (s [^1]);          // fox
string[] firstTwoWords = s [..2];    // (The, quick)
```

Static Constructors

A static constructor executes once per *type* rather than once per *instance*. A type can define only one static constructor, and it must be parameterless and have the same name as the type:

```
class Test
{
  static Test() { Console.WriteLine ("Type
Initialized"); }
}
```

The runtime automatically invokes a static constructor just prior to the type being used. Two things trigger this:

- Instantiating the type
- Accessing a static member in the type

The only modifiers allowed by static constructors are `unsafe` and `extern`.

> **WARNING**
>
> If a static constructor throws an unhandled exception (Chapter 4), that type becomes *unusable* for the life of the application.

> **NOTE**
>
> From C# 9, you can also define *module initializers*, which execute once per assembly (when the assembly is first loaded). To define a module initializer, write a static void method and then apply the `[ModuleInitializer]` attribute to that method:

```
[System.Runtime.CompilerServices.Modu
leInitializer]
internal static void InitAssembly()
```

```
{
  ...
}
```

Static constructors and field initialization order

Static field initializers run just *before* the static constructor is called. If a type has no static constructor, static field initializers will execute just prior to the type being used—or *anytime earlier* at the whim of the runtime.

Static field initializers run in the order in which the fields are declared. The following example illustrates this. `X` is initialized to `0`, and `Y` is initialized to `3`:

```
class Foo
{
  public static int X = Y;    // 0
  public static int Y = 3;    // 3
}
```

If we swap the two field initializers around, both fields are initialized to 3. The next example prints 0 followed by 3 because the field initializer that instantiates a `Foo` executes before `X` is initialized to `3`:

```
Console.WriteLine (Foo.X);    // 3
```

```
class Foo
{
  public static Foo Instance = new Foo();
  public static int X = 3;

  Foo() => Console.WriteLine (X);    // 0
}
```

If we swap the two lines in boldface, the example prints 3 followed by 3.

Static Classes

A class can be marked `static`, indicating that it must be composed solely of static members and cannot be subclassed. The `System.Console` and `System.Math` classes are good examples of static classes.

Finalizers

Finalizers are class-only methods that execute before the garbage collector reclaims the memory for an unreferenced object. The syntax for a finalizer is the name of the class prefixed with the ~ symbol:

```
class Class1
{
  ~Class1()
```

```
  {
    ...
  }
}
```

This is actually C# syntax for overriding `Object`'s `Finalize` method, and the compiler expands it into the following method declaration:

```
protected override void Finalize()
{
  ...
  base.Finalize();
}
```

We discuss garbage collection and finalizers fully in Chapter 12.

Finalizers allow the following modifier:
Unmanaged code modifier
`unsafe`

You can write single-statement finalizers using expression-bodied syntax:

```
~Class1() => Console.WriteLine ("Finalizing");
```

Partial Types and Methods

Partial types allow a type definition to be split—typically across

multiple files. A common scenario is for a partial class to be autogenerated from some other source (such as a Visual Studio template or designer), and for that class to be augmented with additional hand-authored methods:

```
// PaymentFormGen.cs - auto-generated
partial class PaymentForm { ... }

// PaymentForm.cs - hand-authored
partial class PaymentForm { ... }
```

Each participant must have the `partial` declaration; the following is illegal:

```
partial class PaymentForm {}
class PaymentForm {}
```

Participants cannot have conflicting members. A constructor with the same parameters, for instance, cannot be repeated. Partial types are resolved entirely by the compiler, which means that each participant must be available at compile time and must reside in the same assembly.

You can specify a base class on one or more partial class declarations, as long as the base class, if specified, is the same. In addition, each participant can independently specify interfaces to implement. We cover base classes and interfaces in “Inheritance” and “Interfaces”.

The compiler makes no guarantees with regard to field initialization

order between partial type declarations.

Partial methods

A partial type can contain *partial methods*. These let an autogenerated partial type provide customizable hooks for manual authoring. For example:

```
partial class PaymentForm    // In auto-generated
file
{
  ...
  partial void ValidatePayment (decimal amount);
}

partial class PaymentForm    // In hand-authored file
{
  ...
  partial void ValidatePayment (decimal amount)
  {
    if (amount > 100)
      ...
  }
}
```

A partial method consists of two parts: a *definition* and an *implementation*. The definition is typically written by a code generator, and the implementation is typically manually authored. If an implementation is not provided, the definition of the partial method is compiled away (as is the code that calls it). This allows autogenerated code to be liberal in providing hooks without having to worry about bloat. Partial methods must be `void` and are implicitly `private`. They cannot include `out` parameters.

Extended partial methods (C# 9)

Extended partial methods are designed for the reverse code generation scenario, where a programmer defines hooks that a code generator implements. An example of where this might occur is with *source generators*, a Roslyn feature that lets you feed the compiler an assembly that automatically generates portions of your code.

A partial method declaration is *extended* if it begins with an accessibility modifier:

```
public partial class Test
{
  public partial void M1();    // Extended partial
method
  private partial void M2();   // Extended partial
method
}
```

The presence of the accessibility modifier doesn't just affect accessibility: it tells the compiler to treat the declaration differently.

Extended partial methods *must* have implementations; they do not melt away if unimplemented. In this example, both `M1` and `M2` must have implementations because they each specify accessibility modifiers (`public` and `private`).

Because they cannot melt away, extended partial methods can return any type and can include `out` parameters:

```
public partial class Test
{
```

```
  public partial bool IsValid (string identifier);
  internal partial bool TryParse (string number, out
int result);
}
```

The nameof operator

The `nameof` operator returns the name of any symbol (type, member, variable, and so on) as a string:

```
int count = 123;
string name = nameof (count);       // name is
"count"
```

Its advantage over simply specifying a string is that of static type checking. Tools such as Visual Studio can understand the symbol reference, so if you rename the symbol in question, all of its references will be renamed, too.

To specify the name of a type member such as a field or property, include the type as well. This works with both static and instance members:

```
string name = nameof (StringBuilder.Length);
```

This evaluates to `Length`. To return `StringBuilder.Length`, you would do this:

```
nameof (StringBuilder) + "." + nameof
(StringBuilder.Length);
```

Inheritance

A class can *inherit* from another class to extend or customize the original class. Inheriting from a class lets you reuse the functionality in that class instead of building it from scratch. A class can inherit from only a single class but can itself be inherited by many classes, thus forming a class hierarchy. In this example, we begin by defining a class called `Asset`:

```
public class Asset
{
  public string Name;
}
```

Next, we define classes called `Stock` and `House`, which will inherit from `Asset`. `Stock` and `House` get everything an `Asset` has, plus any additional members that they define:

```
public class Stock : Asset   // inherits from Asset
{
  public long SharesOwned;
}

public class House : Asset   // inherits from Asset
{
  public decimal Mortgage;
}
```

Here’s how we can use these classes:

```
Stock msft = new Stock { Name="MSFT",
                         SharesOwned=1000 };

Console.WriteLine (msft.Name);         // MSFT
Console.WriteLine (msft.SharesOwned);  // 1000

House mansion = new House { Name="Mansion",
                            Mortgage=250000 };

Console.WriteLine (mansion.Name);      // Mansion
Console.WriteLine (mansion.Mortgage);  // 250000
```

The *derived classes*, `Stock` and `House`, inherit the `Name` property from the *base class*, `Asset`.

> **NOTE**
>
> A derived class is also called a *subclass*.
>
> A base class is also called a *superclass*.

Polymorphism

References are *polymorphic*. This means a variable of type *x* can refer to an object that subclasses *x*. For instance, consider the following method:

```
public static void Display (Asset asset)
{
  System.Console.WriteLine (asset.Name);
}
```

This method can display both a `Stock` and a `House` because they are both `Assets`:

```
Stock msft    = new Stock ... ;
House mansion = new House ... ;

Display (msft);
Display (mansion);
```

Polymorphism works on the basis that subclasses (`Stock` and `House`) have all the features of their base class (`Asset`). The converse, however, is not true. If `Display` was modified to accept a `House`, you could not pass in an `Asset`:

```
Display (new Asset());     // Compile-time error

public static void Display (House house)         //
Will not accept Asset
{
  System.Console.WriteLine (house.Mortgage);
}
```

Casting and Reference Conversions

An object reference can be:

- Implicitly *upcast* to a base class reference
- Explicitly *downcast* to a subclass reference

Upcasting and downcasting between compatible reference types performs *reference conversions*: a new reference is (logically) created that points to the *same* object. An upcast always succeeds; a downcast succeeds only if the object is suitably typed.

Upcasting

An upcast operation creates a base class reference from a subclass reference:

```
Stock msft = new Stock();
Asset a = msft;              // Upcast
```

After the upcast, variable `a` still references the same `Stock` object as variable `msft`. The object being referenced is not itself altered or converted:

```
Console.WriteLine (a == msft);        // True
```

Although `a` and `msft` refer to the identical object, `a` has a more

restrictive view on that object:

```
Console.WriteLine (a.Name);           // OK
Console.WriteLine (a.SharesOwned);    // Compile-time
error
```

The last line generates a compile-time error because the variable `a` is of type `Asset`, even though it refers to an object of type `Stock`. To get to its `SharesOwned` field, you must *downcast* the `Asset` to a `Stock`.

Downcasting

A downcast operation creates a subclass reference from a base class reference:

```
Stock msft = new Stock();
Asset a = msft;                       // Upcast
Stock s = (Stock)a;                   // Downcast
Console.WriteLine (s.SharesOwned);    // <No error>
Console.WriteLine (s == a);           // True
Console.WriteLine (s == msft);        // True
```

As with an upcast, only references are affected—not the underlying object. A downcast requires an explicit cast because it can potentially fail at runtime:

```
House h = new House();
```

```
Asset a = h;                  // Upcast always succeeds
Stock s = (Stock)a;           // Downcast fails: a is
not a Stock
```

If a downcast fails, an `InvalidCastException` is thrown. This is an example of *runtime type checking* (we elaborate on this concept in "Static and Runtime Type Checking").

The as operator

The `as` operator performs a downcast that evaluates to `null` (rather than throwing an exception) if the downcast fails:

```
Asset a = new Asset();
Stock s = a as Stock;        // s is null; no
exception thrown
```

This is useful when you're going to subsequently test whether the result is `null`:

```
if (s != null) Console.WriteLine (s.SharesOwned);
```

> **NOTE**
>
> Without such a test, a cast is advantageous, because if it fails, a more helpful exception is thrown. We can illustrate by comparing the following two lines of code:

```
int shares = ((Stock)a).SharesOwned;
// Approach #1
int shares = (a as
Stock).SharesOwned;  // Approach #2
```

If `a` is not a `Stock`, the first line throws an `InvalidCastException`, which is an accurate description of what went wrong. The second line throws a `NullReferenceException`, which is ambiguous. Was `a` not a `Stock`, or was `a` null?

Another way of looking at it is that with the cast operator, you're saying to the compiler: "I'm *certain* of a value's type; if I'm wrong, there's a bug in my code, so throw an exception!" With the `as` operator, on the other hand, you're uncertain of its type and want to branch according to the outcome at runtime.

The `as` operator cannot perform *custom conversions* (see "Operator Overloading" in Chapter 4), and it cannot do numeric conversions:

```
long x = 3 as long;    // Compile-time error
```

NOTE

The `as` and cast operators will also perform upcasts, although this is not terribly useful because an implicit conversion will do the job.

The is operator

The `is` operator tests whether a variable matches a *pattern*. C# supports several kinds of patterns, the most important being a *type pattern*, where a type name follows the `is` keyword.

In this context, the `is` operator tests whether a reference conversion would succeed—in other words, whether an object derives from a specified class (or implements an interface). It is often used to test before downcasting:

```
if (a is Stock)
  Console.WriteLine (((Stock)a).SharesOwned);
```

The `is` operator also evaluates to true if an *unboxing conversion* would succeed (see "The object Type"). However, it does not consider custom or numeric conversions.

> **NOTE**
>
> The `is` operator works with many other patterns introduced in recent versions of C#. For a full discussion, see "Patterns" in Chapter 4.

Introducing a pattern variable

You can introduce a variable while using the `is` operator:

```
if (a is Stock s)
  Console.WriteLine (s.SharesOwned);
```

This is equivalent to the following:

```
Stock s;
if (a is Stock)
{
  s = (Stock) a;
  Console.WriteLine (s.SharesOwned);
}
```

The variable that you introduce is available for “immediate” consumption, so the following is legal:

```
if (a is Stock s && s.SharesOwned > 100000)
  Console.WriteLine ("Wealthy");
```

And it remains in scope outside the `is` expression, allowing this:

```
if (a is Stock s && s.SharesOwned > 100000)
  Console.WriteLine ("Wealthy");
Else
  s = new Stock();   // s is in scope

Console.WriteLine (s.SharesOwned);  // Still in scope
```

Virtual Function Members

A function marked as `virtual` can be *overridden* by subclasses wanting to provide a specialized implementation. Methods, properties, indexers, and events can all be declared `virtual`:

```
public class Asset
{
  public string Name;
  public virtual decimal Liability => 0;   //
Expression-bodied property
}
```

(`Liability => 0` is a shortcut for `{ get { return 0; } }`. For more details on this syntax, see “Expression-bodied properties”.)

A subclass overrides a virtual method by applying the `override` modifier:

```
public class Stock : Asset
{
  public long SharesOwned;
}
```

```
public class House : Asset
{
  public decimal Mortgage;
  public override decimal Liability => Mortgage;
}
```

By default, the `Liability` of an `Asset` is `0`. A `Stock` does not need to specialize this behavior. However, the `House` specializes the `Liability` property to return the value of the `Mortgage`:

```
House mansion = new House { Name="McMansion",
Mortgage=250000 };
Asset a = mansion;
Console.WriteLine (mansion.Liability);  // 250000
Console.WriteLine (a.Liability);        // 250000
```

The signatures, return types, and accessibility of the virtual and overridden methods must be identical. An overridden method can call its base class implementation via the `base` keyword (we cover this in “The base Keyword”).

WARNING

Calling virtual methods from a constructor is potentially dangerous because authors of subclasses are unlikely to know, when overriding the method, that they are working with a partially initialized object. In other words, the overriding method might end up accessing methods or properties that

rely on fields not yet initialized by the constructor.

Covariant return types (C# 9)

From C# 9, you can override a method (or property `get` accessor) such that it returns a *more derived* (subclassed) type. For example:

```
public class Asset
{
  public string Name;
  public virtual Asset Clone() => new Asset { Name =
Name };
}

public class House : Asset
{
  public decimal Mortgage;
  public override House Clone() => new House
                                   { Name = Name,
Mortgage = Mortgage };
}
```

This is permitted because it does not break the contract that `Clone` must return an `Asset`: it returns a `House`, which *is* an `Asset` (and more).

Prior to C# 9, you had to override methods with the identical return type:

```
public override Asset Clone() => new House { ... }
```

This still does the job, because the overridden `Clone` method instantiates a `House` rather than an `Asset`. However, to treat the returned object as a `House`, you must then perform a downcast:

```
House mansion1 = new House { Name="McMansion",
Mortgage=250000 };
House mansion2 = (House) mansion1.Clone();
```

Abstract Classes and Abstract Members

A class declared as *abstract* can never be instantiated. Instead, only its concrete *subclasses* can be instantiated.

Abstract classes are able to define *abstract members*. Abstract members are like virtual members except that they don't provide a default implementation. That implementation must be provided by the subclass unless that subclass is also declared abstract:

```
public abstract class Asset
{
  // Note empty implementation
  public abstract decimal NetValue { get; }
}

public class Stock : Asset
{
  public long SharesOwned;
  public decimal CurrentPrice;
```

```
  // Override like a virtual method.
  public override decimal NetValue => CurrentPrice *
SharesOwned;
}
```

Hiding Inherited Members

A base class and a subclass can define identical members. For example:

```
public class A      { public int Counter = 1; }
public class B : A  { public int Counter = 2; }
```

The `Counter` field in class `B` is said to *hide* the `Counter` field in class `A`. Usually, this happens by accident, when a member is added to the base type *after* an identical member was added to the subtype. For this reason, the compiler generates a warning and then resolves the ambiguity as follows:

- References to `A` (at compile time) bind to `A.Counter`
- References to `B` (at compile time) bind to `B.Counter`

Occasionally, you want to hide a member deliberately, in which case you can apply the `new` modifier to the member in the subclass. The `new` modifier *does nothing more than suppress the compiler warning that would otherwise result*:

```
public class A     { public     int Counter = 1; }
```

```
public class B : A { public new int Counter = 2; }
```

The new modifier communicates your intent to the compiler—and other programmers—that the duplicate member is not an accident.

> **NOTE**
>
> C# overloads the new keyword to have independent meanings in different contexts. Specifically, the new *operator* is different from the new *member modifier*.

new versus override

Consider the following class hierarchy:

```
public class BaseClass
{
  public virtual void Foo()  { Console.WriteLine
("BaseClass.Foo"); }
}

public class Overrider : BaseClass
{
  public override void Foo() { Console.WriteLine
("Overrider.Foo"); }
}

public class Hider : BaseClass
```

```
{
  public new void Foo()      { Console.WriteLine
("Hider.Foo"); }
}
```

The differences in behavior between `Overrider` and `Hider` are demonstrated in the following code:

```
Overrider over = new Overrider();
BaseClass b1 = over;
over.Foo();                          // Overrider.Foo
b1.Foo();                            // Overrider.Foo

Hider h = new Hider();
BaseClass b2 = h;
h.Foo();                            // Hider.Foo
b2.Foo();                           // BaseClass.Foo
```

Sealing Functions and Classes

An overridden function member can *seal* its implementation with the `sealed` keyword to prevent it from being overridden by further subclasses. In our earlier virtual function member example, we could have sealed `House`'s implementation of `Liability`, preventing a class that derives from `House` from overriding `Liability`, as follows:

```
public sealed override decimal Liability { get {
return Mortgage; } }
```

You can also apply the `sealed` modifier to the class itself, to prevent subclassing. Sealing a class is more common than sealing a function member.

Although you can seal a function member against overriding, you can't seal a member against being *hidden*.

The base Keyword

The `base` keyword is similar to the `this` keyword. It serves two essential purposes:

- Accessing an overridden function member from the subclass
- Calling a base-class constructor (see the next section)

In this example, `House` uses the `base` keyword to access `Asset`'s implementation of `Liability`:

```
public class House : Asset
{
  ...
  public override decimal Liability => base.Liability
+ Mortgage;
}
```

With the `base` keyword, we access `Asset`'s `Liability` property *nonvirtually*. This means that we will always access `Asset`'s version of this property—regardless of the instance's actual runtime type.

The same approach works if `Liability` is *hidden* rather than *overridden*. (You can also access hidden members by casting to the base class before invoking the function.)

Constructors and Inheritance

A subclass must declare its own constructors. The base class's constructors are *accessible* to the derived class but are never automatically *inherited*. For example, if we define `Baseclass` and `Subclass` as follows

```
public class Baseclass
{
  public int X;
  public Baseclass () { }
  public Baseclass (int x) { this.X = x; }
}

public class Subclass : Baseclass { }
```

the following is illegal:

```
Subclass s = new Subclass (123);
```

`Subclass` must hence "redefine" any constructors it wants to expose. In doing so, however, it can call any of the base class's constructors via the `base` keyword:

```
public class Subclass : Baseclass
{
  public Subclass (int x) : base (x) { }
}
```

The `base` keyword works rather like the `this` keyword except that it calls a constructor in the base class.

Base-class constructors always execute first; this ensures that *base* initialization occurs before *specialized* initialization.

Implicit calling of the parameterless base-class constructor

If a constructor in a subclass omits the `base` keyword, the base type's *parameterless* constructor is implicitly called:

```
public class BaseClass
{
  public int X;
  public BaseClass() { X = 1; }
}

public class Subclass : BaseClass
{
  public Subclass() { Console.WriteLine (X); }  // 1
}
```

If the base class has no accessible parameterless constructor, subclasses are forced to use the `base` keyword in their constructors.

Constructor and field initialization order

When an object is instantiated, initialization takes place in the following order:

1. From subclass to base class:
    a. Fields are initialized.
    b. Arguments to base-class constructor calls are evaluated.
2. From base class to subclass:
    a. Constructor bodies execute.

The following code demonstrates this:

```
public class B
{
  int x = 1;          // Executes 3rd
  public B (int x)
  {
    ...               // Executes 4th
  }
}
public class D : B
{
  int y = 1;          // Executes 1st
  public D (int x)
    : base (x + 1)    // Executes 2nd
  {
     ...              // Executes 5th
  }
}
```

Overloading and Resolution

Inheritance has an interesting impact on method overloading. Consider the following two overloads:

```
static void Foo (Asset a) { }
static void Foo (House h) { }
```

When an overload is called, the most specific type has precedence:

```
House h = new House (...);
Foo(h);                       // Calls Foo(House)
```

The particular overload to call is determined statically (at compile time) rather than at runtime. The following code calls `Foo(Asset)`, even though the runtime type of `a` is `House`:

```
Asset a = new House (...);
Foo(a);                       // Calls Foo(Asset)
```

> **NOTE**
>
> If you cast `Asset` to `dynamic` (Chapter 4), the decision as to which overload to call is deferred until runtime and is then based on the object's actual type:

```
Asset a = new House (...);
Foo ((dynamic)a);   // Calls
Foo(House)
```

The object Type

`object` (`System.Object`) is the ultimate base class for all types. Any type can be upcast to `object`.

To illustrate how this is useful, consider a general-purpose *stack*. A stack is a data structure based on the principle of *LIFO*—"last in, first out." A stack has two operations: *push* an object on the stack, and *pop* an object off the stack. Here is a simple implementation that can hold up to 10 objects:

```
public class Stack
{
  int position;
  object[] data = new object[10];
  public void Push (object obj)   { data[position++]
= obj;  }
  public object Pop()             { return data[--
position]; }
}
```

Because `Stack` works with the object type, we can `Push` and `Pop`

instances of *any type* to and from the `Stack`:

```
Stack stack = new Stack();
stack.Push ("sausage");
string s = (string) stack.Pop();   // Downcast, so
explicit cast is needed

Console.WriteLine (s);             // sausage
```

`object` is a reference type, by virtue of being a class. Despite this, value types, such as `int`, can also be cast to and from `object` and so be added to our stack. This feature of C# is called *type unification* and is demonstrated here:

```
stack.Push (3);
int three = (int) stack.Pop();
```

When you cast between a value type and `object`, the CLR must perform some special work to bridge the difference in semantics between value and reference types. This process is called *boxing* and *unboxing*.

> **NOTE**
>
> In “Generics”, we describe how to improve our `Stack` class to better handle stacks with same-typed elements.

Boxing and Unboxing

Boxing is the act of converting a value-type instance to a reference-type instance. The reference type can be either the `object` class or an interface (which we visit later in the chapter).[1] In this example, we box an `int` into an object:

```
int x = 9;
object obj = x;           // Box the int
```

Unboxing reverses the operation by casting the object back to the original value type:

```
int y = (int)obj;         // Unbox the int
```

Unboxing requires an explicit cast. The runtime checks that the stated value type matches the actual object type and throws an `InvalidCastException` if the check fails. For instance, the following throws an exception because `long` does not exactly match `int`:

```
object obj = 9;           // 9 is inferred to be of
type int
long x = (long) obj;      // InvalidCastException
```

The following succeeds, however:

```
object obj = 9;
long x = (int) obj;
```

As does this:

```
object obj = 3.5;              // 3.5 is inferred to
be of type double
int x = (int) (double) obj;    // x is now 3
```

In the last example, `(double)` performs an *unboxing*, and then `(int)` performs a *numeric conversion*.

> **NOTE**
>
> Boxing conversions are crucial in providing a unified type system. The system is not perfect, however: we'll see in "Generics" that variance with arrays and generics supports only *reference conversions* and not *boxing conversions*:
>
> ```
> object[] a1 = new string[3];   //
> Legal
> object[] a2 = new int[3];      //
> Error
> ```

Copying semantics of boxing and unboxing

Boxing *copies* the value-type instance into the new object, and unboxing *copies* the contents of the object back into a value-type instance. In the following example, changing the value of `i` doesn't change its previously boxed copy:

```
int i = 3;
object boxed = i;
i = 5;
Console.WriteLine (boxed);    // 3
```

Static and Runtime Type Checking

C# programs are type-checked both statically (at compile time) and at runtime (by the CLR).

Static type checking enables the compiler to verify the correctness of your program without running it. The following code will fail because the compiler enforces static typing:

```
int x = "5";
```

Runtime type checking is performed by the CLR when you downcast

via a reference conversion or unboxing:

```
object y = "5";
int z = (int) y;          // Runtime error, downcast
failed
```

Runtime type checking is possible because each object on the heap internally stores a little type token. You can retrieve this token by calling the `GetType` method of `object`.

The GetType Method and typeof Operator

All types in C# are represented at runtime with an instance of `System.Type`. There are two basic ways to get a `System.Type` object:

- Call `GetType` on the instance
- Use the `typeof` operator on a type name

`GetType` is evaluated at runtime; `typeof` is evaluated statically at compile time (when generic type parameters are involved, it's resolved by the JIT compiler).

`System.Type` has properties for such things as the type's name, assembly, base type, and so on:

```
Point p = new Point();
Console.WriteLine (p.GetType().Name);             //
```

```
Point
Console.WriteLine (typeof (Point).Name);          //
Point
Console.WriteLine (p.GetType() == typeof(Point)); //
True
Console.WriteLine (p.X.GetType().Name);           //
Int32
Console.WriteLine (p.Y.GetType().FullName);       //
System.Int32

public class Point { public int X, Y; }
```

`System.Type` also has methods that act as a gateway to the runtime's reflection model, described in Chapter 18.

The ToString Method

The `ToString` method returns the default textual representation of a type instance. This method is overridden by all built-in types. Here is an example of using the `int` type's `ToString` method:

```
int x = 1;
string s = x.ToString();     // s is "1"
```

You can override the `ToString` method on custom types as follows:

```
Panda p = new Panda { Name = "Petey" };
Console.WriteLine (p);   // Petey
```

```
public class Panda
{
  public string Name;
  public override string ToString() => Name;
}
```

If you don’t override `ToString`, the method returns the type name.

> **NOTE**
>
> When you call an *overridden* `object` member such as `ToString` directly on a value type, boxing doesn’t occur. Boxing then occurs only if you cast:
>
> ```
> int x = 1;
> string s1 = x.ToString();    //
> Calling on nonboxed value
> object box = x;
> string s2 = box.ToString();  //
> Calling on boxed value
> ```

Object Member Listing

Here are all the members of `object`:

```
public class Object
{
  public Object();

  public extern Type GetType();

  public virtual bool Equals (object obj);
  public static bool Equals  (object objA, object
objB);
  public static bool ReferenceEquals (object objA,
object objB);

  public virtual int GetHashCode();

  public virtual string ToString();

  protected virtual void Finalize();
  protected extern object MemberwiseClone();
}
```

We describe the `Equals`, `ReferenceEquals`, and `GetHashCode` methods in "Equality Comparison" in Chapter 6.

Structs

A *struct* is similar to a class, with the following key differences:

- A struct is a value type, whereas a class is a reference type.
- A struct does not support inheritance (other than implicitly deriving from `object`, or more precisely, `System.ValueType`).

A struct can have all of the members that a class can except the following:

- A parameterless constructor
- Field initializers
- A finalizer
- Virtual or protected members

A struct is appropriate when value-type semantics are desirable. Good examples of structs are numeric types, where it is more natural for assignment to copy a value rather than a reference. Because a struct is a value type, each instance does not require instantiation of an object on the heap; this results in useful savings when creating many instances of a type. For instance, creating an array of value type requires only a single heap allocation.

Because structs are value types, an instance cannot be null. The default value for a struct is an empty instance, with all fields empty (set to their default values).

Struct Construction Semantics

The construction semantics of a struct are as follows:

- A parameterless constructor that you can’t override implicitly exists. This performs a bitwise-zeroing of its fields (setting them to their default values).
- When you define a struct constructor, you must explicitly

assign every field.

(And you can’t have field initializers.) Here is an example of declaring and calling struct constructors:

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

...
Point p1 = new Point ();       // p1.x and p1.y will
be 0
Point p2 = new Point (1, 1);   // p2.x and p2.y will
be 1
```

The `default` keyword, when applied to a struct, does the same job as its implicit parameterless constructor:

```
Point p1 = default;
```

This can serve as a convenient shortcut when calling methods:

```
void Foo (Point p) { ... }
...
Foo (default);    // Equivalent to Foo (new Point());
```

The next example generates three compile-time errors:

```
public struct Point
{
  int x = 1;                          // Illegal:
field initializer
  int y;
  public Point() {}                   // Illegal:
parameterless constructor
  public Point (int x) {this.x = x;}  // Illegal:
must assign field y
}
```

Changing `struct` to `class` makes this example legal.

Read-Only Structs and Functions

You can apply the `readonly` modifier to a struct to enforce that all fields are `readonly`; this aids in declaring intent as well as affording the compiler more optimization freedom:

```
readonly struct Point
{
  public readonly int X, Y;   // X and Y must be
readonly
}
```

If you need to apply `readonly` at a more granular level, you can apply the `readonly` modifier (from C# 8) to a struct's *functions*.

This ensures that if the function attempts to modify any field, a compile-time error is generated:

```
struct Point
{
  public int X, Y;
  public readonly void ResetX() => X = 0;  // Error!
}
```

If a `readonly` function calls a non-`readonly` function, the compiler generates a warning (and defensively copies the struct to avoid the possibility of a mutation).

Ref Structs

> **NOTE**
>
> Ref structs were introduced in C# 7.2 as a niche feature primarily for the benefit of the `Span<T>` and `ReadOnlySpan<T>` structs that we describe in Chapter 23 (and the highly optimized `Utf8JsonReader` that we describe in Chapter 11). These structs help with a micro-optimization technique that aims to reduce memory allocations.

Unlike reference types, whose instances always live on the heap, value types live *in-place* (wherever the variable was declared). If a value type appears as a parameter or local variable, it will reside on the stack:

```
void SomeMethod()
{
  Point p;   // p will reside on the stack
}
struct Point { public int X, Y; }
```

But if a value type appears as a field in a class, it will reside on the heap:

```
class MyClass
{
  Point p;   // Lives on heap, because MyClass
instances live on the heap
}
```

Similarly, arrays of structs live on the heap, and boxing a struct sends it to the heap.

Adding the `ref` modifier to a struct's declaration ensures that it can only ever reside on the stack. Attempting to use a *ref struct* in such a way that it could reside on the heap generates a compile-time error:

```
var points = new Point [100];           // Error:
will not compile!
```

```
ref struct Point { public int X, Y; }
class MyClass    { Point P;         }   // Error:
will not compile!
```

Ref structs were introduced mainly for the benefit of the `Span<T>` and `ReadOnlySpan<T>` structs. Because `Span<T>` and `ReadOnlySpan<T>` instances can exist only on the stack, it's possible for them to safely wrap stack-allocated memory.

Ref structs cannot partake in any C# feature that directly or indirectly introduces the possibility of existing on the heap. This includes a number of advanced C# features that we describe in Chapter 4, namely lambda expressions, iterators, and asynchronous functions (because, behind the scenes, these features all create hidden classes with fields). Also, ref structs cannot appear inside non-ref structs, and they cannot implement interfaces (because this could result in boxing).

Access Modifiers

To promote encapsulation, a type or type member can limit its *accessibility* to other types and other assemblies by adding one of five *access modifiers* to the declaration:

`public`
: Fully accessible. This is the implicit accessibility for members of an enum or interface.

`internal`

Accessible only within the containing assembly or friend assemblies. This is the default accessibility for non-nested types.

`private`
Accessible only within the containing type. This is the default accessibility for members of a class or struct.

`protected`
Accessible only within the containing type or subclasses.

`protected internal`
The *union* of `protected` and `internal` accessibility. A member that is `protected internal` is accessible in two ways.

`private protected` (from C# 7.2)
The *intersection* of `protected` and `internal` accessibility. A member that is `private protected` is accessible only within the containing type, or subclasses *that reside in the same assembly* (making it *less* accessible than `protected` or `internal` alone).

Examples

`Class2` is accessible from outside its assembly; `Class1` is not:

```
class Class1 {}                 // Class1 is
internal (default)
```

```
public class Class2 {}
```

`ClassB` exposes field `x` to other types in the same assembly; `ClassA` does not:

```
class ClassA { int x;          } // x is private
(default)
class ClassB { internal int x; }
```

Functions within `Subclass` can call `Bar` but not `Foo`:

```
class BaseClass
{
  void Foo()           {}        // Foo is private
(default)
  protected void Bar() {}
}

class Subclass : BaseClass
{
  void Test1() { Foo(); }       // Error - cannot
access Foo
  void Test2() { Bar(); }       // OK
}
```

Friend Assemblies

You can expose `internal` members to other *friend* assemblies by adding the `System.Runtime.CompilerServices.InternalsVisib`

leTo assembly attribute, specifying the name of the friend assembly as follows:

```
[assembly: InternalsVisibleTo ("Friend")]
```

If the friend assembly has a strong name (see Chapter 17), you must specify its *full* 160-byte public key:

```
[assembly: InternalsVisibleTo ("StrongFriend,
PublicKey=0024f000048c...")]
```

You can extract the full public key from a strongly named assembly with a LINQ query (we explain LINQ in detail in Chapter 8):

```
string key = string.Join ("",

Assembly.GetExecutingAssembly().GetName().GetPublicKe
y()
    .Select (b => b.ToString ("x2")));
```

NOTE

The companion sample in LINQPad invites you to browse to an assembly and then copies the assembly's full public key to the clipboard.

Accessibility Capping

A type caps the accessibility of its declared members. The most common example of capping is when you have an `internal` type with `public` members. For example, consider this:

```
class C { public void Foo() {} }
```

`C`'s (default) `internal` accessibility caps `Foo`'s accessibility, effectively making `Foo internal`. A common reason `Foo` would be marked `public` is to make for easier refactoring should `C` later be changed to `public`.

Restrictions on Access Modifiers

When overriding a base class function, accessibility must be identical on the overridden function. For example:

```
class BaseClass             { protected virtual  void
Foo() {} }
class Subclass1 : BaseClass { protected override void
Foo() {} }  // OK
class Subclass2 : BaseClass { public    override void
Foo() {} }  // Error
```

(An exception is when overriding a `protected internal` method in another assembly, in which case the override must simply be `protected`.)

The compiler prevents any inconsistent use of access modifiers. For example, a subclass itself can be less accessible than a base class but not more:

```
internal class A {}
public class B : A {}          // Error
```

Interfaces

An interface is similar to a class, but only *specifies behavior* and does not hold state (data). Consequently:

- An interface can define only functions and not fields.
- Interface members are *implicitly abstract*. (Although nonabstract functions are permitted from C# 8, this is considered a special case, which we describe in "Default Interface Members".)
- A class (or struct) can implement *multiple* interfaces. In contrast, a class can inherit from only a *single* class, and a struct cannot inherit at all (aside from deriving from `System.ValueType`).

An interface declaration is like a class declaration, but it (typically) provides no implementation for its members because its members are

implicitly abstract. These members will be implemented by the classes and structs that implement the interface. An interface can contain only functions, that is, methods, properties, events, and indexers (which noncoincidentally are precisely the members of a class that can be abstract).

Here is the definition of the `IEnumerator` interface, defined in `System.Collections`:

```
public interface IEnumerator
{
  bool MoveNext();
  object Current { get; }
  void Reset();
}
```

Interface members are always implicitly public and cannot declare an access modifier. Implementing an interface means providing a `public` implementation for all of its members:

```
internal class Countdown : IEnumerator
{
  int count = 11;
  public bool MoveNext() => count-- > 0;
  public object Current => count;
  public void Reset() { throw new
NotSupportedException(); }
}
```

You can implicitly cast an object to any interface that it implements:

```
IEnumerator e = new Countdown();
while (e.MoveNext())
  Console.Write (e.Current);      // 109876543210
```

NOTE

Even though `Countdown` is an internal class, its members that implement `IEnumerator` can be called publicly by casting an instance of `Countdown` to `IEnumerator`. For instance, if a public type in the same assembly defined a method as follows

```
public static class Util
{
  public static object GetCountDown()
=> new CountDown();
}
```

a caller from another assembly could do this:

```
IEnumerator e = (IEnumerator)
Util.GetCountDown();
e.MoveNext();
```

If `IEnumerator` were itself defined as `internal`, this wouldn't be possible.

Extending an Interface

Interfaces can derive from other interfaces. For instance:

```
public interface IUndoable             { void Undo();
}
public interface IRedoable : IUndoable { void Redo();
}
```

`IRedoable` "inherits" all the members of `IUndoable`. In other words, types that implement `IRedoable` must also implement the members of `IUndoable`.

Explicit Interface Implementation

Implementing multiple interfaces can sometimes result in a collision between member signatures. You can resolve such collisions by *explicitly implementing* an interface member. Consider the following example:

```
interface I1 { void Foo(); }
interface I2 { int Foo(); }

public class Widget : I1, I2
{
  public void Foo()
  {
    Console.WriteLine ("Widget's implementation of
I1.Foo");
  }
```

```
  int I2.Foo()
  {
    Console.WriteLine ("Widget's implementation of
I2.Foo");
    return 42;
  }
}
```

Because `I1` and `I2` have conflicting `Foo` signatures, `Widget` explicitly implements `I2`'s `Foo` method. This lets the two methods coexist in one class. The only way to call an explicitly implemented member is to cast to its interface:

```
Widget w = new Widget();
w.Foo();                        // Widget's
implementation of I1.Foo
((I1)w).Foo();                  // Widget's
implementation of I1.Foo
((I2)w).Foo();                  // Widget's
implementation of I2.Foo
```

Another reason to explicitly implement interface members is to hide members that are highly specialized and distracting to a type's normal use case. For example, a type that implements `ISerializable` would typically want to avoid flaunting its `ISerializable` members unless explicitly cast to that interface.

Implementing Interface Members Virtually

An implicitly implemented interface member is, by default, sealed. It must be marked `virtual` or `abstract` in the base class in order to be overridden:

```
public interface IUndoable { void Undo(); }

public class TextBox : IUndoable
{
  public virtual void Undo() => Console.WriteLine
("TextBox.Undo");
}

public class RichTextBox : TextBox
{
  public override void Undo() => Console.WriteLine
("RichTextBox.Undo");
}
```

Calling the interface member through either the base class or the interface calls the subclass's implementation:

```
RichTextBox r = new RichTextBox();
r.Undo();                          //
RichTextBox.Undo
((IUndoable)r).Undo();             //
RichTextBox.Undo
((TextBox)r).Undo();               //
RichTextBox.Undo
```

An explicitly implemented interface member cannot be marked `virtual`, nor can it be overridden in the usual manner. It can, however, be *reimplemented*.

Reimplementing an Interface in a Subclass

A subclass can reimplement any interface member already implemented by a base class. Reimplementation hijacks a member implementation (when called through the interface) and works whether or not the member is `virtual` in the base class. It also works whether a member is implemented implicitly or explicitly—although it works best in the latter case, as we will demonstrate.

In the following example, `TextBox` implements `IUndoable.Undo` explicitly, and so it cannot be marked as `virtual`. To "override" it, `RichTextBox` must reimplement `IUndoable`'s `Undo` method:

```
public interface IUndoable { void Undo(); }

public class TextBox : IUndoable
{
  void IUndoable.Undo() => Console.WriteLine
("TextBox.Undo");
}

public class RichTextBox : TextBox, IUndoable
{
  public void Undo() => Console.WriteLine
("RichTextBox.Undo");
}
```

Calling the reimplemented member through the interface calls the subclass's implementation:

```
RichTextBox r = new RichTextBox();
```

```
r.Undo();                  // RichTextBox.Undo
Case 1
((IUndoable)r).Undo();     // RichTextBox.Undo
Case 2
```

Assuming the same `RichTextBox` definition, suppose that `TextBox` implemented `Undo` *implicitly*:

```
public class TextBox : IUndoable
{
  public void Undo() => Console.WriteLine
("TextBox.Undo");
}
```

This would give us another way to call `Undo`, which would "break" the system, as shown in Case 3:

```
RichTextBox r = new RichTextBox();
r.Undo();                  // RichTextBox.Undo
Case 1
((IUndoable)r).Undo();     // RichTextBox.Undo
Case 2
((TextBox)r).Undo();       // TextBox.Undo
Case 3
```

Case 3 demonstrates that reimplementation hijacking is effective only when a member is called through the interface and not through the base class. This is usually undesirable in that it can create inconsistent semantics. This makes reimplementation most appropriate as a strategy for overriding *explicitly* implemented

interface members.

Alternatives to interface reimplementation

Even with explicit member implementation, interface reimplementation is problematic for a couple of reasons:

- The subclass has no way to call the base class method.
- The base class author might not anticipate that a method be reimplemented and might not allow for the potential consequences.

Reimplementation can be a good last resort when subclassing hasn't been anticipated. A better option, however, is to design a base class such that reimplementation will never be required. There are two ways to achieve this:

- When implicitly implementing a member, mark it `virtual` if appropriate.
- When explicitly implementing a member, use the following pattern if you anticipate that subclasses might need to override any logic:

```
public class TextBox : IUndoable
{
  void IUndoable.Undo()         => Undo();    //
Calls method below
  protected virtual void Undo() => Console.WriteLine
("TextBox.Undo");
}

public class RichTextBox : TextBox
```

```
{
  protected override void Undo() =>
Console.WriteLine("RichTextBox.Undo");
}
```

If you don’t anticipate any subclassing, you can mark the class as `sealed` to preempt interface reimplementation.

Interfaces and Boxing

Converting a struct to an interface causes boxing. Calling an implicitly implemented member on a struct does not cause boxing:

```
interface  I { void Foo();          }
struct S : I { public void Foo() {} }

...
S s = new S();
s.Foo();         // No boxing.

I i = s;         // Box occurs when casting to
interface.
i.Foo();
```

Default Interface Members

From C# 8, you can add a default implementation to an interface member, making it optional to implement:

```
interface ILogger
{
  void Log (string text) => Console.WriteLine (text);
}
```

This is advantageous if you want to add a member to an interface defined in a popular library without breaking (potentially thousands of) implementations.

Default implementations are always explicit, so if a class implementing `ILogger` fails to define a `Log` method, the only way to call it is through the interface:

```
class Logger : ILogger { }
...
((ILogger)new Logger()).Log ("message");
```

This prevents a problem of multiple implementation inheritance: if the same default member is added to two interfaces that a class implements, there is never an ambiguity as to which member is called.

Interfaces can also now define static members (including fields), which can be accessed from code inside default implementations:

```
interface ILogger
{
  void Log (string text) =>
    Console.WriteLine (Prefix + text);

  static string Prefix = "";
```

```
}
```

Because interface members are implicitly public, you can also access static members from the outside:

```
ILogger.Prefix = "File log: ";
```

You can restrict this by adding an accessibility modifier to the static interface member (such as `private`, `protected`, or `internal`).

Instance fields are (still) prohibited. This is in line with the principle of interfaces, which is to define *behavior*, not *state*.

WRITING A CLASS VERSUS AN INTERFACE

As a guideline:

- Use classes and subclasses for types that naturally share an implementation.
- Use interfaces for types that have independent implementations.

Consider the following classes:

```
abstract class Animal {}
abstract class Bird           : Animal {}
abstract class Insect         : Animal {}
abstract class FlyingCreature : Animal {}
abstract class Carnivore      : Animal {}
```

```
// Concrete classes:

class Ostrich : Bird {}
class Eagle   : Bird, FlyingCreature, Carnivore
{}  // Illegal
class Bee     : Insect, FlyingCreature {}
// Illegal
class Flea    : Insect, Carnivore {}
// Illegal
```

The `Eagle`, `Bee`, and `Flea` classes do not compile because inheriting from multiple classes is prohibited. To resolve this, we must convert some of the types to interfaces. The question then arises, which types? Following our general rule, we could say that insects share an implementation, and birds share an implementation, so they remain classes. In contrast, flying creatures have independent mechanisms for flying, and carnivores have independent strategies for eating animals, so we would convert `FlyingCreature` and `Carnivore` to interfaces:

```
interface IFlyingCreature {}
interface ICarnivore      {}
```

In a typical scenario, `Bird` and `Insect` might correspond to a Windows control and a web control; `FlyingCreature` and `Carnivore` might correspond to `IPrintable` and `IUndoable`.

Enums

An *enum* is a special value type that lets you specify a group of

named numeric constants. For example:

```
public enum BorderSide { Left, Right, Top, Bottom }
```

We can use this enum type as follows:

```
BorderSide topSide = BorderSide.Top;
bool isTop = (topSide == BorderSide.Top);   // true
```

Each enum member has an underlying integral value. These are by default:

- Underlying values are of type `int`.
- The constants `0`, `1`, `2`... are automatically assigned, in the declaration order of the enum members.

You can specify an alternative integral type, as follows:

```
public enum BorderSide : byte { Left, Right, Top,
Bottom }
```

You can also specify an explicit underlying value for each enum member:

```
public enum BorderSide : byte { Left=1, Right=2,
```

```
Top=10, Bottom=11 }
```

> **NOTE**
>
> The compiler also lets you explicitly assign *some* of the enum members. The unassigned enum members keep incrementing from the last explicit value. The preceding example is equivalent to the following:
>
> ```
> public enum BorderSide : byte
>  { Left=1, Right, Top=10, Bottom }
> ```

Enum Conversions

You can convert an `enum` instance to and from its underlying integral value with an explicit cast:

```
int i = (int) BorderSide.Left;
BorderSide side = (BorderSide) i;
bool leftOrRight = (int) side <= 2;
```

You can also explicitly cast one enum type to another. Suppose that `HorizontalAlignment` is defined as follows:

```
public enum HorizontalAlignment
{
  Left = BorderSide.Left,
  Right = BorderSide.Right,
  Center
}
```

A translation between the enum types uses the underlying integral values:

```
HorizontalAlignment h = (HorizontalAlignment)
BorderSide.Right;
// same as:
HorizontalAlignment h = (HorizontalAlignment) (int)
BorderSide.Right;
```

The numeric literal `0` is treated specially by the compiler in an `enum` expression and does not require an explicit cast:

```
BorderSide b = 0;    // No cast required
if (b == 0) ...
```

There are two reasons for the special treatment of `0`:

- The first member of an enum is often used as the “default” value.
- For *combined enum* types, `0` means “no flags.”

Flags Enums

You can combine enum members. To prevent ambiguities, members of a combinable enum require explicitly assigned values, typically in powers of two:

```
[Flags]
enum BorderSides { None=0, Left=1, Right=2, Top=4,
Bottom=8 }
```

or

```
enum BorderSides { None=0, Left=1, Right=1<<1,
Top=1<<2, Bottom=1<<3 }
```

To work with combined enum values, you use bitwise operators such as | and &. These operate on the underlying integral values:

```
BorderSides leftRight = BorderSides.Left |
BorderSides.Right;

if ((leftRight & BorderSides.Left) != 0)
  Console.WriteLine ("Includes Left");     //
Includes Left

string formatted = leftRight.ToString();   // "Left,
Right"

BorderSides s = BorderSides.Left;
s |= BorderSides.Right;
Console.WriteLine (s == leftRight);   // True
```

```
s ^= BorderSides.Right;               // Toggles
BorderSides.Right
Console.WriteLine (s);                // Left
```

By convention, the `Flags` attribute should always be applied to an enum type when its members are combinable. If you declare such an `enum` without the `Flags` attribute, you can still combine members, but calling `ToString` on an `enum` instance will emit a number rather than a series of names.

By convention, a combinable enum type is given a plural rather than singular name.

For convenience, you can include combination members within an enum declaration itself:

```
[Flags]
enum BorderSides
{
  None=0,
  Left=1, Right=1<<1, Top=1<<2, Bottom=1<<3,
  LeftRight = Left | Right,
  TopBottom = Top  | Bottom,
  All       = LeftRight | TopBottom
}
```

Enum Operators

The operators that work with enums are:

```
=   ==   !=   <   >   <=   >=   +   -   ^  &  |   ~
+=   -=   ++  --   sizeof
```

The bitwise, arithmetic, and comparison operators return the result of processing the underlying integral values. Addition is permitted between an enum and an integral type, but not between two enums.

Type-Safety Issues

Consider the following enum:

```
public enum BorderSide { Left, Right, Top, Bottom }
```

Because an enum can be cast to and from its underlying integral type, the actual value it can have might fall outside the bounds of a legal enum member:

```
BorderSide b = (BorderSide) 12345;
Console.WriteLine (b);                  // 12345
```

The bitwise and arithmetic operators can produce similarly invalid values:

```
BorderSide b = BorderSide.Bottom;
b++;                                    // No errors
```

An invalid `BorderSide` would break the following code:

```
void Draw (BorderSide side)
{
  if      (side == BorderSide.Left)  {...}
  else if (side == BorderSide.Right) {...}
  else if (side == BorderSide.Top)   {...}
  else                               {...} // Assume
BorderSide.Bottom
}
```

One solution is to add another `else` clause:

```
  ...
  else if (side == BorderSide.Bottom) ...
  else throw new ArgumentException ("Invalid
BorderSide: " + side, "side");
```

Another workaround is to explicitly check an enum value for validity. The static `Enum.IsDefined` method does this job:

```
BorderSide side = (BorderSide) 12345;
Console.WriteLine (Enum.IsDefined (typeof
(BorderSide), side));   // False
```

Unfortunately, `Enum.IsDefined` does not work for flagged enums. However, the following helper method (a trick dependent on the behavior of `Enum.ToString()`) returns `true` if a given

flagged enum is valid:

```
for (int i = 0; i <= 16; i++)
{
  BorderSides side = (BorderSides)i;
  Console.WriteLine (IsFlagDefined (side) + " " +
side);
}

bool IsFlagDefined (Enum e)
{
  decimal d;
  return !decimal.TryParse(e.ToString(), out d);
}

[Flags]
public enum BorderSides { Left=1, Right=2, Top=4,
Bottom=8 }
```

Nested Types

A *nested type* is declared within the scope of another type:

```
public class TopLevel
{
  public class Nested { }               // Nested
class
  public enum Color { Red, Blue, Tan }  // Nested
enum
}
```

A nested type has the following features:

- It can access the enclosing type's private members and everything else the enclosing type can access.
- You can declare it with the full range of access modifiers rather than just `public` and `internal`.
- The default accessibility for a nested type is `private` rather than `internal`.
- Accessing a nested type from outside the enclosing type requires qualification with the enclosing type's name (like when accessing static members).

For example, to access `Color.Red` from outside our `TopLevel` class, we'd need to do this:

```
TopLevel.Color color = TopLevel.Color.Red;
```

All types (classes, structs, interfaces, delegates, and enums) can be nested within either a class or a struct.

Here is an example of accessing a private member of a type from a nested type:

```
public class TopLevel
{
  static int x;
  class Nested
  {
    static void Foo() { Console.WriteLine
(TopLevel.x); }
```

```
  }
}
```

Here is an example of applying the `protected` access modifier to a nested type:

```
public class TopLevel
{
  protected class Nested { }
}

public class SubTopLevel : TopLevel
{
  static void Foo() { new TopLevel.Nested(); }
}
```

Here is an example of referring to a nested type from outside the enclosing type:

```
public class TopLevel
{
  public class Nested { }
}

class Test
{
  TopLevel.Nested n;
}
```

Nested types are used heavily by the compiler itself when it generates private classes that capture state for constructs such as

iterators and anonymous methods.

> **NOTE**
>
> If the sole reason for using a nested type is to avoid cluttering a namespace with too many types, consider using a nested namespace, instead. A nested type should be used because of its stronger access control restrictions, or when the nested class must access private members of the containing class.

Generics

C# has two separate mechanisms for writing code that is reusable across different types: *inheritance* and *generics*. Whereas inheritance expresses reusability with a base type, generics express reusability with a "template" that contains "placeholder" types. Generics, when compared to inheritance, can *increase type safety* and *reduce casting and boxing*.

> **NOTE**
>
> C# generics and C++ templates are similar concepts, but they work differently. We explain this difference in "C# Generics Versus C++ Templates".

Generic Types

A generic type declares *type parameters*—placeholder types to be filled in by the consumer of the generic type, which supplies the *type arguments*. Here is a generic type `Stack<T>`, designed to stack instances of type `T`. `Stack<T>` declares a single type parameter `T`:

```
public class Stack<T>
{
  int position;
  T[] data = new T[100];
  public void Push (T obj)  => data[position++] =
obj;
  public T Pop()            => data[--position];
}
```

We can use `Stack<T>` as follows:

```
var stack = new Stack<int>();
stack.Push (5);
stack.Push (10);
int x = stack.Pop();        // x is 10
int y = stack.Pop();        // y is 5
```

`Stack<int>` fills in the type parameter `T` with the type argument `int`, implicitly creating a type on the fly (the synthesis occurs at

runtime). Attempting to push a string onto our `Stack<int>` would, however, produce a compile-time error. `Stack<int>` effectively has the following definition (substitutions appear in bold, with the class name hashed out to avoid confusion):

```
public class ###
{
  int position;
  int[] data = new int[100];
  public void Push (int obj)  => data[position++] =
obj;
  public int Pop()            => data[--position];
}
```

Technically, we say that `Stack<T>` is an *open type*, whereas `Stack<int>` is a *closed type*. At runtime, all generic type instances are closed—with the placeholder types filled in. This means that the following statement is illegal:

```
var stack = new Stack<T>();   // Illegal: What is T?
```

However, it's legal if it's within a class or method that itself defines `T` as a type parameter:

```
public class Stack<T>
{
  ...
  public Stack<T> Clone()
  {
```

```
    Stack<T> clone = new Stack<T>();   // Legal
    ...
  }
}
```

Why Generics Exist

Generics exist to write code that is reusable across different types. Suppose that we need a stack of integers, but we don't have generic types. One solution would be to hardcode a separate version of the class for every required element type (e.g., `IntStack`, `StringStack`, etc.). Clearly, this would cause considerable code duplication. Another solution would be to write a stack that is generalized by using `object` as the element type:

```
public class ObjectStack
{
  int position;
  object[] data = new object[10];
  public void Push (object obj) => data[position++] =
obj;
  public object Pop()           => data[--position];
}
```

An `ObjectStack`, however, wouldn't work as well as a hardcoded `IntStack` for specifically stacking integers. An `ObjectStack` would require boxing and downcasting that could not be checked at compile time:

```
// Suppose we just want to store integers here:
```

```
ObjectStack stack = new ObjectStack();

stack.Push ("s");          // Wrong type, but no
error!
int i = (int)stack.Pop();  // Downcast - runtime
error
```

What we need is both a general implementation of a stack that works for all element types as well as a way to easily specialize that stack to a specific element type for increased type safety and reduced casting and boxing. Generics give us precisely this by allowing us to parameterize the element type. `Stack<T>` has the benefits of both `ObjectStack` and `IntStack`. Like `ObjectStack`, `Stack<T>` is written once to work *generally* across all types. Like `IntStack`, `Stack<T>` is *specialized* for a particular type—the beauty is that this type is `T`, which we substitute on the fly.

> **NOTE**
>
> `ObjectStack` is functionally equivalent to `Stack<object>`.

Generic Methods

A generic method declares type parameters within the signature of a method.

With generic methods, many fundamental algorithms can be implemented in a general-purpose way. Here is a generic method that swaps the contents of two variables of any type T:

```
static void Swap<T> (ref T a, ref T b)
{
  T temp = a;
  a = b;
  b = temp;
}
```

Swap<T> is called as follows:

```
int x = 5;
int y = 10;
Swap (ref x, ref y);
```

Generally, there is no need to supply type arguments to a generic method, because the compiler can implicitly infer the type. If there is ambiguity, generic methods can be called with type arguments as follows:

```
Swap<int> (ref x, ref y);
```

Within a generic *type*, a method is not classed as generic unless it *introduces* type parameters (with the angle bracket syntax). The Pop method in our generic stack merely uses the type's existing type

parameter, T, and is not classed as a generic method.

Methods and types are the only constructs that can introduce type parameters. Properties, indexers, events, fields, constructors, operators, and so on cannot declare type parameters, although they can partake in any type parameters already declared by their enclosing type. In our generic stack example, for instance, we could write an indexer that returns a generic item:

```
public T this [int index] => data [index];
```

Similarly, constructors can partake in existing type parameters but not *introduce* them:

```
public Stack<T>() { }   // Illegal
```

Declaring Type Parameters

Type parameters can be introduced in the declaration of classes, structs, interfaces, delegates (covered in Chapter 4), and methods. Other constructs, such as properties, cannot *introduce* a type parameter, but they can *use* one. For example, the property `Value` uses `T`:

```
public struct Nullable<T>
{
  public T Value { get; }
```

```
}
```

A generic type or method can have multiple parameters:

```
class Dictionary<TKey, TValue> {...}
```

To instantiate:

```
Dictionary<int,string> myDict = new
Dictionary<int,string>();
```

or

```
var myDict = new Dictionary<int,string>();
```

Generic type names and method names can be overloaded as long as the number of type parameters is different. For example, the following three type names do not conflict:

```
class A        {}
class A<T>     {}
class A<T1,T2> {}
```

> **NOTE**
>
> By convention, generic types and methods with a *single* type parameter typically name their parameter `T`, as long as the intent of the parameter is clear. When using *multiple* type parameters, each parameter is prefixed with `T` but has a more descriptive name.

typeof and Unbound Generic Types

Open generic types do not exist at runtime: they are closed as part of compilation. However, it is possible for an *unbound* generic type to exist at runtime—purely as a `Type` object. The only way to specify an unbound generic type in C# is via the `typeof` operator:

```
class A<T> {}
class A<T1,T2> {}
...

Type a1 = typeof (A<>);   // Unbound type (notice no
type arguments).
Type a2 = typeof (A<,>);  // Use commas to indicate
multiple type args.
```

Open generic types are used in conjunction with the Reflection API (Chapter 18).

You can also use the `typeof` operator to specify a closed type:

```
Type a3 = typeof (A<int,int>);
```

Or, you can specify an open type (which is closed at runtime):

```
class B<T> { void X() { Type t = typeof (T); } }
```

The default Generic Value

You can use the `default` keyword to get the default value for a generic type parameter. The default value for a reference type is `null`, and the default value for a value type is the result of bitwise-zeroing the value type's fields:

```
static void Zap<T> (T[] array)
{
  for (int i = 0; i < array.Length; i++)
    array[i] = default(T);
}
```

From C# 7.1, you can omit the type argument for cases in which the compiler is able to infer it. We could replace the last line of code with this:

```
    array[i] = default;
```

Generic Constraints

By default, you can substitute a type parameter with any type whatsoever. *Constraints* can be applied to a type parameter to require more specific type arguments. These are the possible constraints:

```
where T : base-class   // Base-class constraint
where T : interface    // Interface constraint
where T : class        // Reference-type constraint
where T : class?       // (See "Nullable reference
types")
where T : struct       // Value-type constraint
(excludes Nullable types)
where T : unmanaged    // Unmanaged constraint
where T : new()        // Parameterless constructor
constraint
where U : T            // Naked type constraint
where T : notnull      // Non-nullable value type, or
(from C# 8)
                       // a non-nullable reference
type
```

In the following example, `GenericClass<T,U>` requires `T` to derive from (or be identical to) `SomeClass` and implement `Interface1`, and requires `U` to provide a parameterless constructor:

```
class     SomeClass {}
interface Interface1 {}

class GenericClass<T,U> where T : SomeClass,
Interface1
                         where U : new()
{...}
```

You can apply constraints wherever type parameters are defined, in both methods and type definitions.

A *base-class constraint* specifies that the type parameter must subclass (or match) a particular class; an *interface constraint* specifies that the type parameter must implement that interface. These constraints allow instances of the type parameter to be implicitly converted to that class or interface.

For example, suppose that we want to write a generic `Max` method, which returns the maximum of two values. We can take advantage of the generic interface defined in the `System` namespace called `IComparable<T>`:

```
public interface IComparable<T>   // Simplified
version of interface
{
  int CompareTo (T other);
}
```

`CompareTo` returns a positive number if `this` is greater than `other`. Using this interface as a constraint, we can write a `Max` method as follows (to avoid distraction, null checking is omitted):

```
static T Max <T> (T a, T b) where T : IComparable<T>
{
  return a.CompareTo (b) > 0 ? a : b;
}
```

The `Max` method can accept arguments of any type implementing

`IComparable<T>` (which includes most built-in types such as `int` and `string`):

```
int z = Max (5, 10);                // 10
string last = Max ("ant", "zoo");   // zoo
```

The *class constraint* and *struct constraint* specify that `T` must be a reference type or (non-nullable) value type. A great example of the struct constraint is the `System.Nullable<T>` struct (we discuss this class in depth in "Nullable Value Types" in Chapter 4):

```
struct Nullable<T> where T : struct {...}
```

The *unmanaged constraint* (introduced in C# 7.3) is a stronger version of a struct constraint: `T` must be a simple value type or a struct that is (recursively) free of any reference types.

The *parameterless constructor constraint* requires `T` to have a public parameterless constructor. If this constraint is defined, you can call `new()` on `T`:

```
static void Initialize<T> (T[] array) where T : new()
{
  for (int i = 0; i < array.Length; i++)
    array[i] = new T();
}
```

The *naked type constraint* requires one type parameter to derive from (or match) another type parameter. In this example, the method `FilteredStack` returns another `Stack`, containing only the subset of elements where the type parameter `U` is of the type parameter `T`:

```
class Stack<T>
{
  Stack<U> FilteredStack<U>() where U : T {...}
}
```

Subclassing Generic Types

A generic class can be subclassed just like a nongeneric class. The subclass can leave the base class's type parameters open, as in the following example:

```
class Stack<T>                    {...}
class SpecialStack<T> : Stack<T> {...}
```

Or, the subclass can close the generic type parameters with a concrete type:

```
class IntStack : Stack<int>  {...}
```

A subtype can also introduce fresh type arguments:

```
class List<T>                        {...}
class KeyedList<T,TKey> : List<T> {...}
```

> **NOTE**
>
> Technically, *all* type arguments on a subtype are fresh: you could say that a subtype closes and then reopens the base type arguments. This means that a subclass can give new (and potentially more meaningful) names to the type arguments that it reopens:
>
> ```
> class List<T> {...}
> class KeyedList<TElement,TKey> :
> List<TElement> {...}
> ```

Self-Referencing Generic Declarations

A type can name *itself* as the concrete type when closing a type argument:

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

public class Balloon : IEquatable<Balloon>
{
```

```
  public string Color { get; set; }
  public int CC { get; set; }

  public bool Equals (Balloon b)
  {
    if (b == null) return false;
    return b.Color == Color && b.CC == CC;
  }
}
```

The following are also legal:

```
class Foo<T> where T : IComparable<T> { ... }
class Bar<T> where T : Bar<T> { ... }
```

Static Data

Static data is unique for each closed type:

```
Console.WriteLine (++Bob<int>.Count);     // 1
Console.WriteLine (++Bob<int>.Count);     // 2
Console.WriteLine (++Bob<string>.Count);  // 1
Console.WriteLine (++Bob<object>.Count);  // 1

class Bob<T> { public static int Count; }
```

Type Parameters and Conversions

C#’s cast operator can perform several kinds of conversion, including:

- Numeric conversion
- Reference conversion
- Boxing/unboxing conversion
- Custom conversion (via operator overloading; see Chapter 4)

The decision as to which kind of conversion will take place happens at *compile time*, based on the known types of the operands. This creates an interesting scenario with generic type parameters, because the precise operand types are unknown at compile time. If this leads to ambiguity, the compiler generates an error.

The most common scenario is when you want to perform a reference conversion:

```
StringBuilder Foo<T> (T arg)
{
  if (arg is StringBuilder)
    return (StringBuilder) arg;   // Will not compile
  ...
}
```

Without knowledge of `T`'s actual type, the compiler is concerned that you might have intended this to be a *custom conversion*. The simplest solution is to instead use the `as` operator, which is unambiguous because it cannot perform custom conversions:

```
StringBuilder Foo<T> (T arg)
{
```

```
  StringBuilder sb = arg as StringBuilder;
  if (sb != null) return sb;
  ...
}
```

A more general solution is to first cast to `object`. This works because conversions to/from `object` are assumed not to be custom conversions, but reference or boxing/unboxing conversions. In this case, `StringBuilder` is a reference type, so it must be a reference conversion:

```
return (StringBuilder) (object) arg;
```

Unboxing conversions can also introduce ambiguities. The following could be an unboxing, numeric, or custom conversion:

```
int Foo<T> (T x) => (int) x;     // Compile-time
error
```

The solution, again, is to first cast to `object` and then to `int` (which then unambiguously signals an unboxing conversion in this case):

```
int Foo<T> (T x) => (int) (object) x;
```

Covariance

Assuming A is convertible to B, X has a covariant type parameter if X<A> is convertible to X<B>.

> **NOTE**
>
> With C#'s notion of covariance (and contravariance), "convertible" means convertible via an *implicit reference conversion*—such as A *subclassing* B, or A *implementing* B. Numeric conversions, boxing conversions, and custom conversions are not included.

For instance, type `IFoo<T>` has a covariant `T` if the following is legal:

```
IFoo<string> s = ...;
IFoo<object> b = s;
```

Interfaces permit covariant type parameters (as do delegates; see Chapter 4), but classes do not. Arrays also allow covariance (`A[]` can be converted to `B[]` if A has an implicit reference conversion to B) and are discussed here for comparison.

> **NOTE**
>
> Covariance and contravariance (or simply “variance”) are advanced concepts. The motivation behind introducing and enhancing variance in C# was to allow generic interface and generic types (in particular, those defined in .NET, such as `IEnumerable<T>`) to work *more as you’d expect*. You can benefit from this without understanding the details behind covariance and contravariance.

Variance is not automatic

To ensure static type safety, type parameters are not automatically variant. Consider the following:

```
class Animal {}
class Bear : Animal {}
class Camel : Animal {}

public class Stack<T>   // A simple Stack
implementation
{
  int position;
  T[] data = new T[100];
  public void Push (T obj)  => data[position++] =
obj;
  public T Pop()            => data[--position];
}
```

The following fails to compile:

```
Stack<Bear> bears = new Stack<Bear>();
Stack<Animal> animals = bears;            // Compile-
time error
```

That restriction prevents the possibility of runtime failure with the following code:

```
animals.Push (new Camel());      // Trying to add
Camel to bears
```

Lack of covariance, however, can hinder reusability. Suppose, for instance, that we wanted to write a method to `Wash` a stack of animals:

```
public class ZooCleaner
{
  public static void Wash (Stack<Animal> animals)
{...}
}
```

Calling `Wash` with a stack of bears would generate a compile-time error. One workaround is to redefine the `Wash` method with a constraint:

```
class ZooCleaner
{
  public static void Wash<T> (Stack<T> animals) where
T : Animal { ... }
}
```

We can now call `Wash` as follows:

```
Stack<Bear> bears = new Stack<Bear>();
ZooCleaner.Wash (bears);
```

Another solution is to have `Stack<T>` implement an interface with a covariant type parameter, as you'll see shortly.

Arrays

For historical reasons, array types support covariance. This means that `B[]` can be cast to `A[]` if `B` subclasses `A` (and both are reference types):

```
Bear[] bears = new Bear[3];
Animal[] animals = bears;     // OK
```

The downside of this reusability is that element assignments can fail at runtime:

```
animals[0] = new Camel();     // Runtime error
```

Declaring a covariant type parameter

Type parameters on interfaces and delegates can be declared covariant by marking them with the `out` modifier. This modifier ensures that, unlike with arrays, covariant type parameters are fully type-safe.

We can illustrate this with our `Stack<T>` class by having it implement the following interface:

```
public interface IPoppable<out T> { T Pop(); }
```

The `out` modifier on `T` indicates that `T` is used only in *output positions* (e.g., return types for methods). The `out` modifier flags the type parameter as *covariant* and allows us to do this:

```
var bears = new Stack<Bear>();
bears.Push (new Bear());
// Bears implements IPoppable<Bear>. We can convert
to IPoppable<Animal>:
IPoppable<Animal> animals = bears;   // Legal
Animal a = animals.Pop();
```

The conversion from `bears` to `animals` is permitted by the compiler—by virtue of the type parameter being covariant. This is type-safe because the case the compiler is trying to avoid—pushing a `Camel` onto the stack—can't occur, because there's no way to feed a `Camel` *in*to an interface where `T` can appear only in *out*put positions.

NOTE

Covariance (and contravariance) in interfaces is something that you typically *consume*: it's less common that you need to *write* variant interfaces.

NOTE

Curiously, method parameters marked as `out` are not eligible for covariance, due to a limitation in the CLR.

We can leverage the ability to cast covariantly to solve the reusability problem described earlier:

```
public class ZooCleaner
{
  public static void Wash (IPoppable<Animal> animals)
{ ... }
}
```

NOTE

The `IEnumerator<T>` and `IEnumerable<T>` interfaces

described in Chapter 7 have a covariant `T`. This allows you to cast `IEnumerable<string>` to `IEnumerable<object>`, for instance.

The compiler will generate an error if you use a covariant type parameter in an *input* position (e.g., a parameter to a method or a writable property).

NOTE

Covariance (and contravariance) works only for elements with *reference conversions*—not *boxing conversions*. (This applies both to type parameter variance and array variance.) So, if you wrote a method that accepted a parameter of type `IPoppable<object>`, you could call it with `IPoppable<string>` but not `IPoppable<int>`.

Contravariance

We previously saw that, assuming that A allows an implicit reference conversion to B, a type X has a covariant type parameter if X<A> allows a reference conversion to X<B>. *Contravariance* is when you can convert in the reverse direction—from X<B> to X<A>. This is supported if the type parameter appears only in *input* positions and is

designated with the `in` modifier. Extending our previous example, suppose the `Stack<T>` class implements the following interface:

```
public interface IPushable<in T> { void Push (T obj);
}
```

We can now legally do this:

```
IPushable<Animal> animals = new Stack<Animal>();
IPushable<Bear> bears = animals;    // Legal
bears.Push (new Bear());
```

No member in `IPushable` *outputs* a `T`, so we can't get into trouble by casting `animals` to `bears` (there's no way to `Pop`, for instance, through that interface).

NOTE

Our `Stack<T>` class can implement both `IPushable<T>` and `IPoppable<T>`—despite `T` having opposing variance annotations in the two interfaces! This works because you must exercise variance through the interface and not the class; therefore, you must commit to the lens of either `IPoppable` or `IPushable` before performing a variant conversion. This lens then restricts you to the operations that are legal under the appropriate variance rules.

This also illustrates why *classes* do not allow variant type

parameters: concrete implementations typically require data to flow in both directions.

To give another example, consider the following interface, defined in the `System` namespace:

```
public interface IComparer<in T>
{
  // Returns a value indicating the relative ordering
of a and b
  int Compare (T a, T b);
}
```

Because the interface has a contravariant `T`, we can use an `IComparer<object>` to compare two *strings*:

```
var objectComparer = Comparer<object>.Default;
// objectComparer implements IComparer<object>
IComparer<string> stringComparer = objectComparer;
int result = stringComparer.Compare ("Brett",
"Jemaine");
```

Mirroring covariance, the compiler will report an error if you try to use a contravariant type parameter in an output position (e.g., as a return value or in a readable property).

C# Generics Versus C++ Templates

C# generics are similar in application to C++ templates, but they work very differently. In both cases, a synthesis between the producer and consumer must take place in which the placeholder types of the producer are filled in by the consumer. However, with C# generics, producer types (i.e., open types such as `List<T>`) can be compiled into a library (such as *mscorlib.dll*). This works because the synthesis between the producer and the consumer that produces closed types doesn't actually happen until runtime. With C++ templates, this synthesis is performed at compile time. This means that in C++ you don't deploy template libraries as *.dll*s—they exist only as source code. It also makes it difficult to dynamically inspect, let alone create, parameterized types on the fly.

To dig deeper into why this is the case, consider again the `Max` method in C#:

```
static T Max <T> (T a, T b) where T : IComparable<T>
  => a.CompareTo (b) > 0 ? a : b;
```

Why couldn't we have implemented it like this?

```
static T Max <T> (T a, T b)
  => (a > b ? a : b);             // Compile error
```

The reason is that `Max` needs to be compiled once and work for all possible values of `T`. Compilation cannot succeed because there is no

single meaning for > across all values of T—in fact, not every T even has a > operator. In contrast, the following code shows the same Max method written with C++ templates. This code will be compiled separately for each value of T, taking on whatever semantics > has for a particular T, and failing to compile if a particular T does not support the > operator:

```
template <class T> T Max (T a, T b)
{
  return a > b ? a : b;
}
```

---

1 The reference type can also be System.ValueType or System.Enum (Chapter 6).