

# Chapter 2. C# Language Basics

---

In this chapter, we introduce the basics of the C# language.

## NOTE

All programs and code snippets in this and the following two chapters are available as interactive samples in LINQPad. Working through these samples in conjunction with the book accelerates learning in that you can edit the samples and instantly see the results without needing to set up projects and solutions in Visual Studio.

To download them in *LINQPad*, click the Samples tab, and then click “Download more samples.”

## A First C# Program

Following is a program that multiplies 12 by 30 and prints the result, 360, to the screen. The double forward slash indicates that the remainder of a line is a *comment*:

```
using System; // Importing namespace

class Test // Class declaration
{
    static void Main() // Method declaration
    {
```

```
int x = 12 * 30;           // Statement 1
Console.WriteLine (x);     // Statement 2
}
// End of method
}
// End of class
```

At the heart of this program lie two *statements*:

```
int x = 12 * 30;
Console.WriteLine (x);
```

Statements in C# execute sequentially and are terminated by a semicolon (or a *code block*, as you'll see later). The first statement computes the *expression* `12 * 30` and stores the result in a *local variable*, named `x`, which is an integer type. The second statement calls the `Console` class's *WriteLine method*, to print the variable `x` to a text window on the screen.

A *method* performs an action in a series of statements, called a *statement block*—a pair of braces containing zero or more statements. We defined a single method named `Main`:

```
static void Main()
{
    ...
}
```

Writing higher-level functions that call upon lower-level functions simplifies a program. We can *refactor* our program with a reusable method that multiplies an integer by 12, as follows:

```
using System;

class Test
{
    static void Main()
    {
        Console.WriteLine (FeetToInches (30));          // 360
        Console.WriteLine (FeetToInches (100));         // 1200
    }

    static int FeetToInches (int feet)
    {
        int inches = feet * 12;
        return inches;
    }
}
```

A method can receive *input* data from the caller by specifying *parameters* and *output* data back to the caller by specifying a *return type*. We defined a method called **FeetToInches** that has a parameter for inputting feet, and a return type for outputting inches:

```
static int FeetToInches (int feet) {...}
```

The *literals* **30** and **100** are the *arguments* passed to the **FeetToInches** method. The **Main** method in our example has empty parentheses because it has no parameters; it is **void** because it doesn't return any value to its caller:

```
static void Main()
```

C# recognizes a method called `Main` as signaling the default entry point of execution. The `Main` method can optionally return an integer (rather than `void`) in order to return a value to the execution environment (where a nonzero value typically indicates an error). The `Main` method can also optionally accept an array of strings as a parameter (that will be populated with any arguments passed to the executable); for example:

```
static int Main (string[] args) {...}
```

### NOTE

An array (such as `string[]`) represents a fixed number of elements of a particular type. Arrays are specified by placing square brackets after the element type. We describe them in “[Arrays](#)”.

(The `Main` method can also be declared `async` and return a `Task` or `Task<int>` in support of asynchronous programming, which we cover in [Chapter 14](#).)

Methods are one of several kinds of functions in C#. Another kind of function we used in our example program was the *\* operator*, which performs multiplication. There are also *constructors*, *properties*, *events*, *indexers*, and *finalizers*.

In our example, the two methods are grouped into a *class*. A class groups function members and data members to form an object-oriented building block. The `Console` class groups members that

handle command-line input/output (I/O) functionality, such as the `WriteLine` method. Our `Test` class groups two methods—the `Main` method and the `FeetToInches` method. A class is a kind of *type*, which we examine in “[Type Basics](#)”.

At the outermost level of a program, types are organized into *namespaces*. The `using` directive makes the `System` namespace available to our application, to use the `Console` class. We could define all of our classes within the `TestPrograms` namespace, as follows:

```
using System;

namespace TestPrograms
{
    class Test { ... }
    class Test2 { ... }
}
```

The .NET Core libraries are organized into nested namespaces. For example, this is the namespace that contains types for handling text:

```
using System.Text;
```

The `using` directive is there for convenience; you can also refer to a type by its fully qualified name, which is the type name prefixed with its namespace, such as `System.Text.StringBuilder`.

## Compilation

The C# compiler compiles source code (as a set of files with the `.cs` extension) into an *assembly*. An assembly is the unit of packaging and deployment in .NET. An assembly can be either an *application* or a *library*. A normal console or Windows application has a `Main` method (the *entry point*), whereas a library does not. The purpose of a library is to be called upon (*referenced*) by an application or by other libraries. .NET Core itself is a set of assemblies (as well as a runtime environment).

### NOTE

Unlike .NET Framework, .NET Core assemblies never have a `.exe` extension. The `.exe` you might see after building a .NET Core application is a platform-specific native loader responsible for starting your application's `.dll` assembly.

.NET Core also allows you to create a self-contained deployment that includes the loader, your assemblies, and the .NET Core Framework—all in a single `.exe` file.

The `dotnet` tool (`dotnet.exe` on Windows) helps you to manage .NET source code and binaries from the command line. You can use it to both build and run your program, as an alternative to using an Integrated Development Environment (IDE) such as Visual Studio or Visual Studio Code.

You can obtain the `dotnet` tool either by installing the .NET Core SDK or by installing Visual Studio. Its default location is `%ProgramFiles%\dotnet` on Windows or `/usr/bin/dotnet` on Ubuntu Linux.

To compile an application, the `dotnet` tool requires a *project file* as well as one or more C# files. The following command *scaffolds* a new console project (creates its basic structure):

```
dotnet new Console -n MyFirstProgram
```

This creates a subfolder called *MyFirstProgram* containing a project file called *MyFirstProgram.csproj* and a C# file called *Program.cs* with a `Main` method that prints “Hello, world”.

To build and run your program, run this command from the *MyFirstProgram* folder:

```
dotnet run MyFirstProgram
```

Or, if you just want to build without running:

```
dotnet build MyFirstProgram.csproj
```

The output assembly will be written to a subdirectory under *bin\debug*.

We explain assemblies in detail in [Chapter 18](#).

## Syntax

C# syntax is inspired by C and C++ syntax. In this section, we describe C#'s elements of syntax, using the following program:

```
using System;

class Test
{
    static void Main()
    {
        int x = 12 * 30;
        Console.WriteLine (x);
    }
}
```

## Identifiers and Keywords

*Identifiers* are names that programmers choose for their classes, methods, variables, and so on. Here are the identifiers in our example program, in the order in which they appear:

```
System  Test   Main   x   Console  WriteLine
```

An identifier must be a whole word, essentially made up of Unicode characters starting with a letter or underscore. C# identifiers are case sensitive. By convention, parameters, local variables, and private fields should be in *camel case* (e.g., `myVariable`), and all other identifiers should be in *Pascal case* (e.g., `MyMethod`).

*Keywords* are names that mean something special to the compiler. These are the keywords in our example program:

```
using  class   static   void   int
```

Most keywords are *reserved*, which means that you can't use them as identifiers. Here is the full list of C# reserved keywords:

abstract	do	in	protected	true
as	double	int	public	try
base	else	interface	readonly	typeof
bool	enum	internal	ref	uint
break	event	is	return	ulong
byte	explicit	lock	sbyte	unchecked
case	extern	long	sealed	unsafe
catch	false	namespace	short	ushort
char	finally	new	sizeof	using
checked	fixed	null	stackalloc	virtual
class	float	object	static	void
const	for	operator	string	volatile
continue	foreach	out	struct	while
decimal	goto	override	switch	
default	if	params	this	
delegate	implicit	private	throw	

## AVOIDING CONFLICTS

If you really want to use an identifier that clashes with a reserved keyword, you can do so by qualifying it with the @ prefix. For instance:

```
class class {...}      // Illegal
class @class {...}    // Legal
```

The @ symbol doesn't form part of the identifier itself. So, `@myVariable` is the same as `myVariable`.

## NOTE

The @ prefix can be useful when consuming libraries written in other .NET languages that have different keywords.

## CONTEXTUAL KEYWORDS

Some keywords are *contextual*, meaning that you also can use them as identifiers—without an @ symbol:

add	dynamic	into	remove	where
alias	equals	join	select	yield
ascending	from	let	set	
async	get	nameof	unmanaged	
await	global	on	value	
by	group	orderby	var	
descending	in	partial	when	

With contextual keywords, ambiguity cannot arise within the context in which they are used.

## Literals, Punctuators, and Operators

*Literals* are primitive pieces of data lexically embedded into the program. The literals we used in our example program are 12 and 30.

*Punctuators* help demarcate the structure of the program. These are the punctuators we used in our example program:

```
{ } ;
```

The braces group multiple statements into a *statement block*.

The semicolon terminates a statement. (Statement blocks, however, do not require a semicolon.) Statements can wrap multiple lines:

```
Console.WriteLine  
(1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10);
```

An *operator* transforms and combines expressions. Most operators in C# are denoted with a symbol, such as the multiplication operator, \*. We discuss operators in more detail later in this chapter. These are the operators we used in our example program:

```
. () * =
```

A period denotes a member of something (or a decimal point with numeric literals). Parentheses are used when declaring or calling a method; empty parentheses are used when the method accepts no arguments. (Parentheses also have other purposes that you'll see later in this chapter.) An equals sign performs *assignment*. (The double equals sign, ==, performs equality comparison, as you'll see later.)

## Comments

C# offers two different styles of source-code documentation: *single-line comments* and *multiline comments*. A single-line comment begins

with a double forward slash and continues until the end of the line; for example:

```
int x = 3; // Comment about assigning 3 to x
```

A multiline comment begins with `/*` and ends with `*/`; for example:

```
int x = 3; /* This is a comment that  
spans two lines */
```

Comments can embed XML documentation tags, which we explain in “[XML Documentation](#)” in [Chapter 4](#).

## Type Basics

A *type* defines the blueprint for a value. In our example, we used two literals of type `int` with values `12` and `30`. We also declared a *variable* of type `int` whose name was `x`:

```
static void Main()  
{  
    int x = 12 * 30;  
    Console.WriteLine (x);  
}
```

A *variable* denotes a storage location that can contain different values over time. In contrast, a *constant* always represents the same value (more on this later):

```
const int y = 360;
```

All values in C# are *instances* of a type. The meaning of a value and the set of possible values a variable can have are determined by its type.

## Predefined Type Examples

Predefined types are types that are specially supported by the compiler. The `int` type is a predefined type for representing the set of integers that fit into 32 bits of memory, from  $-2^{31}$  to  $2^{31}-1$ , and is the default type for numeric literals within this range. We can perform functions such as arithmetic with instances of the `int` type, as follows:

```
int x = 12 * 30;
```

Another predefined C# type is `string`. The `string` type represents a sequence of characters, such as “.NET” or “<http://oreilly.com>”. We can work with strings by calling functions on them, as follows:

```
string message = "Hello world";
string upperMessage = message.ToUpper();
Console.WriteLine (upperMessage); // HELLO WORLD

int x = 2015;
message = message + x.ToString();
Console.WriteLine (message); // Hello world2015
```

The predefined `bool` type has exactly two possible values: `true` and `false`. The `bool` type is commonly used with an `if` statement to conditionally branch execution flow:

```
bool simpleVar = false;
if (simpleVar)
    Console.WriteLine ("This will not print");

int x = 5000;
bool lessThanAMile = x < 5280;
if (lessThanAMile)
    Console.WriteLine ("This will print");
```

### NOTE

In C#, predefined types (also referred to as built-in types) are recognized with a C# keyword. The `System` namespace in .NET Core contains many important types that are not predefined by C# (e.g., `DateTime`).

## Custom Type Examples

Just as we can build complex functions from simple functions, we can build complex types from primitive types. In this next example, we define a custom type named `UnitConverter`—a class that serves as a blueprint for unit conversions:

```
using System;

public class UnitConverter
{
    int ratio;                                // Field
```

```

    public UnitConverter (int unitRatio) {ratio = unitRatio; } // 
Constructor
    public int Convert (int unit)    {return unit * ratio; } // Method
}

class Test
{
    static void Main()
{
    UnitConverter feetToInchesConverter = new UnitConverter (12);
    UnitConverter milesToFeetConverter = new UnitConverter (5280);

    Console.WriteLine (feetToInchesConverter.Convert(30));      // 360
    Console.WriteLine (feetToInchesConverter.Convert(100));     // 1200
    Console.WriteLine (feetToInchesConverter.Convert(
                      milesToFeetConverter.Convert(1)));   // 63360
}
}

```

## MEMBERS OF A TYPE

A type contains *data members* and *function members*. The data member of `UnitConverter` is the *field* called `ratio`. The function members of `UnitConverter` are the `Convert` method and the `UnitConverter`'s *constructor*.

## SYMMETRY OF PREDEFINED TYPES AND CUSTOM TYPES

A beautiful aspect of C# is that predefined types and custom types have few differences. The predefined `int` type serves as a blueprint for integers. It holds data—32 bits—and provides function members that use that data, such as `ToString`. Similarly, our custom `UnitConverter` type acts as a blueprint for unit conversions. It holds data—the ratio—and provides function members to use that data.

## CONSTRUCTORS AND INSTANTIATION

Data is created by *instantiating* a type. Predefined types can be instantiated simply by using a literal such as 12 or "Hello world". The `new` operator creates instances of a custom type. We created and declared an instance of the `UnitConverter` type with this statement:

```
UnitConverter feetToInchesConverter = new UnitConverter (12);
```

Immediately after the `new` operator instantiates an object, the object's *constructor* is called to perform initialization. A constructor is defined like a method, except that the method name and return type are reduced to the name of the enclosing type:

```
public class UnitConverter
{
    ...
    public UnitConverter (int unitRatio) { ratio = unitRatio;
}
    ...
}
```

## INSTANCE VERSUS STATIC MEMBERS

The data members and function members that operate on the *instance* of the type are called instance members. The `UnitConverter`'s `Convert` method and the `int`'s `ToString` method are examples of instance members. By default, members are instance members.

Data members and function members that don't operate on the instance of the type but rather on the type itself must be marked as

`static`. The `Test.Main` and `Console.WriteLine` methods are static methods. The `Console` class is actually a *static class*, which means that *all* of its members are static. You never actually create instances of a `Console`—one console is shared across the entire application.

Let's contrast instance from static members. In the following code, the instance field `Name` pertains to an instance of a particular `Panda`, whereas `Population` pertains to the set of all `Panda` instances:

```
public class Panda
{
    public string Name;           // Instance field
    public static int Population; // Static field

    public Panda (string n)      // Constructor
    {
        Name = n;               // Assign the instance field
        Population = Population + 1; // Increment the static Population
        field
    }
}
```

The following code creates two instances of the `Panda`, prints their names, and then prints the total population:

```
using System;

class Test
{
    static void Main()
    {
        Panda p1 = new Panda ("Pan Dee");
        Panda p2 = new Panda ("Pan Dah");
```

```
        Console.WriteLine (p1.Name);      // Pan Dee
        Console.WriteLine (p2.Name);      // Pan Dah

        Console.WriteLine (Panda.Population); // 2
    }
}
```

Attempting to evaluate `p1.Population` or `Panda.Name` will generate a compile-time error.

## THE PUBLIC KEYWORD

The `public` keyword exposes members to other classes. In this example, if the `Name` field in `Panda` was not marked as public, it would be private and the `Test` class could not access it. Marking a member `public` is how a type communicates: “Here is what I want other types to see—everything else is my own private implementation details.” In object-oriented terms, we say that the public members *encapsulate* the private members of the class.

## Conversions

C# can convert between instances of compatible types. A conversion always creates a new value from an existing one. Conversions can be either *implicit* or *explicit*: implicit conversions happen automatically, and explicit conversions require a *cast*. In the following example, we *implicitly* convert an `int` to a `long` type (which has twice the bit capacity of an `int`), and we *explicitly* cast an `int` to a `short` type (which has half the bit capacity of an `int`):

```
int x = 12345;      // int is a 32-bit integer
```

```
long y = x;           // Implicit conversion to 64-bit integer
short z = (short)x;  // Explicit conversion to 16-bit integer
```

Implicit conversions are allowed when both of the following are true:

- The compiler can guarantee that they will always succeed.
- No information is lost in conversion.<sup>1</sup>

Conversely, *explicit* conversions are required when one of the following is true:

- The compiler cannot guarantee that they will always succeed.
- Information might be lost during conversion.

(If the compiler can determine that a conversion will *always* fail, both kinds of conversion are prohibited. Conversions that involve generics can also fail in certain conditions—see “[Type Parameters and Conversions](#)” in Chapter 3.)

### NOTE

The *numeric conversions* that we just saw are built into the language. C# also supports *reference conversions* and *boxing conversions* (see Chapter 3) as well as *custom conversions* (see “[Operator Overloading](#)” in Chapter 4). The compiler doesn’t enforce the aforementioned rules with custom conversions, so it’s possible for badly designed types to behave otherwise.

## Value Types versus Reference Types

All C# types fall into the following categories:

- Value types
- Reference types
- Generic type parameters
- Pointer types

### NOTE

In this section, we describe value types and reference types. We cover generic type parameters in “Generics” in Chapter 3, and pointer types in “Unsafe Code and Pointers” in Chapter 4.

*Value types* comprise most built-in types (specifically, all numeric types, the `char` type, and the `bool` type) as well as custom `struct` and `enum` types.

*Reference types* comprise all class, array, delegate, and interface types. (This includes the predefined `string` type.)

The fundamental difference between value types and reference types is how they are handled in memory.

## VALUE TYPES

The content of a *value-type* variable or constant is simply a value. For example, the content of the built-in value type, `int`, is 32 bits of data.

You can define a custom value type with the `struct` keyword (see Figure 2-1):

```
public struct Point { public int X; public int Y; }
```

or more tersely:

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

## Point struct



Figure 2-1. A value-type instance in memory

The assignment of a value-type instance always *copies* the instance; for example:

```
static void Main()
{
    Point p1 = new Point();
    p1.X = 7;

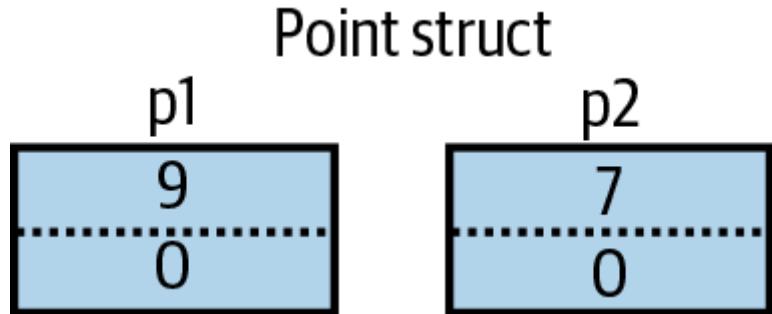
    Point p2 = p1;           // Assignment causes copy

    Console.WriteLine (p1.X); // 7
    Console.WriteLine (p2.X); // 7

    p1.X = 9;               // Change p1.X

    Console.WriteLine (p1.X); // 9
    Console.WriteLine (p2.X); // 7
}
```

Figure 2-2 shows that `p1` and `p2` have independent storage.



*Figure 2-2. Assignment copies a value-type instance*

## REFERENCE TYPES

A reference type is more complex than a value type, having two parts: an *object* and the *reference* to that object. The content of a reference-type variable or constant is a reference to an object that contains the value. Here is the `Point` type from our previous example rewritten as a class rather than a `struct` (shown in Figure 2-3):

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

## Point class

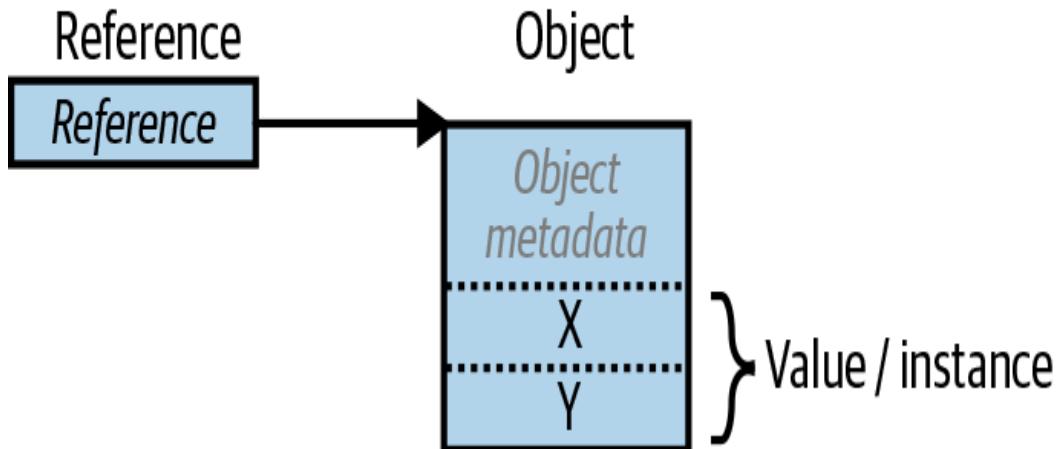


Figure 2-3. A reference-type instance in memory

Assigning a reference-type variable copies the reference, not the object instance. This allows multiple variables to refer to the same object—something not ordinarily possible with value types. If we repeat the previous example, but with `Point` now a class, an operation to `p1` affects `p2`:

```
static void Main()
{
    Point p1 = new Point();
    p1.X = 7;

    Point p2 = p1;           // Copies p1 reference

    Console.WriteLine (p1.X); // 7
    Console.WriteLine (p2.X); // 7

    p1.X = 9;               // Change p1.X

    Console.WriteLine (p1.X); // 9
```

```
    Console.WriteLine (p2.X); // 9  
}
```

Figure 2-4 shows that p1 and p2 are two references that point to the same object.

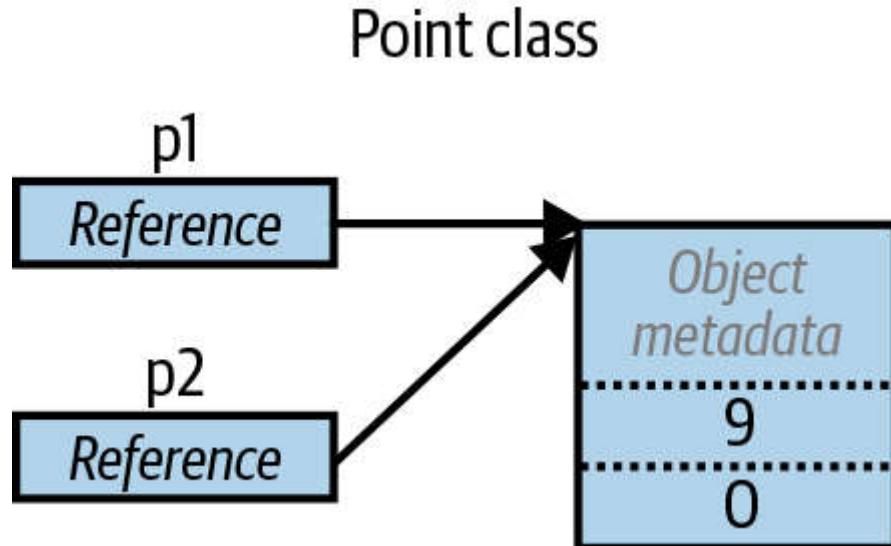


Figure 2-4. Assignment copies a reference

## NULL

A reference can be assigned the literal `null`, indicating that the reference points to no object:

```
class Point {...}  
...  
  
Point p = null;  
Console.WriteLine (p == null); // True  
  
// The following line generates a runtime error  
// (a NullReferenceException is thrown):  
Console.WriteLine (p.X);
```

## NOTE

C# 8 introduces a new feature to reduce accidental `NullReferenceException` errors. For more on this, see “[Nullable Reference Types \(C# 8\)](#)” in [Chapter 4](#).

In contrast, a value type cannot ordinarily have a null value:

```
struct Point {...}  
...  
  
Point p = null; // Compile-time error  
int x = null; // Compile-time error
```

## NOTE

C# also has a construct called *nullable value types* for representing value-type nulls. For more information, see “[Nullable Reference Types \(C# 8\)](#)” in [Chapter 4](#).

## STORAGE OVERHEAD

Value-type instances occupy precisely the memory required to store their fields. In this example, `Point` takes eight bytes of memory:

```
struct Point  
{  
    int x; // 4 bytes  
    int y; // 4 bytes  
}
```

## NOTE

Technically, the CLR positions fields within the type at an address that's a multiple of the fields' size (up to a maximum of eight bytes). Thus, the following actually consumes 16 bytes of memory (with the seven bytes following the first field “wasted”):

```
struct A { byte b; long l; }
```

You can override this behavior by applying the `StructLayout` attribute (see “[Mapping a Struct to Unmanaged Memory](#)” in Chapter 25).

Reference types require separate allocations of memory for the reference and object. The object consumes as many bytes as its fields, plus additional administrative overhead. The precise overhead is intrinsically private to the implementation of the .NET runtime, but at minimum, the overhead is eight bytes, used to store a key to the object’s type as well as temporary information such as its lock state for multithreading and a flag to indicate whether it has been fixed from movement by the garbage collector. Each reference to an object requires an extra four or eight bytes, depending on whether the .NET runtime is running on a 32- or 64-bit platform.

## Predefined Type Taxonomy

The predefined types in C# are as follows:

### *Value types*

- Numeric

- Signed integer (`sbyte`, `short`, `int`, `long`)
- Unsigned integer (`byte`, `ushort`, `uint`, `ulong`)
- Real number (`float`, `double`, `decimal`)
- Logical (`bool`)
- Character (`char`)

### *Reference types*

- String (`string`)
- Object (`object`)

Predefined types in C# alias .NET Core types in the `System` namespace. There is only a syntactic difference between these two statements:

```
int i = 5;
System.Int32 i = 5;
```

The set of predefined *value* types excluding `decimal` are known as *primitive types* in the CLR. Primitive types are so called because they are supported directly via instructions in compiled code, and this usually translates to direct support on the underlying processor; for example:

```
// Underlying hexadecimal representation
int i = 7;           // 0x7
```

```
bool b = true;      // 0x1
char c = 'A';       // 0x41
float f = 0.5f;     // uses IEEE floating-point encoding
```

The `System.IntPtr` and `System.UIntPtr` types are also primitive (see Chapter 25).

## Numeric Types

C# has the predefined numeric types shown in Table 2-1.

*T*  
*a*  
*b*  
*l*  
*e*  
*2*  
*-*  
*1*  
*.*  
*P*  
*r*  
*e*  
*d*  
*e*  
*f*  
*i*  
*n*  
*e*  
*d*  
*n*  
*u*  
*m*

e  
r  
i  
c  
t  
y  
p  
e  
s  
i  
n  
C  
#

C# type	System type	Suffix	Size	Range
---------	-------------	--------	------	-------

### Integral—signed

sbyte	SByte		8 bits	- $2^7$ to $2^7-1$
short	Int16		16 bits	- $2^{15}$ to $2^{15}-1$
int	Int32		32 bits	- $2^{31}$ to $2^{31}-1$
long	Int64	L	64 bits	- $2^{63}$ to $2^{63}-1$

### Integral—unsigned

byte	Byte		8 bits	0 to $2^8-1$
ushort	UInt16		16 bits	0 to $2^{16}-1$
uint	UInt32	U	32 bits	0 to $2^{32}-1$
ulong	UInt64	UL	64 bits	0 to $2^{64}-1$

### Real

<code>float</code>	<code>Single</code>	<code>F</code>	32 bits	$\pm (\sim 10^{-45} \text{ to } 10^{38})$
<code>double</code>	<code>Double</code>	<code>D</code>	64 bits	$\pm (\sim 10^{-324} \text{ to } 10^{308})$
<code>decimal</code>	<code>Decimal</code>	<code>M</code>	128 bits	$\pm (\sim 10^{-28} \text{ to } 10^{28})$

Of the *integral* types, `int` and `long` are first-class citizens and are favored by both C# and the runtime. The other integral types are typically used for interoperability or when space efficiency is paramount.

Of the *real* number types, `float` and `double` are called *floating-point types*<sup>2</sup> and are typically used for scientific and graphical calculations. The `decimal` type is typically used for financial calculations, for which base-10-accurate arithmetic and high precision are required.

## Numeric Literals

*Integral-type literals* can use decimal or hexadecimal notation; hexadecimal is denoted with the `0x` prefix; for example:

```
int x = 127;
long y = 0x7F;
```

From C# 7, you can insert an underscore anywhere within a numeric literal to make it more readable:

```
int million = 1_000_000;
```

C# 7 and above also lets you specify numbers in binary with the `0b` prefix:

```
var b = 0b1010_1011_1100_1101_1110_1111;
```

*Real literals* can use decimal and/or exponential notation:

```
double d = 1.5;
double million = 1E06;
```

## NUMERIC LITERAL TYPE INFERENCE

By default, the compiler *infers* a numeric literal to be either a `double` or an integral type:

- If the literal contains a decimal point or the exponential symbol (E), it is a `double`.
- Otherwise, the literal's type is the first type in this list that can fit the literal's value: `int`, `uint`, `long`, and `ulong`.

For example:

```
Console.WriteLine ( 1.0.GetType()); // Double (double)
Console.WriteLine ( 1E06.GetType()); // Double (double)
Console.WriteLine ( 1.GetType()); // Int32 (int)
Console.WriteLine ( 0xF0000000.GetType()); // UInt32 (uint)
Console.WriteLine (0x100000000.GetType()); // Int64 (long)
```

## NUMERIC SUFFIXES

*Numeric suffixes* explicitly define the type of a literal. Suffixes can be either lowercase or uppercase, and are as follows:

Category	C# type	Example
F	float	float f = 1.0F;
D	double	double d = 1D;
M	decimal	decimal d = 1.0M;
U	uint	uint i = 1U;
L	long	long i = 1L;
UL	ulong	ulong i = 1UL;

The suffixes U and L are rarely necessary because the `uint`, `long`, and `ulong` types can nearly always be either *inferred* or *implicitly converted* from `int`:

```
long i = 5;      // Implicit lossless conversion from int literal to long
```

The D suffix is technically redundant in that all literals with a decimal point are inferred to be `double`. And you can always add a decimal point to a numeric literal:

```
double x = 4.0;
```

The F and M suffixes are the most useful and should always be applied when specifying `float` or `decimal` literals. Without the F suffix, the following line would not compile, because 4.5 would be inferred to be of type `double`, which has no implicit conversion to `float`:

```
float f = 4.5F;
```

The same principle is true for a decimal literal:

```
decimal d = -1.23M;      // Will not compile without the M suffix.
```

We describe the semantics of numeric conversions in detail in the following section.

## Numeric Conversions

### CONVERTING BETWEEN INTEGRAL TYPES

Integral type conversions are *implicit* when the destination type can represent every possible value of the source type. Otherwise, an *explicit* conversion is required; for example:

```
int x = 12345;          // int is a 32-bit integer
long y = x;             // Implicit conversion to 64-bit integral type
short z = (short)x;    // Explicit conversion to 16-bit integral type
```

### CONVERTING BETWEEN FLOATING-POINT TYPES

A `float` can be implicitly converted to a `double` given that a `double` can represent every possible value of a `float`. The reverse

conversion must be explicit.

## CONVERTING BETWEEN FLOATING-POINT AND INTEGRAL TYPES

All integral types can be implicitly converted to all floating-point types:

```
int i = 1;  
float f = i;
```

The reverse conversion must be explicit:

```
int i2 = (int)f;
```

### NOTE

When you cast from a floating-point number to an integral type, any fractional portion is truncated; no rounding is performed. The static class `System.Convert` provides methods that round while converting between various numeric types (see Chapter 6).

Implicitly converting a large integral type to a floating-point type preserves *magnitude* but can occasionally lose *precision*. This is because floating-point types always have more magnitude than integral types, but can have less precision. Rewriting our example with a larger number demonstrates this:

```
int i1 = 100000001;
```

```
float f = i1;           // Magnitude preserved, precision lost
int i2 = (int)f;        // 100000000
```

## DECIMAL CONVERSIONS

All integral types can be implicitly converted to the decimal type given that a decimal can represent every possible C# integral-type value. All other numeric conversions to and from a decimal type must be explicit because they introduce the possibility of either a value being out of range or precision being lost.

## Arithmetic Operators

The arithmetic operators (+, -, \*, /, %) are defined for all numeric types except the 8- and 16-bit integral types:

+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Remainder after division

## Increment and Decrement Operators

The increment and decrement operators (++, --, respectively) increment and decrement numeric types by 1. The operator can either follow or precede the variable, depending on whether you want its value *before* or *after* the increment/decrement; for example:

```
int x = 0, y = 0;
Console.WriteLine (x++);    // Outputs 0; x is now 1
Console.WriteLine (++y);    // Outputs 1; y is now 1
```

## Specialized Operations on Integral Types

The *integral types* are `int`, `uint`, `long`, `ulong`, `short`, `ushort`, `byte`, and `sbyte`.

### DIVISION

Division operations on integral types always truncate remainders (round toward zero). Dividing by a variable whose value is zero generates a runtime error (a `DivideByZeroException`):

```
int a = 2 / 3;      // 0

int b = 0;
int c = 5 / b;      // throws DivideByZeroException
```

Dividing by the *literal* or *constant* 0 generates a compile-time error.

### OVERFLOW

At runtime, arithmetic operations on integral types can overflow. By default, this happens silently—no exception is thrown, and the result exhibits “wraparound” behavior, as though the computation were done on a larger integer type and the extra significant bits discarded. For example, decrementing the minimum possible `int` value results in the maximum possible `int` value:

```
int a = int.MinValue;
a--;
Console.WriteLine (a == int.MaxValue); // True
```

### OVERFLOW CHECK OPERATORS

The **checked** operator instructs the runtime to generate an **OverflowException** rather than overflowing silently when an integral-type expression or statement exceeds the arithmetic limits of that type. The **checked** operator affects expressions with the `++`, `--`, `+`, `-` (binary and unary), `*`, `/`, and explicit conversion operators between integral types. Overflow checking incurs a small performance cost.

### NOTE

The **checked** operator has no effect on the **double** and **float** types (which overflow to special “infinite” values, as you’ll see soon) and no effect on the **decimal** type (which is always checked).

You can use **checked** around either an expression or a statement block:

```
int a = 1000000;
int b = 1000000;

int c = checked (a * b);      // Checks just the expression.

checked                      // Checks all expressions
{
    ...                         // in statement block.
    c = a * b;
    ...
}
```

You can make arithmetic overflow checking the default for all expressions in a program by selecting the *checked* option at the project level (in Visual Studio, go to Advanced Build Settings). If you then need to disable overflow checking just for specific expressions or statements, you can do so with the **unchecked** operator. For example, the following code will not throw exceptions—even if the project’s *checked* option is selected:

```
int x = int.MaxValue;
int y = unchecked (x + 1);
unchecked { int z = x + 1; }
```

## OVERFLOW CHECKING FOR CONSTANT EXPRESSIONS

Regardless of the “checked” project setting, expressions evaluated at compile time are always overflow-checked—unless you apply the **unchecked** operator:

```
int x = int.MaxValue + 1;           // Compile-time error
int y = unchecked (int.MaxValue + 1); // No errors
```

## BITWISE OPERATORS

C# supports the following bitwise operators:

Operator	Meaning	Sample expression	Result
----------	---------	-------------------	--------

<code>~</code>	Complement	<code>~0xfU</code>	<code>0xffffffffU</code>
<code>&amp;</code>	And	<code>0xf0 &amp; 0x33</code>	<code>0x30</code>
<code> </code>	Or	<code>0xf0   0x33</code>	<code>0xf3</code>
<code>^</code>	Exclusive Or	<code>0xff00 ^ 0x0ff0</code>	<code>0xf0f0</code>
<code>&lt;&lt;</code>	Shift left	<code>0x20 &lt;&lt; 2</code>	<code>0x80</code>
<code>&gt;&gt;</code>	Shift right	<code>0x20 &gt;&gt; 1</code>	<code>0x10</code>

## 8- and 16-Bit Integral Types

The 8- and 16-bit integral types are `byte`, `sbyte`, `short`, and `ushort`. These types lack their own arithmetic operators, so C# implicitly converts them to larger types as required. This can cause a compile-time error when trying to assign the result back to a small integral type:

```
short x = 1, y = 1;
short z = x + y;           // Compile-time error
```

In this case, `x` and `y` are implicitly converted to `int` so that the addition can be performed. This means that the result is also an `int`, which cannot be implicitly cast back to a `short` (because it could cause loss of data). To make this compile, we must add an explicit cast:

```
short z = (short) (x + y); // OK
```

## Special Float and Double Values

Unlike integral types, floating-point types have values that certain operations treat specially. These special values are NaN (Not a Number),  $+\infty$ ,  $-\infty$ , and  $-0$ . The `float` and `double` classes have constants for NaN,  $+\infty$ , and  $-\infty$ , as well as other values (`MaxValue`, `MinValue`, and `Epsilon`); for example:

```
Console.WriteLine (double.NegativeInfinity); // -Infinity
```

The constants that represent special values for `double` and `float` are as follows:

Special value	Double constant	Float constant
---------------	-----------------	----------------

NaN	<code>double.NaN</code>	<code>float.NaN</code>
$+\infty$	<code>double.PositiveInfinity</code>	<code>float.PositiveInfinity</code>
$-\infty$	<code>double.NegativeInfinity</code>	<code>float.NegativeInfinity</code>
$-0$	<code>-0.0</code>	<code>-0.0f</code>

Dividing a nonzero number by zero results in an infinite value:

```
Console.WriteLine ( 1.0 / 0.0); // Infinity
Console.WriteLine (-1.0 / 0.0); // -Infinity
Console.WriteLine ( 1.0 / -0.0); // -Infinity
Console.WriteLine (-1.0 / -0.0); // Infinity
```

Dividing zero by zero, or subtracting infinity from infinity, results in a NaN:

```
Console.WriteLine ( 0.0 / 0.0);           // NaN
Console.WriteLine ((1.0 / 0.0) - (1.0 / 0.0)); // NaN
```

When using `==`, a NaN value is never equal to another value, even another NaN value:

```
Console.WriteLine (0.0 / 0.0 == double.NaN); // False
```

To test whether a value is NaN, you must use the `float.IsNaN` or `double.IsNaN` method:

```
Console.WriteLine (double.IsNaN (0.0 / 0.0)); // True
```

When using `object.Equals`, however, two NaN values are equal:

```
Console.WriteLine (object.Equals (0.0 / 0.0, double.NaN)); // True
```

### NOTE

Nans are sometimes useful in representing special values. In Windows Presentation Foundation (WPF), `double.NaN` represents a measurement whose value is “Automatic”. Another way to represent such a value is with a nullable type ([Chapter 4](#)); another is with a custom struct that wraps a numeric type and adds an additional field ([Chapter 3](#)).

`float` and `double` follow the specification of the IEEE 754 format types, supported natively by almost all processors. You can find detailed information on the behavior of these types on the [IEEE website](#).

## double Versus decimal

`double` is useful for scientific computations (such as computing spatial coordinates). `decimal` is useful for financial computations and values that are *man-made* rather than the result of real-world measurements. Here's a summary of the differences.

Category	double	decimal
Internal representation	Base 2	Base 10
Decimal precision	15–16 significant figures	28–29 significant figures
Range	$\pm(\sim 10^{-324} \text{ to } \sim 10^{308})$	$\pm(\sim 10^{-28} \text{ to } \sim 10^{28})$
Special values	+0, -0, $+\infty$ , $-\infty$ , and NaN	None
Speed	Native to processor	Non-native to processor (about 10 times slower than <code>double</code> )

## Real-Number Rounding Errors

`float` and `double` internally represent numbers in base 2. For this reason, only numbers expressible in base 2 are represented precisely. Practically, this means most literals with a fractional component (which are in base 10) will not be represented precisely; for example:

```
float tenth = 0.1f;                                // Not quite 0.1
float one   = 1f;
Console.WriteLine (one - tenth * 10f);    // -1.490116E-08
```

This is why `float` and `double` are bad for financial calculations. In contrast, `decimal` works in base 10 and so can precisely represent numbers expressible in base 10 (as well as its factors, base 2 and base 5). Because real literals are in base 10, `decimal` can precisely represent numbers such as 0.1. However, neither `double` nor `decimal` can precisely represent a fractional number whose base 10 representation is recurring:

```
decimal m = 1M / 6M;                            // 0.166666666666666666666666667M
double   d = 1.0 / 6.0;                          // 0.1666666666666666
```

This leads to accumulated rounding errors:

```
decimal notQuiteWholeM = m+m+m+m+m+m;  //
1.000000000000000000000000000000002M
double   notQuiteWholeD = d+d+d+d+d+d;  // 0.99999999999999989
```

which breaks equality and comparison operations:

```
Console.WriteLine (notQuiteWholeM == 1M);    // False
```

```
Console.WriteLine (notQuiteWholeD < 1.0); // True
```

## Boolean Type and Operators

C#'s `bool` type (aliasing the `System.Boolean` type) is a logical value that can be assigned the literal `true` or `false`.

Although a Boolean value requires only one bit of storage, the runtime will use one byte of memory because this is the minimum chunk that the runtime and processor can efficiently work with. To avoid space inefficiency in the case of arrays, .NET provides a `BitArray` class in the `System.Collections` namespace that is designed to use just one bit per Boolean value.

### bool Conversions

No casting conversions can be made from the `bool` type to numeric types, or vice versa.

## Equality and Comparison Operators

`==` and `!=` test for equality and inequality of any type but always return a `bool` value.<sup>3</sup> Value types typically have a very simple notion of equality:

```
int x = 1;
int y = 2;
int z = 1;
Console.WriteLine (x == y);      // False
Console.WriteLine (x == z);      // True
```

For reference types, equality, by default, is based on *reference*, as opposed to the actual *value* of the underlying object (more on this in [Chapter 6](#)):

```
public class Dude
{
    public string Name;
    public Dude (string n) { Name = n; }
}
...
Dude d1 = new Dude ("John");
Dude d2 = new Dude ("John");
Console.WriteLine (d1 == d2);      // False
Dude d3 = d1;
Console.WriteLine (d1 == d3);      // True
```

The equality and comparison operators, `==`, `!=`, `<`, `>`, `>=`, and `<=`, work for all numeric types, but you should use them with caution with real numbers (as we saw in “[Real-Number Rounding Errors](#)”). The comparison operators also work on `enum` type members by comparing their underlying integral-type values. We describe this in “[Enums](#)” in [Chapter 3](#).

We explain the equality and comparison operators in greater detail in “[Operator Overloading](#)” in [Chapter 4](#), and in “[Equality Comparison](#)” and “[Order Comparison](#)” in [Chapter 6](#).

## Conditional Operators

The `&&` and `||` operators test for *and* and *or* conditions. They are frequently used in conjunction with the `!` operator, which expresses *not*. In the following example, the `UseUmbrella` method returns `true`

if it's rainy or sunny (to protect us from the rain or the sun), as long as it's not also windy (umbrellas are useless in the wind):

```
static bool UseUmbrella (bool rainy, bool sunny, bool windy)
{
    return !windy && (rainy || sunny);
}
```

The `&&` and `||` operators *short-circuit* evaluation when possible. In the preceding example, if it is windy, the expression `(rainy || sunny)` is not even evaluated. Short-circuiting is essential in allowing expressions such as the following to run without throwing a `NullReferenceException`:

```
if (sb != null && sb.Length > 0) ...
```

The `&` and `|` operators also test for *and* and *or* conditions:

```
return !windy & (rainy | sunny);
```

The difference is that they *do not short-circuit*. For this reason, they are rarely used in place of conditional operators.

### NOTE

Unlike in C and C++, the `&` and `|` operators perform (non-short-circuiting) Boolean comparisons when applied to `bool` expressions. The `&` and `|` operators perform *bitwise* operations only when applied to numbers.

## CONDITIONAL OPERATOR (TERNARY OPERATOR)

The *conditional operator* (more commonly called the *ternary operator* because it's the only operator that takes three operands) has the form `q ? a : b`; thus, if condition `q` is true, `a` is evaluated, else `b` is evaluated:

```
static int Max (int a, int b)
{
    return (a > b) ? a : b;
}
```

The conditional operator is particularly useful in LINQ expressions (Chapter 8).

## Strings and Characters

C#'s `char` type (aliasing the `System.Char` type) represents a Unicode character and occupies 2 bytes (UTF-16). A `char` literal is specified within single quotes:

```
char c = 'A';           // Simple character
```

*Escape sequences* express characters that cannot be expressed or interpreted literally. An escape sequence is a backslash followed by a character with a special meaning; for example:

```
char newLine = '\n';
char backSlash = '\\';
```

Table 2-2 shows the escape sequence characters.

*T*

*a*

*b*

*l*

*e*

*2*

*-*

*2*

*.*

*E*

*s*

*c*

*a*

*p*

*e*

*s*

*e*

*q*

*u*

*e*

*n*

*c*

*e*

*c*

*h*

*a*

*r*

*a*

*c*

*t*

*e  
r  
s*

Char	Meaning	Value
\'	Single quote	0x0027
\"	Double quote	0x0022
\\"	Backslash	0x005C
\0	Null	0x0000
\a	Alert	0x0007
\b	Backspace	0x0008
\f	Form feed	0x000C
\n	Newline	0x000A
\r	Carriage return	0x000D
\t	Horizontal tab	0x0009
\v	Vertical tab	0x000B

The \u (or \x) escape sequence lets you specify any Unicode character via its four-digit hexadecimal code:

```
char copyrightSymbol = '\u00A9';
char omegaSymbol     = '\u03A9';
char newLine         = '\u000A';
```

## char Conversions

An implicit conversion from a `char` to a numeric type works for the numeric types that can accommodate an unsigned `short`. For other numeric types, an explicit conversion is required.

## String Type

C#'s string type (aliasing the `System.String` type, covered in depth in [Chapter 6](#)) represents an immutable (unmodifiable) sequence of Unicode characters. A string literal is specified within double quotes:

```
string a = "Heat";
```

### NOTE

`string` is a reference type rather than a value type. Its equality operators, however, follow value-type semantics:

```
string a = "test";
string b = "test";
Console.WriteLine(a == b); // True
```

The escape sequences that are valid for `char` literals also work inside strings:

```
string a = "Here's a tab:\t";
```

The cost of this is that whenever you need a literal backslash, you must write it twice:

```
string a1 = "\\\server\\fileshare\\helloworld.cs";
```

To avoid this problem, C# allows *verbatim* string literals. A verbatim string literal is prefixed with @ and does not support escape sequences. The following verbatim string is identical to the preceding one:

```
string a2 = @"\server\fileshare\helloworld.cs";
```

A verbatim string literal can also span multiple lines:

```
string escaped = "First Line\r\nSecond Line";
string verbatim = @"First Line
Second Line";

// True if your text editor uses CR-LF line separators:
Console.WriteLine (escaped == verbatim);
```

You can include the double-quote character in a verbatim literal by writing it twice:

```
string xml = @"<customer id=""123""></customer>";
```

## STRING CONCATENATION

The + operator concatenates two strings:

```
string s = "a" + "b";
```

One of the operands might be a nonstring value, in which case `ToString` is called on that value:

```
string s = "a" + 5; // a5
```

Using the `+` operator repeatedly to build up a string is inefficient: a better solution is to use the `System.Text.StringBuilder` type (described in [Chapter 6](#)).

## STRING INTERPOLATION

A string preceded with the `$` character is called an *interpolated string*. Interpolated strings can include expressions enclosed in braces:

```
int x = 4;
Console.Write($"A square has {x} sides"); // Prints: A square has 4
sides
```

Any valid C# expression of any type can appear within the braces, and C# will convert the expression to a string by calling its `ToString` method or equivalent. You can change the formatting by appending the expression with a colon and a *format string* (format strings are described in “[string.Format and composite format strings](#)” in [Chapter 6](#)):

```
string s = $"255 in hex is {byte.MaxValue:X2}"; // X2 = 2-digit
```

```
hexadecimal  
// Evaluates to "255 in hex is FF"
```

Interpolated strings must complete on a single line, unless you also specify the verbatim string operator:

```
int x = 2;  
// Note that $ must appear before @ prior to C# 8:  
string s = $$@"this spans {  
x} lines";
```

To include a brace literal in an interpolated string, repeat the desired brace character.

## STRING COMPARISONS

`string` does not support `<` and `>` operators for comparisons. You must use the `string`'s `CompareTo` method, described in [Chapter 6](#).

## Arrays

An array represents a fixed number of variables (called *elements*) of a particular type. The elements in an array are always stored in a contiguous block of memory, providing highly efficient access.

An array is denoted with square brackets after the element type:

```
char[] vowels = new char[5]; // Declare an array of 5 characters
```

Square brackets also *index* the array, accessing a particular element by position:

```
vowels[0] = 'a';
vowels[1] = 'e';
vowels[2] = 'i';
vowels[3] = 'o';
vowels[4] = 'u';
Console.WriteLine (vowels[1]);      // e
```

This prints “e” because array indexes start at 0. We can use a **for** loop statement to iterate through each element in the array. The **for** loop in this example cycles the integer **i** from 0 to 4:

```
for (int i = 0; i < vowels.Length; i++)
    Console.Write (vowels[i]);           // aeiou
```

The **Length** property of an array returns the number of elements in the array. After an array has been created, you cannot change its length. The **System.Collection** namespace and subnamespaces provide higher-level data structures, such as dynamically sized arrays and dictionaries.

An *array initialization expression* lets you declare and populate an array in a single step:

```
char[] vowels = new char[] {'a','e','i','o','u'};
```

or simply:

```
char[] vowels = {'a','e','i','o','u'};
```

All arrays inherit from the `System.Array` class, providing common services for all arrays. These members include methods to get and set elements regardless of the array type. We describe them in “[The Array Class](#)” in Chapter 7.

## Default Element Initialization

Creating an array always preinitializes the elements with default values. The default value for a type is the result of a bitwise zeroing of memory. For example, consider creating an array of integers. Because `int` is a value type, this allocates 1,000 integers in one contiguous block of memory. The default value for each element will be 0:

```
int[] a = new int[1000];
Console.Write (a[123]);           // 0
```

## VALUE TYPES VERSUS REFERENCE TYPES

Whether an array element type is a value type or a reference type has important performance implications. When the element type is a value type, each element value is allocated as part of the array, as shown here:

```
public struct Point { public int X, Y; }
...
Point[] a = new Point[1000];
int x = a[500].X;                 // 0
```

Had `Point` been a class, creating the array would have merely allocated 1,000 null references:

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

...
Point[] a = new Point[1000];
int x = a[500].X;                                // Runtime error,
NullReferenceException
```

To avoid this error, we must explicitly instantiate 1,000 `Points` after instantiating the array:

```
Point[] a = new Point[1000];
for (int i = 0; i < a.Length; i++) // Iterate i from 0 to 999
    a[i] = new Point();           // Set array element i with new point
```

An array *itself* is always a reference-type object, regardless of the element type. For instance, the following is legal:

```
int[] a = null;
```

## Indices and Ranges (C# 8)

C# 8 introduces *indices and ranges* to simplify working with elements or portions of an array.

## NOTE

Indices and ranges also work with the CLR types `Span<T>` and `ReadOnlySpan<T>` (see “`Span<T>` and `Memory<T>`” in Chapter 5).

You can also make your own types work with indices and ranges, by defining an indexer of type `Index` or `Range` (see “`Indexers`” in Chapter 3).

## INDICES

Indices let you refer to elements relative to the *end* of an array, with the `^` operator. `^1` refers to the last element, `^2` refers to the second-to-last element, and so on:

```
char[] vowels = new char[] {'a','e','i','o','u'};
char lastElement = vowels [^1];    // 'u'
char secondToLast = vowels [^2];   // 'o'
```

(`^0` equals the length of the array, so `vowels[^0]` generates an error.)

C# implements indices with the help of the `Index` type, so you can also do the following:

```
Index first = 0;
Index last = ^1;
char firstElement = vowels [first];    // 'a'
char lastElement = vowels [last];      // 'u'
```

## RANGES

Ranges let you “slice” an array by using the `..` operator:

```
char[] firstTwo = vowels [..2];    // 'a', 'e'  
char[] lastThree = vowels [2..];   // 'i', 'o', 'u'  
char[] middleOne = vowels [2..3];  // 'i'
```

The second number in the range is *exclusive*, so `..2` returns the elements *before* `vowels[2]`.

You can also use the `^` symbol in ranges. The following returns the last two characters:

```
char[] lastTwo = vowels [^2..];    // 'o', 'u'
```

C# implements ranges with the help of the `Range` type, so you can also do the following:

```
Range firstTwoRange = 0..2;  
char[] firstTwo = vowels [firstTwoRange]; // 'a', 'e'
```

## Multidimensional Arrays

Multidimensional arrays come in two varieties: *rectangular* and *jagged*. Rectangular arrays represent an  $n$ -dimensional block of memory, and jagged arrays are arrays of arrays.

### RECTANGULAR ARRAYS

Rectangular arrays are declared using commas to separate each dimension. The following declares a rectangular two-dimensional array for which the dimensions are 3 by 3:

```
int[,] matrix = new int[3,3];
```

The `GetLength` method of an array returns the length for a given dimension (starting at 0):

```
for (int i = 0; i < matrix.GetLength(0); i++)
    for (int j = 0; j < matrix.GetLength(1); j++)
        matrix[i,j] = i * 3 + j;
```

You can initialize a rectangular array with explicit values. The following code creates an array identical to the previous example:

```
int[,] matrix = new int[,]
{
    {0,1,2},
    {3,4,5},
    {6,7,8}
};
```

## JAGGED ARRAYS

Jagged arrays are declared using successive square brackets to represent each dimension. Here is an example of declaring a jagged two-dimensional array for which the outermost dimension is 3:

```
int[][] matrix = new int[3][];
```

## NOTE

Interestingly, this is `new int[3][]` and not `new int[][][3]`. Eric Lippert has written an excellent article on why this is so.

The inner dimensions aren't specified in the declaration because, unlike a rectangular array, each inner array can be an arbitrary length. Each inner array is implicitly initialized to null rather than an empty array. You must manually create each inner array:

```
for (int i = 0; i < matrix.Length; i++)
{
    matrix[i] = new int[3]; // Create inner array
    for (int j = 0; j < matrix[i].Length; j++)
        matrix[i][j] = i * 3 + j;
}
```

You can initialize a jagged array with explicit values. The following code creates an array identical to the previous example with an additional element at the end:

```
int[][] matrix = new int[][]
{
    new int[] {0,1,2},
    new int[] {3,4,5},
    new int[] {6,7,8,9}
};
```

## Simplified Array Initialization Expressions

There are two ways to shorten array initialization expressions. The first is to omit the `new` operator and type qualifications:

```
char[] vowels = {'a','e','i','o','u'};  
  
int[,] rectangularMatrix =  
{  
    {0,1,2},  
    {3,4,5},  
    {6,7,8}  
};  
  
int[][] jaggedMatrix =  
{  
    new int[] {0,1,2},  
    new int[] {3,4,5},  
    new int[] {6,7,8,9}  
};
```

The second approach is to use the `var` keyword, which instructs the compiler to implicitly type a local variable:

```
var i = 3;           // i is implicitly of type int  
var s = "sausage"; // s is implicitly of type string  
  
// Therefore:  
  
var rectMatrix = new int[,]      // rectMatrix is implicitly of type  
int[,]  
{  
    {0,1,2},  
    {3,4,5},  
    {6,7,8}  
};
```

```
var jaggedMat = new int[][] // jaggedMat is implicitly of type int[][]
[]
{
    new int[] {0,1,2},
    new int[] {3,4,5},
    new int[] {6,7,8,9}
};
```

Implicit typing can be taken one stage further with arrays: you can omit the type qualifier after the `new` keyword and have the compiler *infer* the array type:

```
var vowels = new[] {'a','e','i','o','u'}; // Compiler infers char[]
```

For this to work, the elements must all be implicitly convertible to a single type (and at least one of the elements must be of that type, and there must be exactly one best type), as in the following example:

```
var x = new[] {1,10000000000}; // all convertible to long
```

## Bounds Checking

All array indexing is bounds-checked by the runtime. An `IndexOutOfRangeException` is thrown if you use an invalid index:

```
int[] arr = new int[3];
arr[3] = 1; // IndexOutOfRangeException thrown
```

Array bounds checking is necessary for type safety and simplifies debugging.

## NOTE

Generally, the performance hit from bounds checking is minor, and the Just-In-Time (JIT) compiler can perform optimizations, such as determining in advance whether all indexes will be safe before entering a loop, thus avoiding a check on each iteration. In addition, C# provides “unsafe” code that can explicitly bypass bounds checking (see “Unsafe Code and Pointers” in Chapter 4).

# Variables and Parameters

A variable represents a storage location that has a modifiable value. A variable can be a *local variable*, *parameter* (*value*, *ref*, *out*, or *in*), *field* (*instance* or *static*), or *array element*.

## The Stack and the Heap

The stack and the heap are the places where variables reside. Each has very different lifetime semantics.

### STACK

The stack is a block of memory for storing local variables and parameters. The stack logically grows and shrinks as a method or function is entered and exited. Consider the following method (to avoid distraction, input argument checking is ignored):

```
static int Factorial (int x)
{
    if (x == 0) return 1;
    return x * Factorial (x-1);
}
```

This method is recursive, meaning that it calls itself. Each time the method is entered, a new `int` is allocated on the stack, and each time the method exits, the `int` is deallocated.

## HEAP

The heap is the memory in which *objects* (i.e., reference-type instances) reside. Whenever a new object is created, it is allocated on the heap, and a reference to that object is returned. During a program's execution, the heap begins filling up as new objects are created. The runtime has a garbage collector that periodically deallocates objects from the heap, so your program does not run out of memory. An object is eligible for deallocation as soon as it's not referenced by anything that's itself *alive*.

In the following example, we begin by creating a `StringBuilder` object referenced by the variable `ref1` and then write out its content. That `StringBuilder` object is then immediately eligible for garbage collection because nothing subsequently uses it.

Then, we create another `StringBuilder` referenced by variable `ref2` and copy that reference to `ref3`. Even though `ref2` is not used after that point, `ref3` keeps the same `StringBuilder` object alive—ensuring that it doesn't become eligible for collection until we've finished using `ref3`.

```
using System;
using System.Text;

class Test
```

```
{  
    static void Main()  
    {  
        StringBuilder ref1 = new StringBuilder ("object1");  
        Console.WriteLine (ref1);  
        // The StringBuilder referenced by ref1 is now eligible for GC.  
  
        StringBuilder ref2 = new StringBuilder ("object2");  
        StringBuilder ref3 = ref2;  
        // The StringBuilder referenced by ref2 is NOT yet eligible for GC.  
  
        Console.WriteLine (ref3);                // object2  
    }  
}
```

Value-type instances (and object references) live wherever the variable was declared. If the instance was declared as a field within a class type, or as an array element, that instance lives on the heap.

### NOTE

You can't explicitly delete objects in C#, as you can in C++. An unreferenced object is eventually collected by the garbage collector.

The heap also stores static fields. Unlike objects allocated on the heap (which can be garbage-collected), these live until the application domain is torn down.

## Definite Assignment

C# enforces a definite assignment policy. In practice, this means that outside of an `unsafe` context, it's impossible to access uninitialized

memory. Definite assignment has three implications:

- Local variables must be assigned a value before they can be read.
- Function arguments must be supplied when a method is called (unless marked as optional; see “[Optional parameters](#)”).
- All other variables (such as fields and array elements) are automatically initialized by the runtime.

For example, the following code results in a compile-time error:

```
static void Main()
{
    int x;
    Console.WriteLine (x);           // Compile-time error
}
```

Fields and array elements are automatically initialized with the default values for their type. The following code outputs `0` because array elements are implicitly assigned to their default values:

```
static void Main()
{
    int[] ints = new int[2];
    Console.WriteLine (ints[0]);     // 0
}
```

The following code outputs `0`, because fields are implicitly assigned a default value:

```
class Test
{
    static int x;
    static void Main() { Console.WriteLine (x); }    // 0
}
```

## Default Values

All type instances have a default value. The default value for the predefined types is the result of a bitwise zeroing of memory:

Type	Default value
All reference types	null
All numeric and enum types	0
char type	'\0'
bool type	false

You can obtain the default value for any type via the **default** keyword:

```
Console.WriteLine (default (decimal));    // 0
```

From C# 7.1, you can optionally omit the type when it can be inferred:

```
decimal d = default;
```

The default value in a custom value type (i.e., `struct`) is the same as the default value for each field defined by the custom type.

## Parameters

A method may have a sequence of parameters. Parameters define the set of arguments that must be provided for that method. In the following example, the method `Foo` has a single parameter named `p`, of type `int`:

```
static void Foo (int p)
{
    p = p + 1;                      // Increment p by 1
    Console.WriteLine (p);           // Write p to screen
}

static void Main()
{
    Foo (8);                      // Call Foo with an argument of 8
}
```

You can control how parameters are passed with the `ref`, `in`, and `out` modifiers:

Parameter modifier	Passed by	Variable must be definitely assigned
--------------------	-----------	--------------------------------------

(None)	Value	Going in
--------	-------	----------

<code>ref</code>	Reference	Going in
<code>in</code>	Reference (read-only)	Going in
<code>out</code>	Reference	Going out

## PASSING ARGUMENTS BY VALUE

By default, arguments in C# are *passed by value*, which is by far the most common case. This means that a copy of the value is created when passed to the method:

```
class Test
{
    static void Foo (int p)
    {
        p = p + 1;           // Increment p by 1
        Console.WriteLine (p); // Write p to screen
    }

    static void Main()
    {
        int x = 8;
        Foo (x);           // Make a copy of x
        Console.WriteLine (x); // x will still be 8
    }
}
```

Assigning `p` a new value does not change the contents of `x`, because `p` and `x` reside in different memory locations.

Passing a reference-type argument by value copies the *reference*, but not the object. In the following example, `Foo` sees the same

`StringBuilder` object that `Main` instantiated, but has an independent *reference* to it. In other words, `sb` and `fooSB` are separate variables that reference the same `StringBuilder` object:

```
class Test
{
    static void Foo (StringBuilder fooSB)
    {
        fooSB.Append ("test");
        fooSB = null;
    }

    static void Main()
    {
        StringBuilder sb = new StringBuilder();
        Foo (sb);
        Console.WriteLine (sb.ToString());    // test
    }
}
```

Because `fooSB` is a *copy* of a reference, setting it to `null` doesn't make `sb` null. (If, however, `fooSB` was declared and called with the `ref` modifier, `sb` *would* become null.)

## THE REF MODIFIER

To *pass by reference*, C# provides the `ref` parameter modifier. In the following example, `p` and `x` refer to the same memory locations:

```
class Test
{
    static void Foo (ref int p)
    {
        p = p + 1;           // Increment p by 1
```

```

        Console.WriteLine (p);    // Write p to screen
    }

    static void Main()
    {
        int x = 8;
        Foo (ref x);           // Ask Foo to deal directly with x
        Console.WriteLine (x);   // x is now 9
    }
}

```

Now assigning `p` a new value changes the contents of `x`. Notice how the `ref` modifier is required both when writing and when calling the method.<sup>4</sup> This makes it very clear what's going on.

The `ref` modifier is essential in implementing a swap method (in “Generics” in Chapter 3, we show how to write a swap method that works with any type):

```

class Test
{
    static void Swap (ref string a, ref string b)
    {
        string temp = a;
        a = b;
        b = temp;
    }

    static void Main()
    {
        string x = "Penn";
        string y = "Teller";
        Swap (ref x, ref y);
        Console.WriteLine (x);   // Teller
        Console.WriteLine (y);   // Penn
    }
}

```

```
    }  
}
```

### NOTE

A parameter can be passed by reference or by value, regardless of whether the parameter type is a reference type or a value type.

## THE OUT MODIFIER

An **out** argument is like a **ref** argument except for the following:

- It need not be assigned before going into the function.
- It must be assigned before it comes *out* of the function.

The **out** modifier is most commonly used to get multiple return values back from a method; for example:

```
class Test  
{  
    static void Split (string name, out string firstNames,  
                      out string lastName)  
    {  
        int i = name.LastIndexOf (' ');  
        firstNames = name.Substring (0, i);  
        lastName   = name.Substring (i + 1);  
    }  
  
    static void Main()  
    {  
        string a, b;  
        Split ("Stevie Ray Vaughan", out a, out b);  
    }  
}
```

```
    Console.WriteLine (a);           // Stevie Ray
    Console.WriteLine (b);           // Vaughan
}
}
```

Like a `ref` parameter, an `out` parameter is passed by reference.

## OUT VARIABLES AND DISCARDS

From C# 7, you can declare variables on the fly when calling methods with `out` parameters. We can shorten the `Main` method in our preceding example as follows:

```
static void Main()
{
    Split ("Stevie Ray Vaughan", out string a, out string b);
    Console.WriteLine (a);           // Stevie Ray
    Console.WriteLine (b);           // Vaughan
}
```

When calling methods with multiple `out` parameters, sometimes you're not interested in receiving values from all the parameters. In such cases, you can *discard* the ones in which you're not interested by using an underscore:

```
Split ("Stevie Ray Vaughan", out string a, out _); // Discard the 2nd
param
Console.WriteLine (a);
```

In this case, the compiler treats the underscore as a special symbol, called a *discard*. You can include multiple discards in a single call.

Assuming `SomeBigMethod` has been defined with seven **out** parameters, we can ignore all but the fourth, as follows:

```
SomeBigMethod (out _, out _, out _, out int x, out _, out _, out _);
```

For backward compatibility, this language feature will not take effect if a real underscore variable is in scope:

```
string _;
Split ("Stevie Ray Vaughan", out string a, out _);
Console.WriteLine (_);      // Vaughan
```

## IMPLICATIONS OF PASSING BY REFERENCE

When you pass an argument by reference, you alias the storage location of an existing variable rather than create a new storage location. In the following example, the variables `x` and `y` represent the same instance:

```
class Test
{
    static int x;

    static void Main() { Foo (out x); }

    static void Foo (out int y)
    {
        Console.WriteLine (x);                      // x is 0
        y = 1;                                     // Mutate y
        Console.WriteLine (x);                      // x is 1
    }
}
```

## THE IN MODIFIER

An `in` parameter is similar to a `ref` parameter except that the argument's value cannot be modified by the method (doing so generates a compile-time error). This modifier is most useful when passing a large value type to the method because it allows the compiler to avoid the overhead of copying the argument prior to passing it in while still protecting the original value from modification.

Overloading solely on the presence of `in` is permitted:

```
void Foo ( SomeBigStruct a) { ... }
void Foo (in SomeBigStruct a) { ... }
```

To call the second overload, the caller must use the `in` modifier:

```
SomeBigStruct x = ...;
Foo (x);      // Calls the first overload
Foo (in x);   // Calls the second overload
```

When there's no ambiguity:

```
void Bar (in SomeBigStruct a) { ... }
```

use of the `in` modifier is optional for the caller:

```
Bar (x);      // OK (calls the 'in' overload)
Bar (in x);   // OK (calls the 'in' overload)
```

To make this example meaningful, `SomeBigStruct` would be defined as a struct (see “[Structs](#)” in Chapter 3).

## THE PARAMS MODIFIER

You can specify the `params` parameter modifier on the last parameter of a method so that the method accepts any number of arguments of a particular type. The parameter type must be declared as an array, as shown in the following example:

```
class Test
{
    static int Sum (params int[] ints)
    {
        int sum = 0;
        for (int i = 0; i < ints.Length; i++)
            sum += ints[i];                                // Increase sum by ints[i]
        return sum;
    }

    static void Main()
    {
        int total = Sum (1, 2, 3, 4);
        Console.WriteLine (total);                      // 10
    }
}
```

You can also supply a `params` argument as an ordinary array. The first line in `Main` is semantically equivalent to this:

```
int total = Sum (new int[] { 1, 2, 3, 4 } );
```

## OPTIONAL PARAMETERS

Methods, constructors, and indexers (Chapter 3) can declare *optional parameters*. A parameter is optional if it specifies a *default value* in its declaration:

```
void Foo (int x = 23) { Console.WriteLine (x); }
```

You can omit optional parameters when calling the method:

```
Foo(); // 23
```

The *default argument* of 23 is actually *passed* to the optional parameter x—the compiler bakes the value 23 into the compiled code at the *calling* side. The preceding call to Foo is semantically identical to:

```
Foo (23);
```

because the compiler simply substitutes the default value of an optional parameter wherever it is used.

### NOTE

Adding an optional parameter to a public method that's called from another assembly requires recompilation of both assemblies—just as though the parameter were mandatory.

The default value of an optional parameter must be specified by a constant expression or a parameterless constructor of a value type.

Optional parameters cannot be marked with `ref` or `out`.

Mandatory parameters must occur *before* optional parameters in both the method declaration and the method call (the exception is with `params` arguments, which still always come last). In the following example, the explicit value of 1 is passed to `x`, and the default value of 0 is passed to `y`:

```
void Foo (int x = 0, int y = 0) { Console.WriteLine (x + ", " + y); }

void Test()
{
    Foo(1);      // 1, 0
}
```

To do the converse (pass a default value to `x` and an explicit value to `y`) you must combine optional parameters with *named arguments*.

## NAMED ARGUMENTS

Rather than identifying an argument by position, you can identify an argument by name:

```
void Foo (int x, int y) { Console.WriteLine (x + ", " + y); }

void Test()
{
    Foo (x:1, y:2);  // 1, 2
}
```

Named arguments can occur in any order. The following calls to `Foo` are semantically identical:

```
Foo (x:1, y:2);
Foo (y:2, x:1);
```

## NOTE

A subtle difference is that argument expressions are evaluated in the order in which they appear at the *calling* site. In general, this makes a difference only with interdependent side-effecting expressions such as the following, which writes 0, 1:

```
int a = 0;
Foo (y: ++a, x: --a); // ++a is evaluated first
```

Of course, you would almost certainly avoid writing such code in practice!

You can mix named and positional arguments:

```
Foo (1, y:2);
```

However, there is a restriction: positional arguments must come before named arguments unless they are used in the correct position. So, we could call `Foo` like this:

```
Foo (x:1, 2);           // OK. Arguments in the declared positions
```

but not like this:

```
Foo (y:2, 1);           // Compile-time error. y isn't in the first
```

```
position
```

Named arguments are particularly useful in conjunction with optional parameters. For instance, consider the following method:

```
void Bar (int a = 0, int b = 0, int c = 0, int d = 0) { ... }
```

We can call this supplying only a value for `d`, as follows:

```
Bar (d:3);
```

This is particularly useful when calling COM APIs, which we discuss in detail in [Chapter 25](#).

## Ref Locals

C# 7 added an esoteric feature, whereby you can define a local variable that *references* an element in an array or field in an object:

```
int[] numbers = { 0, 1, 2, 3, 4 };
ref int numRef = ref numbers [2];
```

In this example, `numRef` is a *reference* to `numbers[2]`. When we modify `numRef`, we modify the array element:

```
numRef *= 10;
Console.WriteLine (numRef);           // 20
Console.WriteLine (numbers [2]);     // 20
```

The target for a `ref` local must be an array element, field, or local variable; it cannot be a *property* (Chapter 3). *Ref locals* are intended for specialized micro-optimization scenarios and are typically used in conjunction with *ref returns*.

## Ref Returns

### NOTE

The `Span<T>` and `ReadOnlySpan<T>` types that we describe in Chapter 24 use `ref` returns to implement a highly efficient indexer. Outside such scenarios, `ref` returns are not commonly used; you can consider them a micro-optimization feature.

You can return a *ref local* from a method. This is called a *ref return*:

```
static string x = "Old Value";

static ref string GetX() => ref x;      // This method returns a ref

static void Main()
{
    ref string xRef = ref GetX();        // Assign result to a ref local
    xRef = "New Value";
    Console.WriteLine (x);            // New Value
}
```

If you omit the `ref` modifier on the calling side, it reverts to returning an ordinary value:

```
string localX = GetX(); // Legal: localX is an ordinary non-ref variable.
```

You also can use `ref` returns when defining a property or indexer:

```
static ref string Prop => ref x;
```

Such a property is implicitly writable, despite there being no `set` accessor:

```
Prop = "New Value";
```

You can prevent such modification by using `ref readonly`:

```
static ref readonly string Prop => ref x;
```

The `ref readonly` modifier prevents modification while still enabling the performance gain of returning by reference. The gain would be very small in this case, because `x` is of type `string` (a reference type): no matter how long the string, the only inefficiency that we can hope to avoid is the copying of a single 32- or 64-bit *reference*. Real gains can occur with custom value types (see “[Structs](#)” in Chapter 3), but only if the struct is marked as `readonly` (otherwise, the compiler will perform a defensive copy).

Attempting to define an explicit `set` accessor on a `ref` return property or indexer is illegal.

## var—Implicitly Typed Local Variables

It is often the case that you declare and initialize a variable in one step. If the compiler is able to infer the type from the initialization expression, you can use the keyword `var` (introduced in C# 3.0) in place of the type declaration; for example:

```
var x = "hello";
var y = new System.Text.StringBuilder();
var z = (float)Math.PI;
```

This is precisely equivalent to the following:

```
string x = "hello";
System.Text.StringBuilder y = new System.Text.StringBuilder();
float z = (float)Math.PI;
```

Because of this direct equivalence, implicitly typed variables are statically typed. For example, the following generates a compile-time error:

```
var x = 5;
x = "hello";    // Compile-time error; x is of type int
```

## NOTE

`var` can decrease code readability in the case when *you can't deduce the type purely by looking at the variable declaration*. For example:

```
Random r = new Random();
var x = r.Next();
```

What type is `x`?

In “Anonymous Types” in [Chapter 4](#), we will describe a scenario in which the use of `var` is mandatory.

## Expressions and Operators

An *expression* essentially denotes a value. The simplest kinds of expressions are constants and variables. Expressions can be transformed and combined using operators. An *operator* takes one or more input *operands* to output a new expression.

Here is an example of a *constant expression*:

12

We can use the `*` operator to combine two operands (the literal expressions `12` and `30`), as follows:

12 \* 30

We can build complex expressions because an operand can itself be an expression, such as the operand `(12 * 30)` in the following example:

```
1 + (12 * 30)
```

Operators in C# can be classed as *unary*, *binary*, or *ternary*, depending on the number of operands they work on (one, two, or three). The binary operators always use *infix* notation, in which the operator is placed *between* the two operands.

## Primary Expressions

Primary expressions include expressions composed of operators that are intrinsic to the basic plumbing of the language. Here is an example:

```
Math.Log (1)
```

This expression is composed of two primary expressions. The first expression performs a member lookup (with the `.` operator), and the second expression performs a method call (with the `()` operator).

## Void Expressions

A void expression is an expression that has no value, such as this:

```
Console.WriteLine (1)
```

Because it has no value, you cannot use a void expression as an operand to build more complex expressions:

```
1 + Console.WriteLine (1)      // Compile-time error
```

## Assignment Expressions

An assignment expression uses the = operator to assign the result of another expression to a variable; for example:

```
x = x * 5
```

An assignment expression is not a void expression—it has a value of whatever was assigned, and so can be incorporated into another expression. In the following example, the expression assigns 2 to x and 10 to y:

```
y = 5 * (x = 2)
```

You can use this style of expression to initialize multiple values:

```
a = b = c = d = 0
```

The *compound assignment operators* are syntactic shortcuts that combine assignment with another operator:

```
x *= 2    // equivalent to x = x * 2
x <= 1    // equivalent to x = x <= 1
```

(A subtle exception to this rule is with *events*, which we describe in Chapter 4: the `+=` and `-=` operators here are treated specially and map to the event's `add` and `remove` accessors.)

## Operator Precedence and Associativity

When an expression contains multiple operators, *precedence* and *associativity* determine the order of their evaluation. Operators with higher precedence execute before operators of lower precedence. If the operators have the same precedence, the operator's associativity determines the order of evaluation.

### PRECEDENCE

The following expression:

```
1 + 2 * 3
```

is evaluated as follows because `*` has a higher precedence than `+`:

```
1 + (2 * 3)
```

### LEFT-ASSOCIATIVE OPERATORS

Binary operators (except for assignment, lambda, and null-coalescing operators) are *left-associative*; in other words, they are evaluated from left to right. For example, the following expression:

```
8 / 4 / 2
```

is evaluated as follows:

```
( 8 / 4 ) / 2    // 1
```

You can insert parentheses to change the actual order of evaluation:

```
8 / ( 4 / 2 )    // 4
```

## RIGHT-ASSOCIATIVE OPERATORS

The *assignment operators*, as well as the lambda, null coalescing, and conditional operators, are *right-associative*; in other words, they are evaluated from right to left. Right associativity allows multiple assignments such as the following to compile:

```
x = y = 3;
```

This first assigns 3 to y and then assigns the result of that expression (3) to x.

## Operator Table

Table 2-3 lists C#'s operators in order of precedence. Operators in the same category have the same precedence. We explain user-overloadable operators in “Operator Overloading” in Chapter 4.

T  
a  
b

l  
e  
2  
-  
3  
.C  
#  
o  
p  
e  
r  
a  
t  
o  
r  
s  
(  
c  
a  
t  
e  
g  
o  
r  
i  
e  
s  
i  
n  
o  
r  
d  
e  
r  
o  
f

*p  
r  
e  
c  
e  
d  
e  
n  
c  
e  
)*

Category	Operator symbol	Operator name	Example	User-overloadable
Primary	.	Member access	x.y	No
	? . and ? [ ]	Null-conditional	x?.y or x?[0]	No
	-> (unsafe)	Pointer to struct	x->y	No
	()	Function call	x()	No
	[ ]	Array/index	a[x]	Via index er
	++	Post-increment	x++	Yes
	--	Post-decrement	x--	Yes
	new	Create instance	new Foo()	No
	stack	Unsafe stack	stackalloc(10)	No

	<code>alloc</code>	allocation		
	<code>typeof</code>	Get type from identifier	<code>typeof(int)</code>	No
	<code>nameof</code>	Get name of identifier	<code>nameof(x)</code>	No
	<code>checked</code>	Integral overflow check on	<code>checked(x)</code>	No
	<code>unchecked</code>	Integral overflow check off	<code>unchecked(x)</code>	No
	<code>default</code>	Default value	<code>default(char)</code>	No
Unary	<code>await</code>	Await	<code>await myTask</code>	No
	<code>sizeof</code>	Get size of struct	<code>sizeof(int)</code>	No
	<code>+</code>	Positive value of	<code>+x</code>	Yes
	<code>-</code>	Negative value of	<code>-x</code>	Yes
	<code>!</code>	Not	<code>!x</code>	Yes
	<code>~</code>	Bitwise complement	<code>~x</code>	Yes
	<code>++</code>	Pre-increment	<code>++x</code>	Yes
	<code>--</code>	Pre-decrement	<code>--x</code>	Yes
	<code>()</code>	Cast	<code>(int)x</code>	No
	<code>*</code>	Value at address	<code>*x</code>	No
	<code>(unsafe)</code>			
	<code>&amp;</code>	Address of value	<code>&amp;x</code>	No
	<code>(unsafe)</code>			
Range	<code>..</code>	Start and end of a	<code>x..y</code>	No

range of indices				
Multiplicative	*	Multiply	$x * y$	Yes
	/	Divide	$x / y$	Yes
	%	Remainder	$x \% y$	Yes
Additive	+	Add	$x + y$	Yes
	-	Subtract	$x - y$	Yes
Shift	<<	Shift left	$x << 1$	Yes
	>>	Shift right	$x >> 1$	Yes
Relational	<	Less than	$x < y$	Yes
	>	Greater than	$x > y$	Yes
	<=	Less than or equal to	$x <= y$	Yes
	>=	Greater than or equal to	$x >= y$	Yes
	is	Type is or is subclass of	$x \text{ is } y$	No
	as	Type conversion	$x \text{ as } y$	No
Equality	==	Equals	$x == y$	Yes
	!=	Not equals	$x != y$	Yes
Logical And	&	And	$x \& y$	Yes
Logical Xor	^	Exclusive Or	$x ^ y$	Yes
Logical Or		Or	$x   y$	Yes
Conditional And	&&	Conditional And	$x \&& y$	Via &
Conditional		Conditional Or	$x    y$	Via

Or

Null coalescing	??	Null coalescing	x ?? y	No
Conditional	?:	Conditional	isTrue ? thenThisValue : elseThisValue	No
Assignment & Lambda	=	Assign	x = y	No
	*=	Multiply self by	x *= 2	Via *
	/=	Divide self by	x /= 2	Via /
	+=	Add to self	x += 2	Via +
	-=	Subtract from self	x -= 2	Via -
	<<=	Shift self left by	x <<= 2	Via <<
	>>=	Shift self right by	x >>= 2	Via >>
	&=	And self by	x &= 2	Via &
	^=	Exclusive-Or self by	x ^= 2	Via ^
	=	Or self by	x  = 2	Via
	??=	Null-coalescing assignment	x ??= 0	No
	=>	Lambda	x => x + 1	No

## Null Operators

C# provides three operators to make it easier to work with nulls: the *null-coalescing operator*, the *null-coalescing assignment operator*,

and the *null-conditional operator*.

## Null-Coalescing Operator

The ?? operator is the *null-coalescing operator*. It says, “If the operand to the left is non-null, give it to me; otherwise, give me another value.” For example:

```
string s1 = null;
string s2 = s1 ?? "nothing"; // s2 evaluates to "nothing"
```

If the lefthand expression is non-null, the righthand expression is never evaluated. The null-coalescing operator also works with nullable value types (see “[Nullable Value Types](#)” in Chapter 4).

## Null-Coalescing Assignment Operator (C# 8)

The ??= operator is the *null-coalescing assignment operator*. It says, “If the operand to the left is null, assign the right operand to the left operand.” For example:

```
string s1 = null;
s1 ??= "something";
Console.WriteLine (s1); // something

s1 ??= "everything";
Console.WriteLine (s1); // something
```

The operator is useful to replace the pattern

```
if (myVariable == null) myVariable = someDefault;
```

with:

```
myVariable ??= someDefault;
```

## Null-Conditional Operator

The `?.` operator is the *null-conditional* or “Elvis” operator (after the Elvis emoticon). It allows you to call methods and access members just like the standard dot operator except that if the operand on the left is null, the expression evaluates to null instead of throwing a `NullReferenceException`:

```
System.Text.StringBuilder sb = null;
string s = sb?.ToString(); // No error; s instead evaluates to null
```

The last line is equivalent to the following:

```
string s = (sb == null ? null : sb.ToString());
```

Upon encountering a null, the Elvis operator short-circuits the remainder of the expression. In the following example, `s` evaluates to null, even with a standard dot operator between `ToString()` and `ToUpper()`:

```
System.Text.StringBuilder sb = null;
string s = sb?.ToString().ToUpper(); // s evaluates to null without error
```

Repeated use of Elvis is necessary only if the operand immediately to its left might be null. The following expression is robust to both `x` being null and `x.y` being null:

```
x?.y?.z
```

It is equivalent to the following (except that `x.y` is evaluated only once):

```
x == null ? null  
    : (x.y == null ? null : x.y.z)
```

The final expression must be capable of accepting a null. The following is illegal:

```
System.Text.StringBuilder sb = null;  
int length = sb?.ToString().Length; // Illegal : int cannot be null
```

We can fix this with the use of nullable value types (see “[Nullable Value Types](#)” in Chapter 4). If you’re already familiar with nullable value types, here’s a preview:

```
int? length = sb?.ToString().Length; // OK: int? can be null
```

You can also use the null-conditional operator to call a void method:

```
someObject?.SomeVoidMethod();
```

If `someObject` is null, this becomes a “no-operation” rather than throwing a `NullReferenceException`.

You can use the null-conditional operator with the commonly used type members that we describe in [Chapter 3](#), including *methods*, *fields*, *properties*, and *indexers*. It also combines well with the *null-coalescing operator*:

```
System.Text.StringBuilder sb = null;  
string s = sb?.ToString() ?? "nothing"; // s evaluates to "nothing"
```

## Statements

Functions comprise statements that execute sequentially in the textual order in which they appear. A *statement block* is a series of statements appearing between braces (the {} tokens).

### Declaration Statements

A declaration statement declares a new variable, optionally initializing the variable with an expression. A declaration statement ends in a semicolon. You may declare multiple variables of the same type in a comma-separated list:

```
string someWord = "rosebud";  
int someNumber = 42;  
bool rich = true, famous = false;
```

A constant declaration is like a variable declaration except that it cannot be changed after it has been declared, and the initialization must occur with the declaration (see “[Constants](#)” in Chapter 3):

```
const double c = 2.99792458E08;  
c += 10;                                // Compile-time error
```

## LOCAL VARIABLES

The scope of a local variable or local constant extends throughout the current block. You cannot declare another local variable with the same name in the current block or in any nested blocks:

```
static void Main()  
{  
    int x;  
    {  
        int y;  
        int x;          // Error - x already defined  
    }  
    {  
        int y;          // OK - y not in scope  
    }  
    Console.WriteLine(y); // Error - y is out of scope  
}
```

## NOTE

A variable's scope extends in *both directions* throughout its code block. This means that if we moved the initial declaration of `x` in this example to the bottom of the method, we'd get the same error. This is in contrast to C++ and is somewhat peculiar, given that it's not legal to refer to a variable or constant before it's declared.

## Expression Statements

Expression statements are expressions that are also valid statements. An expression statement must either change state or call something that might change state. Changing state essentially means changing a variable. Following are the possible expression statements:

- Assignment expressions (including increment and decrement expressions)
- Method call expressions (both void and nonvoid)
- Object instantiation expressions

Here are some examples:

```
// Declare variables with declaration statements:  
string s;  
int x, y;  
System.Text.StringBuilder sb;  
  
// Expression statements  
x = 1 + 2;           // Assignment expression  
x++;                // Increment expression
```

```
y = Math.Max (x, 5);      // Assignment expression  
Console.WriteLine (y);    // Method call expression  
sb = new StringBuilder(); // Assignment expression  
new StringBuilder();     // Object instantiation expression
```

When you call a constructor or a method that returns a value, you're not obliged to use the result. However, unless the constructor or method changes state, the statement is completely useless:

```
new StringBuilder();      // Legal, but useless  
new string ('c', 3);    // Legal, but useless  
x.Equals (y);          // Legal, but useless
```

## Selection Statements

C# has the following mechanisms to conditionally control the flow of program execution:

- Selection statements (**if**, **switch**)
- Conditional operator (**? :**)
- Loop statements (**while**, **do-while**, **for**, **foreach**)

This section covers the simplest two constructs: the **if** statement and the **switch** statement.

### THE IF STATEMENT

An **if** statement executes a statement if a **bool** expression is true:

```
if (5 < 2 * 3)
```

```
Console.WriteLine ("true");      // true
```

The statement can be a code block:

```
if (5 < 2 * 3)
{
    Console.WriteLine ("true");
    Console.WriteLine ("Let's move on!");
}
```

## THE ELSE CLAUSE

An **if** statement can optionally feature an **else** clause:

```
if (2 + 2 == 5)
    Console.WriteLine ("Does not compute");
else
    Console.WriteLine ("False");      // False
```

Within an **else** clause, you can nest another **if** statement:

```
if (2 + 2 == 5)
    Console.WriteLine ("Does not compute");
else
    if (2 + 2 == 4)
        Console.WriteLine ("Computes"); // Computes
```

## CHANGING THE FLOW OF EXECUTION WITH BRACES

An **else** clause always applies to the immediately preceding **if** statement in the statement block:

```
if (true)
    if (false)
        Console.WriteLine();
    else
        Console.WriteLine ("executes");
```

This is semantically identical to the following:

```
if (true)
{
    if (false)
        Console.WriteLine();
    else
        Console.WriteLine ("executes");
}
```

We can change the execution flow by moving the braces:

```
if (true)
{
    if (false)
        Console.WriteLine();
}
else
    Console.WriteLine ("does not execute");
```

With braces, you explicitly state your intention. This can improve the readability of nested `if` statements—even when not required by the compiler. A notable exception is with the following pattern:

```
static void TellMeWhatICanDo (int age)
{
```

```
if (age >= 35)
    Console.WriteLine ("You can be president!");
else if (age >= 21)
    Console.WriteLine ("You can drink!");
else if (age >= 18)
    Console.WriteLine ("You can vote!");
else
    Console.WriteLine ("You can wait!");
}
```

Here, we've arranged the `if` and `else` statements to mimic the `elseif` construct of other languages (and C#'s `#elif` preprocessor directive). Visual Studio's auto-formatting recognizes this pattern and preserves the indentation. Semantically, though, each `if` statement following an `else` statement is functionally nested within the `else` clause.

## THE SWITCH STATEMENT

`switch` statements let you branch program execution based on a selection of possible values that a variable might have. `switch` statements can result in cleaner code than multiple `if` statements because `switch` statements require an expression to be evaluated only once:

```
static void ShowCard (int cardNumber)
{
    switch (cardNumber)
    {
        case 13:
            Console.WriteLine ("King");
            break;
        case 12:
            Console.WriteLine ("Queen");
            break;
```

```

case 11:
    Console.WriteLine ("Jack");
    break;
case -1:                      // Joker is -1
    goto case 12;              // In this game joker counts as
queen
default:                      // Executes for any other
cardNumber
    Console.WriteLine (cardNumber);
    break;
}
}

```

This example demonstrates the most common scenario, which is switching on *constants*. When you specify a constant, you're restricted to the built-in integral types, `bool`, `char`, `enum` types, and the `string` type.

At the end of each `case` clause, you must specify explicitly where execution is to go next, with some kind of jump statement (unless your code ends in an infinite loop). Here are the options:

- `break` (jumps to the end of the `switch` statement)
- `goto case X` (jumps to another `case` clause)
- `goto default` (jumps to the `default` clause)
- Any other jump statement—namely, `return`, `throw`, `continue`, or `goto label`

When more than one value should execute the same code, you can list the common `cases` sequentially:

```
switch (cardNumber)
{
    case 13:
    case 12:
    case 11:
        Console.WriteLine ("Face card");
        break;
    default:
        Console.WriteLine ("Plain card");
        break;
}
```

This feature of a `switch` statement can be pivotal in terms of producing cleaner code than multiple `if-else` statements.

## SWITCHING ON TYPES

### NOTE

Switching on a type is a special case of switching on a *pattern*. A number of other (moderately useful) patterns were introduced in C# 7 and C# 8; see “[Patterns](#)” in [Chapter 4](#) for a full discussion.

From C# 7, you can also switch on *types*:

```
static void Main()
{
    TellMeTheType (12);
    TellMeTheType ("hello");
    TellMeTheType (true);
}
```

```

static void TellMeTheType (object x)    // object allows any type.
{
    switch (x)
    {
        case int i:
            Console.WriteLine ("It's an int!");
            Console.WriteLine ($"The square of {i} is {i * i}");
            break;
        case string s:
            Console.WriteLine ("It's a string");
            Console.WriteLine ($"The length of {s} is {s.Length}");
            break;
        default:
            Console.WriteLine ("I don't know what x is");
            break;
    }
}

```

(The `object` type allows for a variable of any type; we discuss this fully in “[Inheritance](#)” and “[The object Type](#)” in [Chapter 3](#).)

Each `case` clause specifies a type upon which to match, and a variable upon which to assign the typed value if the match succeeds (the “pattern” variable). Unlike with constants, there’s no restriction on what types you can use.

You can predicate a `case` with the `when` keyword:

```

switch (x)
{
    case bool b when b == true:      // Fires only when b is true
        Console.WriteLine ("True!");
        break;
    case bool b:
        Console.WriteLine ("False!");
}

```

```
        break;  
    }
```

The order of the case clauses can matter when switching on type (unlike when switching on constants). This example would give a different result if we reversed the two cases (in fact, it would not even compile, because the compiler would determine that the second case is unreachable). An exception to this rule is the `default` clause, which is always executed last, regardless of where it appears.

If you want to switch on a type, but are uninterested in its value, you can use a *discard* (`_`):

```
case DateTime _:  
    Console.WriteLine ("It's a DateTime");
```

You can stack multiple case clauses. The `Console.WriteLine` in the following code will execute for any floating-point type greater than 1,000:

```
switch (x)  
{  
    case float f when f > 1000:  
    case double d when d > 1000:  
    case decimal m when m > 1000:  
        Console.WriteLine ("We can refer to x here but not f or d or m");  
        break;  
}
```

In this example, the compiler lets us consume the pattern variables `f`, `d`, and `m`, *only* in the `when` clauses. When we call

`Console.WriteLine`, its unknown which one of those three variables will be assigned, so the compiler puts all of them out of scope.

You can mix and match constants and patterns in the same `switch` statement. And you can also switch on the null value:

```
case null:  
    Console.WriteLine ("Nothing here");  
    break;
```

## SWITCH EXPRESSIONS (C# 8)

From C# 8, you can use `switch` in the context of an *expression*. Assuming that `cardNumber` is of type `int`, the following illustrates its use:

```
string cardName = cardNumber switch  
{  
    13 => "King",  
    12 => "Queen",  
    11 => "Jack",  
    _ => "Pip card" // equivalent to 'default'  
};
```

Notice that the `switch` keyword appears *after* the variable name, and that the case clauses are expressions (terminated by commas) rather than statements. `switch` expressions are more compact than their `switch` statement counterparts, and you can use them in LINQ queries (Chapter 8).

If you omit the default expression `(_)` and the switch fails to match, an exception is thrown.

You can also switch on multiple values (the *tuple* pattern):

```
int cardNumber = 12;
string suit = "spades";

string cardName = (cardNumber, suit) switch
{
    (13, "spades") => "King of spades",
    (13, "clubs") => "King of clubs",
    ...
};
```

Many more options are possible through the use of *patterns* (see “[Patterns](#)” in [Chapter 4](#)).

## Iteration Statements

C# enables a sequence of statements to execute repeatedly with the `while`, `do-while`, `for`, and `foreach` statements.

### WHILE AND DO-WHILE LOOPS

`while` loops repeatedly execute a body of code while a `bool` expression is true. The expression is tested *before* the body of the loop is executed:

```
int i = 0;
while (i < 3)
{
```

```
    Console.WriteLine (i);
    i++;
}
```

OUTPUT:

```
0
1
2
```

**do-while** loops differ in functionality from **while** loops only in that they test the expression *after* the statement block has executed (ensuring that the block is always executed at least once). Here's the preceding example rewritten with a **do-while** loop:

```
int i = 0;
do
{
    Console.WriteLine (i);
    i++;
}
while (i < 3);
```

## FOR LOOPS

**for** loops are like **while** loops with special clauses for *initialization* and *iteration* of a loop variable. A **for** loop contains three clauses as follows:

```
for (initialization-clause; condition-clause; iteration-clause)
    statement-or-statement-block
```

Here's what each clause does:

## Initialization clause

Executed before the loop begins; used to initialize one or more *iteration* variables

## Condition clause

The `bool` expression that, while true, will execute the body

## Iteration clause

Executed *after* each iteration of the statement block; typically used to update the iteration variable

For example, the following prints the numbers 0 through 2:

```
for (int i = 0; i < 3; i++)
    Console.WriteLine (i);
```

The following prints the first 10 Fibonacci numbers (in which each number is the sum of the previous two):

```
for (int i = 0, prevFib = 1, curFib = 1; i < 10; i++)
{
    Console.WriteLine (prevFib);
    int newFib = prevFib + curFib;
    prevFib = curFib; curFib = newFib;
}
```

Any of the three parts of the `for` statement can be omitted. You can implement an infinite loop such as the following (though `while(true)` can be used, instead):

```
for (;;) {
```

```
Console.WriteLine ("interrupt me");
```

## FOREACH LOOPS

The `foreach` statement iterates over each element in an enumerable object. Most of the types in C# and .NET Core that represent a set or list of elements are enumerable. For example, both an array and a string are enumerable. Here is an example of enumerating over the characters in a string, from the first character through to the last:

```
foreach (char c in "beer") // c is the iteration variable
```

```
    Console.WriteLine (c);
```

OUTPUT:

```
b  
e  
e  
r
```

We define enumerable objects in “Enumeration and Iterators” in [Chapter 4](#).

## Jump Statements

The C# jump statements are `break`, `continue`, `goto`, `return`, and `throw`.

## NOTE

Jump statements obey the reliability rules of `try` statements (see “[try Statements and Exceptions](#)” in Chapter 4). This means that:

- A jump out of a `try` block always executes the `try`’s `finally` block before reaching the target of the jump.
- A jump cannot be made from the inside to the outside of a `finally` block (except via `throw`).

## THE BREAK STATEMENT

The `break` statement ends the execution of the body of an iteration or `switch` statement:

```
int x = 0;
while (true)
{
    if (x++ > 5)
        break;      // break from the loop
}
// execution continues here after break
...
```

## THE CONTINUE STATEMENT

The `continue` statement forgoes the remaining statements in a loop and makes an early start on the next iteration. The following loop skips even numbers:

```
for (int i = 0; i < 10; i++)
{
```

```
if ((i % 2) == 0)      // If i is even,  
    continue;           // continue with next iteration  
  
    Console.Write (i + " ");  
}  
  
OUTPUT: 1 3 5 7 9
```

## THE GOTO STATEMENT

The **goto** statement transfers execution to another label within a statement block. The form is as follows:

```
goto statement-label;
```

Or, when used within a **switch** statement:

```
goto case case-constant;    // (Only works with constants, not  
patterns)
```

A label is a placeholder in a code block that precedes a statement, denoted with a colon suffix. The following iterates the numbers 1 through 5, mimicking a **for** loop:

```
int i = 1;  
startLoop:  
if (i <= 5)  
{  
    Console.Write (i + " ");  
    i++;  
    goto startLoop;  
}
```

```
OUTPUT: 1 2 3 4 5
```

The `goto case case-constant` transfers execution to another case in a `switch` block (see “[The switch statement](#)”).

## THE RETURN STATEMENT

The `return` statement exits the method and must return an expression of the method’s return type if the method is nonvoid:

```
static decimal AsPercentage (decimal d)
{
    decimal p = d * 100m;
    return p;           // Return to the calling method with value
}
```

A `return` statement can appear anywhere in a method (except in a `finally` block), and can be used more than once.

## THE THROW STATEMENT

The `throw` statement throws an exception to indicate an error has occurred (see “[try Statements and Exceptions](#)” in Chapter 4):

```
if (w == null)
    throw new ArgumentNullException (...);
```

## Miscellaneous Statements

The `using` statement provides an elegant syntax for calling `Dispose` on objects that implement `IDisposable`, within a `finally` block (see

“try Statements and Exceptions” in Chapter 4 and “IDisposable, Dispose, and Close” in Chapter 12).

### NOTE

C# overloads the `using` keyword to have independent meanings in different contexts. Specifically, the `using directive` is different from the `using statement`.

The `lock` statement is a shortcut for calling the `Enter` and `Exit` methods of the `Monitor` class (see Chapters 14 and 23).

## Namespaces

A namespace is a domain for type names. Types are typically organized into hierarchical namespaces, making them easier to find and avoiding conflicts. For example, the RSA type that handles public-key encryption is defined within the following namespace:

```
System.Security.Cryptography
```

A namespace forms an integral part of a type’s name. The following code calls RSA’s `Create` method:

```
System.Security.Cryptography.RSA rsa =
    System.Security.Cryptography.RSA.Create();
```

## NOTE

Namespaces are independent of assemblies, which are units of deployment such as an `.exe` or `.dll` (described in [Chapter 18](#)).

Namespaces also have no impact on member visibility—`public`, `internal`, `private`, and so on.

The `namespace` keyword defines a namespace for types within that block; for example:

```
namespace Outer.Middle.Inner
{
    class Class1 {}
    class Class2 {}
}
```

The dots in the namespace indicate a hierarchy of nested namespaces. The code that follows is semantically identical to the preceding example:

```
namespace Outer
{
    namespace Middle
    {
        namespace Inner
        {
            class Class1 {}
            class Class2 {}
        }
    }
}
```

You can refer to a type with its *fully qualified name*, which includes all namespaces from the outermost to the innermost. For example, we could refer to `Class1` in the preceding example as `Outer.Middle.Inner.Class1`.

Types not defined in any namespace are said to reside in the *global namespace*. The global namespace also includes top-level namespaces, such as `Outer` in our example.

## The using Directive

The `using` directive *imports* a namespace, allowing you to refer to types without their fully qualified names. The following imports the previous example's `Outer.Middle.Inner` namespace:

```
using Outer.Middle.Inner;

class Test
{
    static void Main()
    {
        Class1 c;      // Don't need fully qualified name
    }
}
```

## NOTE

It's legal (and often desirable) to define the same type name in different namespaces. However, you'd typically do so only if it was unlikely for a consumer to want to import both namespaces at once. A good example is the `TextBox` class, which is defined both in `System.Windows.Controls` (WPF) and `System.Windows.Forms.Controls` (Windows Forms).

## using static

The `using static` directive imports a *type* rather than a namespace. All static members of the imported type can then be used without qualification. In the following example, we call the `Console` class's static `WriteLine` method without needing to refer to the type:

```
using static System.Console;

class Test
{
    static void Main() { WriteLine ("Hello"); }
}
```

The `using static` directive imports all accessible static members of the type, including fields, properties, and nested types ([Chapter 3](#)). You can also apply this directive to enum types, in which case their members are imported. So, if we import the following enum type:

```
using static System.Windows.Visibility;
```

we can specify `Hidden` instead of `Visibility.Hidden`:

```
var textBox = new TextBox { Visibility = Hidden }; // XAML-style
```

Should an ambiguity arise between multiple static imports, the C# compiler is not smart enough to infer the correct type from the context and will generate an error.

## Rules Within a Namespace

### NAME SCOPING

You can use names declared in outer namespaces unqualified within inner namespaces. In this example, `Class1` does not need qualification within `Inner`:

```
namespace Outer
{
    class Class1 {}

    namespace Inner
    {
        class Class2 : Class1 {}
    }
}
```

If you want to refer to a type in a different branch of your namespace hierarchy, you can use a partially qualified name. In the following example, we base `SalesReport` on `Common.ReportBase`:

```
namespace MyTradingCompany
{
    namespace Common
    {
```

```
    class ReportBase {}

}

namespace ManagementReporting
{
    class SalesReport : Common.ReportBase {}
}
```

## NAME HIDING

If the same type name appears in both an inner and an outer namespace, the inner name wins. To refer to the type in the outer namespace, you must qualify its name:

```
namespace Outer
{
    class Foo { }

    namespace Inner
    {
        class Foo { }

        class Test
        {
            Foo f1;           // = Outer.Inner.Foo
            Outer.Foo f2;    // = Outer.Foo
        }
    }
}
```

## NOTE

All type names are converted to fully qualified names at compile time.  
Intermediate Language (IL) code contains no unqualified or partially qualified names.

## REPEATED NAMESPACES

You can repeat a namespace declaration, as long as the type names within the namespaces don't conflict:

```
namespace Outer.Middle.Inner
{
    class Class1 {}
}

namespace Outer.Middle.Inner
{
    class Class2 {}
}
```

We can even break the example into two source files such that we could compile each class into a different assembly.

Source file 1:

```
namespace Outer.Middle.Inner
{
    class Class1 {}
}
```

Source file 2:

```
namespace Outer.Middle.Inner
{
    class Class2 {}
}
```

## NESTED USING DIRECTIVES

You can nest a `using` directive within a namespace. This allows you to scope the `using` directive within a namespace declaration. In the following example, `Class1` is visible in one scope, but not in another:

```
namespace N1
{
    class Class1 {}
}

namespace N2
{
    using N1;

    class Class2 : Class1 {}
}

namespace N2
{
    class Class3 : Class1 {}    // Compile-time error
}
```

## Aliasing Types and Namespaces

Importing a namespace can result in type-name collision. Rather than importing the entire namespace, you can import just the specific types that you need, giving each type an alias:

```
using PropertyInfo2 = System.Reflection.PropertyInfo;
class Program { PropertyInfo2 p; }
```

An entire namespace can be aliased, as follows:

```
using R = System.Reflection;
class Program { R.PropertyInfo p; }
```

## Advanced Namespace Features

### EXTERN

Extern aliases allow your program to reference two types with the same fully qualified name (i.e., the namespace and type name are identical). This is an unusual scenario and can occur only when the two types come from different assemblies. Consider the following example.

Library 1, compiled to *Widgets1.dll*:

```
namespace Widgets
{
    public class Widget {}
}
```

Library 2, compiled to *Widgets2.dll*:

```
namespace Widgets
{
    public class Widget {}
}
```

Application, which references *Widgets1.dll* and *Widgets2.dll*:

```
using Widgets;

class Test
{
    static void Main()
    {
        Widget w = new Widget();
    }
}
```

The application cannot compile, because `Widget` is ambiguous.

Extern aliases can resolve the ambiguity. The first step is to modify the application's `.csproj` file, assigning a unique alias to each reference:

```
<ItemGroup>
    <Reference Include="Widgets1">
        <Aliases>W1</Aliases>
    </Reference>
    <Reference Include="Widgets2">
        <Aliases>W2</Aliases>
    </Reference>
</ItemGroup>
```

The second step is to use the `extern alias` directive:

```
extern alias W1;
extern alias W2;

class Test
{
```

```
static void Main()
{
    W1.Widgets.Widget w1 = new W1.Widgets.Widget();
    W2.Widgets.Widget w2 = new W2.Widgets.Widget();
}
```

## NAMESPACE ALIAS QUALIFIERS

As we mentioned earlier, names in inner namespaces hide names in outer namespaces. However, sometimes even the use of a fully qualified type name does not resolve the conflict. Consider the following example:

```
namespace N
{
    class A
    {
        static void Main() => new A.B();      // Instantiate class B
        public class B {}                      // Nested type
    }
}

namespace A
{
    class B {}
}
```

The `Main` method could be instantiating either the nested class `B`, or the class `B` within the namespace `A`. The compiler always gives higher precedence to identifiers in the current namespace—in this case, the nested `B` class.

To resolve such conflicts, a namespace name can be qualified, relative to one of the following:

- The global namespace—the root of all namespaces (identified with the contextual keyword **global**)
- The set of extern aliases

The `::` token performs namespace alias qualification. In this example, we qualify using the global namespace (this is most commonly seen in autogenerated code to avoid name conflicts):

```
namespace N
{
    class A
    {
        static void Main()
        {
            System.Console.WriteLine (new A.B());
            System.Console.WriteLine (new global::A.B());
        }

        public class B {}
    }
}

namespace A
{
    class B {}
}
```

Here is an example of qualifying with an alias (adapted from the example in “Extern”):

```
extern alias W1;
extern alias W2;

class Test
{
    static void Main()
    {
        W1::Widgets.Widget w1 = new W1::Widgets.Widget();
        W2::Widgets.Widget w2 = new W2::Widgets.Widget();
    }
}
```

- 
- 1 A minor caveat is that very large `long` values lose some precision when converted to `double`.
  - 2 Technically, `decimal` is a floating-point type, too, although it's not referred to as such in the C# language specification.
  - 3 It's possible to *overload* these operators ([Chapter 4](#)) such that they return a non-`bool` type, but this is almost never done in practice.
  - 4 An exception to this rule is when calling Component Object Model (COM) methods. We discuss this in [Chapter 24](#).

# 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	<code>static</code>
Access modifiers	<code>public internal private protected</code>
Inheritance modifier	<code>new</code>
Unsafe code modifier	<code>unsafe</code>
Read-only modifier	<code>readonly</code>
Threading modifier	<code>volatile</code>

## 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 avoids 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:

```
static void Main()
{
    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);
```

## OVERLOADING METHODS

A type can overload methods (have 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
```

## PASS-BY-VALUE VERSUS PASS-BY-REFERENCE

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
```

## 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](#)).

The `static` modifier is invalid for local methods. They are implicitly static if the enclosing method is static.

## STATIC LOCAL METHODS (C# 8)

Adding the `static` modifier to a local method prevents it from seeing the local variables and parameters of the enclosing method. This helps to reduce coupling as well as enabling the local method to declare variables as it pleases, without risk of colliding with those in the containing method.

## 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:

```
public class Panda
{
    string name;                      // Define field
    public Panda (string n)           // Define constructor
```

```
{  
    name = n;                      // Initialization code (set up field)  
}  
}  
...  
  
Panda p = new Panda ("Petey"); // Call constructor
```

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 avoid 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 were introduced in C# 3.0.

---

## 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);
```

An advantage of this approach is that we could make `Bunny`'s fields (or *properties*, which we explain shortly) read-only if we choose. 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.

The disadvantage in this approach 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 can be 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.)

Hence, you should exercise caution with optional parameters in public functions if you want to offer binary compatibility between assembly versions.

## 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	<code>static</code>
Access modifiers	<code>public internal private protected</code>
Inheritance modifiers	<code>new virtual abstract override sealed</code>
Unmanaged code modifiers	<code>unsafe extern</code>

## 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.

## 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) {...}
```

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.

With properties in Windows Runtime libraries, the compiler assumes the `put_XXX` naming convention rather than `set_XXX`.

## 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 (C# 8)

You can support indices and ranges (see “[Indices and Ranges \(C# 8\)](#)” in [Chapter 2](#)) 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`.

### NOTE

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

## 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:

```
class Program
{
    static void Main() { 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`.

## 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**:

```
static void Main() { 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 `InvalidOperationException` 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!" Whereas with the `as` operator, 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 (somewhat less useful) kinds of patterns, introduced in C# 7 and C# 8. 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](#)”).

### NOTE

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.

## 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 seal the class itself, implicitly sealing all the virtual functions, by applying the `sealed` modifier to the class itself. Sealing a class is more common than sealing a function member.

Although you can seal 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:

1. Fields are initialized

2. Arguments to base-class constructor calls are evaluated

2. From base class to subclass:

1. Constructor bodies execute

The following code demonstrates:

```
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).<sup>1</sup> 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 `InvalidOperationException` 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:

```
using System;

public class Point { public int X, Y; }

class Test
{
    static void Main()
    {
        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
    }
}
```

`System.Type` also has methods that act as a gateway to the runtime's reflection model, described in [Chapter 19](#).

## 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:

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

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

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 a 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 (); // 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

From C# 7.2, you can apply the `readonly` modifier to a struct to enforce that all fields are `readonly`; this aids in declaring intent as well as allowing 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, C# 8 assists with a new feature whereby you can apply the `readonly` modifier 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 24](#) (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:

```
struct Point { public int X, Y; }
...
```

```
void SomeMethod()
{
    Point p; // p will reside on the stack
}
```

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.

From C# 7.2, you can add the `ref` modifier to a struct's declaration to ensure 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:

```
ref struct Point { public int X, Y; }
class MyClass { Point P; } // Error: will not compile!
...
var points = new Point [100]; // 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 six *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.InternalsVisibleTo` assembly attribute, specifying the name of the friend assembly as follows:

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

If the friend assembly has a strong name (see [Chapter 18](#)), 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().GetPublicKey()
        .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 \(C# 8\)](#)”.)
- 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` was 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:

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 (C# 8)

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`).

## WRITING A CLASS VERSUS AN INTERFACE

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

As a guideline:

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`.

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

## 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 BorderSide { None=0, Left=1, Right=2, Top=4, Bottom=8 }
```

or:

```
enum BorderSide { 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:

```
BorderSide leftRight = BorderSide.Left | BorderSide.Right;  
  
if ((leftRight & BorderSide.Left) != 0)  
    Console.WriteLine ("Includes Left");      // Includes Left  
  
string formatted = leftRight.ToString();    // "Left, Right"  
  
BorderSide s = BorderSide.Left;  
s |= BorderSide.Right;  
Console.WriteLine (s == leftRight);      // True  
  
s ^= BorderSide.Right;                  // Toggles BorderSide.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:

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

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

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

# 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?
```

unless 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 needed a stack of integers, but we didn'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: open generic types 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 19).

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 framework 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:

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

class Test
{
    static void Main()
    {
        Console.WriteLine (++Bob<int>.Count);      // 1
        Console.WriteLine (++Bob<int>.Count);      // 2
        Console.WriteLine (++Bob<string>.Count);   // 1
        Console.WriteLine (++Bob<object>.Count);   // 1
    }
}
```

```
    }  
}
```

## Type Parameters and Conversions

C#'s cast operator can perform several kinds of conversion, including the following:

- 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 Core, 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();
```

## NOTE

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

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` into an interface where `T` can appear only in *output* positions.

## NOTE

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

We can take advantage of 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, if the `Stack<T>` class implements the following interface:

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

we can 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 as part of .NET Core:

```
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 `.dlls`—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`, 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;
}
```

---

<sup>1</sup> The reference type can also be `System.ValueType` or `System.Enum` (Chapter 6).

# Chapter 7. Collections

---

.NET Core provides a standard set of types for storing and managing collections of objects. These include resizable lists, linked lists, sorted and unsorted dictionaries, as well as arrays. Of these, only arrays form part of the C# language; the remaining collections are just classes you instantiate like any other.

We can divide the types in the Framework for collections into the following categories:

- Interfaces that define standard collection protocols
- Ready-to-use collection classes (lists, dictionaries, etc.)
- Base classes for writing application-specific collections

This chapter covers each of these categories, with an additional section on the types used in determining element equality and order.

The collection namespaces are as follows:

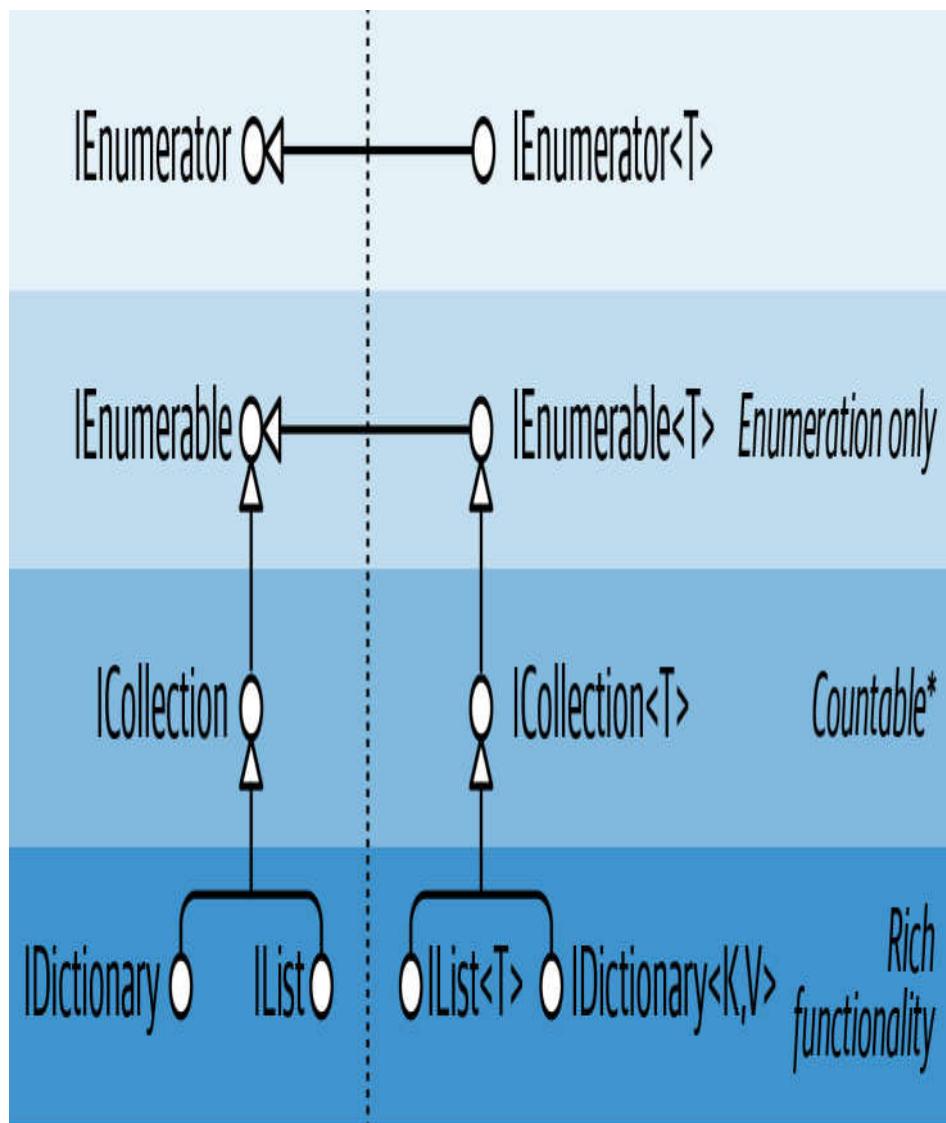
Namespace	Contains
<code>System.Collections</code>	Nongeneric collection classes and interfaces
<code>System.Collections.Specialized</code>	Strongly typed nongeneric collection classes

<code>System.Collections.Generic</code>	Generic collection classes and interfaces
---	---

| `System.Collections.ObjectModel` | Proxies and bases for custom collections |
| `System.Collections.Concurrent` | Thread-safe collections (see Chapter 23) |

## Enumeration

In computing, there are many different kinds of collections, ranging from simple data structures such as arrays or linked lists, to more complex ones such as red/black trees and hashtables. Although the internal implementation and external characteristics of these data structures vary widely, the ability to traverse the contents of the collection is an almost universal need. The Framework supports this need via a pair of interfaces (`IEnumerable`, `IEnumerator`, and their generic counterparts) that allow different data structures to expose a common traversal API. These are part of a larger set of collection interfaces illustrated in [Figure 7-1](#).



\**ICollection<T> has added functionality*

*Figure 7-1. Collection interfaces*

## IEnumerable and IEnumerator

The `IEnumerator` interface defines the basic low-level protocol by which elements in a collection are traversed—or enumerated—in a forward-only manner. Its declaration is as follows:

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

`MoveNext` advances the current element or “cursor” to the next position, returning `false` if there are no more elements in the collection. `Current` returns the element at the current position (usually cast from `object` to a more specific type). `MoveNext` must be called before retrieving the first element—this is to allow for an empty collection. The `Reset` method, if implemented, moves back to the start, allowing the collection to be enumerated again. `Reset` exists mainly for Component Object Model (COM) interoperability; calling it directly is generally avoided because it’s not universally supported (and is unnecessary in that it’s usually just as easy to instantiate a new enumerator.)

Collections do not usually *implement* enumerators; instead, they *provide* enumerators, via the interface `IEnumerable`:

```
public interface IEnumerable
{
    IEnumerator GetEnumerator();
}
```

By defining a single method returning an enumerator, **IEnumerable** provides flexibility in that the iteration logic can be farmed off to another class. Moreover, it means that several consumers can enumerate the collection at once without interfering with one another. You can think of **IEnumerable** as “**IEnumeratorProvider**,” and it is the most basic interface that collection classes implement.

The following example illustrates low-level use of **IEnumerable** and **IEnumerator**:

```
string s = "Hello";

// Because string implements IEnumerable, we can call GetEnumerator():
IEnumerator rator = s.GetEnumerator();

while (rator.MoveNext())
{
    char c = (char) rator.Current;
    Console.Write (c + ".");
}

// Output: H.e.l.l.o.
```

However, it's rare to call methods on enumerators directly in this manner because C# provides a syntactic shortcut: the **foreach** statement. Here's the same example rewritten using **foreach**:

```
string s = "Hello";      // The string class implements IEnumerable

foreach (char c in s)
    Console.Write (c + ".");
```

## IEnumerable<T> and IEnumerator<T>

IEnumerator and IEnumerable are nearly always implemented in conjunction with their extended generic versions:

```
public interface IEnumerator<T> : IEnumerator, IDisposable
{
    T Current { get; }
}

public interface IEnumerable<T> : IEnumerable
{
    IEnumerator<T> GetEnumerator();
}
```

By defining a typed version of `Current` and `GetEnumerator`, these interfaces strengthen static type safety, avoid the overhead of boxing with value-type elements, and are more convenient to the consumer. Arrays automatically implement `IEnumerable<T>` (where `T` is the member type of the array).

Thanks to the improved static type safety, calling the following method with an array of characters will generate a compile-time error:

```
void Test (IEnumerable<int> numbers) { ... }
```

It's a standard practice for collection classes to publicly expose `IEnumerable<T>` while “hiding” the nongeneric `IEnumerable` through explicit interface implementation. This is so that if you directly call `GetEnumerator()`, you get back the type-safe generic

`IEnumerator<T>`. There are times, though, when this rule is broken for reasons of backward compatibility (generics did not exist prior to C# 2.0). A good example is arrays—these must return the nongeneric (the nice way of putting it is *classic*) `IEnumerator` to avoid breaking earlier code. To get a generic `IEnumerator<T>`, you must cast to expose the explicit interface:

```
int[] data = { 1, 2, 3 };
var rator = ((IEnumerable <int>)data).GetEnumerator();
```

Fortunately, you rarely need to write this sort of code, thanks to the `foreach` statement.

## IENUMERABLE<T> AND IDISPOSABLE

`IEnumerator<T>` inherits from `IDisposable`. This allows enumerators to hold references to resources such as database connections—and ensure that those resources are released when enumeration is complete (or abandoned partway through). The `foreach` statement recognizes this detail and translates the following:

```
foreach (var element in somethingEnumerable) { ... }
```

into the logical equivalent of this:

```
using (var rator = somethingEnumerable.GetEnumerator())
{
    while (rator.MoveNext())
    {
        var element = rator.Current;
    }
}
```

```
    ...  
}
```

The `using` block ensures disposal—more on `IDisposable` in Chapter 12.

## Implementing the Enumeration Interfaces

You might want to implement `IEnumerable` or `IEnumerable<T>` for one or more of the following reasons:

- To support the `foreach` statement
- To interoperate with anything expecting a standard collection
- To meet the requirements of a more sophisticated collection interface
- To support collection initializers

## WHEN TO USE THE NONGENERIC INTERFACES

Given the extra type safety of the generic collection interfaces such as `IEnumerable<T>`, the question arises: do you ever need to use the nongeneric `IEnumerable` (or `ICollection` or `IList`)?

In the case of `IEnumerable`, you must implement this interface in conjunction with `IEnumerable<T>`—because the latter derives from the former. However, it's very rare that you actually implement these interfaces from scratch: in nearly all cases, you can take the higher-level approach of using iterator methods, `Collection<T>`, and LINQ.

So, what about as a consumer? In nearly all cases, you can manage entirely with the generic interfaces. The nongeneric interfaces are still occasionally useful, though, in their ability to provide type unification for collections across all element types. The following method, for instance, counts elements in any collection *recursively*:

```
public static int Count (IEnumerable e)
{
    int count = 0;
    foreach (object element in e)
    {
        var subCollection = element as IEnumerable;
        if (subCollection != null)
            count += Count (subCollection);
        else
            count++;
    }
    return count;
}
```

Because C# offers covariance with generic interfaces, it might seem valid to have this method instead accept `IEnumerable<object>`. This, however, would fail with value-type elements and with legacy collections that don't implement `IEnumerable<T>`—an example is `ControlCollection` in Windows Forms.

(On a slight tangent, you might have noticed a potential bug in our example: *cyclic* references will cause infinite recursion and crash the method. We could fix this most easily with the use of a `HashSet` [see “[HashSet<T>](#) and [SortedSet<T>](#)”].)

To implement `IEnumerable/IEnumerable<T>`, you must provide an enumerator. You can do this in one of three ways:

- If the class is “wrapping” another collection, by returning the wrapped collection’s enumerator
- Via an iterator using `yield return`
- By instantiating your own `IEnumerator/IEnumerator<T>` implementation

#### NOTE

You can also subclass an existing collection: `Collection<T>` is designed just for this purpose (see “[Customizable Collections and Proxies](#)”). Yet another approach is to use the LINQ query operators, which we cover in [Chapter 8](#).

Returning another collection’s enumerator is just a matter of calling `GetEnumerator` on the inner collection. However, this is viable only in the simplest scenarios in which the items in the inner collection are exactly what are required. A more flexible approach is to write an iterator, using C#’s `yield return` statement. An *iterator* is a C# language feature that assists in writing collections, in the same way the `foreach` statement assists in consuming collections. An iterator

automatically handles the implementation of `IEnumerable` and `IEnumerator`—or their generic versions. Here's a simple example:

```
public class MyCollection : IEnumerable
{
    int[] data = { 1, 2, 3 };

    public IEnumerator GetEnumerator()
    {
        foreach (int i in data)
            yield return i;
    }
}
```

Notice the *black magic*: `GetEnumerator` doesn't appear to return an enumerator at all! Upon parsing the `yield return` statement, the compiler writes a hidden nested enumerator class behind the scenes, and then refactors `GetEnumerator` to instantiate and return that class. Iterators are powerful and simple (and are used extensively in the implementation of LINQ-to-Object's standard query operators).

Keeping with this approach, we can also implement the generic interface `IEnumerable<T>`:

```
public class MyGenCollection : IEnumerable<int>
{
    int[] data = { 1, 2, 3 };

    public IEnumerator<int> GetEnumerator()
    {
        foreach (int i in data)
            yield return i;
    }
}
```

```
// Explicit implementation keeps it hidden:  
IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();  
}
```

Because `IEnumerable<T>` inherits from `IEnumerable`, we must implement both the generic and the nongeneric versions of `GetEnumerator`. In accordance with standard practice, we've implemented the nongeneric version explicitly. It can simply call the generic `GetEnumerator` because `IEnumerable<T>` inherits from `IEnumerable`.

The class we've just written would be suitable as a basis from which to write a more sophisticated collection. However, if we need nothing above a simple `IEnumerable<T>` implementation, the `yield return` statement allows for an easier variation. Rather than writing a class, you can move the iteration logic into a method returning a generic `IEnumerable<T>` and let the compiler take care of the rest. Here's an example:

```
public static IEnumerable <int> GetSomeIntegers()  
{  
    yield return 1;  
    yield return 2;  
    yield return 3;  
}
```

Here's our method in use:

```
foreach (int i in Test.GetSomeIntegers())  
    Console.WriteLine (i);
```

The final approach in writing `GetEnumerator` is to write a class that implements `IEnumerable` directly. This is exactly what the compiler does behind the scenes, in resolving iterators. (Fortunately, it's rare that you'll need to go this far yourself.) The following example defines a collection that's hardcoded to contain the integers 1, 2, and 3:

```
public class MyIntList : IEnumerable
{
    int[] data = { 1, 2, 3 };

    public IEnumerator GetEnumerator() => new Enumerator (this);

    class Enumerator : IEnumerator           // Define an inner class
    {                                       // for the enumerator.
        MyIntList collection;
        int currentIndex = -1;

        public Enumerator (MyIntList items) => this.collection = items;

        public object Current
        {
            get
            {
                if (currentIndex == -1)
                    throw new InvalidOperationException ("Enumeration not
started!");
                if (currentIndex == collection.data.Length)
                    throw new InvalidOperationException ("Past end of list!");
                return collection.data [currentIndex];
            }
        }

        public bool MoveNext()
        {
```

```
        if (currentIndex >= collection.data.Length - 1) return false;
        return ++currentIndex < collection.data.Length;
    }

    public void Reset() => currentIndex = -1;
}
}
```

## NOTE

Implementing `Reset` is optional—you can instead throw a `NotSupportedException`.

Note that the first call to `MoveNext` should move to the first (and not the second) item in the list.

To get on par with an iterator in functionality, we must also implement `IEnumerator<T>`. Here's an example with bounds checking omitted for brevity:

```
class MyIntList : IEnumerable<int>
{
    int[] data = { 1, 2, 3 };

    // The generic enumerator is compatible with both IEnumerable and
    // IEnumerable<T>. We implement the nongeneric GetEnumerator method
    // explicitly to avoid a naming conflict.

    public IEnumerator<int> GetEnumerator() => new Enumerator(this);
    IEnumerable.GetEnumerator() => new Enumerator(this);

    class Enumerator : IEnumerator<int>
    {
```

```

int currentIndex = -1;
MyIntList collection;

public Enumerator (MyIntList items) => this.items = items;

public int Current => collection.data [currentIndex];
object IEnumerator.Current => Current;

public bool MoveNext() => ++currentIndex < collection.data.Length;

public void Reset() => currentIndex = -1;

// Given we don't need a Dispose method, it's good practice to
// implement it explicitly, so it's hidden from the public
interface.
void IDisposable.Dispose() {}
}
}

```

The example with generics is faster because `IEnumerator<int>.Current` doesn't require casting from `int` to `object` and so avoids the overhead of boxing.

## The **ICollection** and **IList** Interfaces

Although the enumeration interfaces provide a protocol for forward-only iteration over a collection, they don't provide a mechanism to determine the size of the collection, access a member by index, search, or modify the collection. For such functionality, the .NET Framework defines the **ICollection**, **IList**, and **IDictionary** interfaces. Each comes in both generic and nongeneric versions; however, the nongeneric versions exist mostly for legacy support.

[Figure 7-1](#) showed the inheritance hierarchy for these interfaces. The easiest way to summarize them is as follows:

**IEnumerable<T> (and IEnumerable)**

Provides minimum functionality (enumeration only)

**ICollection<T> (and ICollection)**

Provides medium functionality (e.g., the Count property)

**IList<T>/IDictionary<K,V>** and their nongeneric versions

Provide maximum functionality (including “random” access by index/key)

**NOTE**

It’s rare that you’ll need to *implement* any of these interfaces. In nearly all cases when you need to write a collection class, you can instead subclass [Collection<T>](#) (see “Customizable Collections and Proxies”). LINQ provides yet another option that covers many scenarios.

The generic and nongeneric versions differ in ways over and above what you might expect, particularly in the case of **ICollection**. The reasons for this are mostly historical: because generics came later, the generic interfaces were developed with the benefit of hindsight, leading to a different (and better) choice of members. For this reason, **ICollection<T>** does not extend **ICollection**, **IList<T>** does not extend **IList**, and **IDictionary<TKey, TValue>** does not extend **IDictionary**. Of course, a collection class itself is free to implement both versions of an interface if beneficial (which it often is).

## NOTE

Another, subtler reason for `IList<T>` not extending `IList` is that casting to `IList<T>` would then return an interface with both `Add(T)` and `Add(object)` members. This would effectively defeat static type safety because you could call `Add` with an object of any type.

This section covers `ICollection<T>`, `IList<T>`, and their nongeneric versions; “Dictionaries” covers the dictionary interfaces.

## NOTE

There is no *consistent* rationale in the way the words *collection* and *list* are applied throughout the .NET Framework. For instance, because `IList<T>` is a more functional version of `ICollection<T>`, you might expect the class `List<T>` to be correspondingly more functional than the class `Collection<T>`. This is not the case. It’s best to consider the terms *collection* and *list* as broadly synonymous, except when a specific type is involved.

## **ICollection<T> and ICollection**

`ICollection<T>` is the standard interface for countable collections of objects. It provides the ability to determine the size of a collection (`Count`), determine whether an item exists in the collection (`Contains`), copy the collection into an array (`ToArray`), and determine whether the collection is read-only (`IsReadOnly`). For writable collections, you can also `Add`, `Remove`, and `Clear` items from the collection. And because it extends `IEnumerable<T>`, it can also be traversed via the `foreach` statement:

```
public interface ICollection<T> : IEnumerable<T>, IEnumerable
{
    int Count { get; }

    bool Contains (T item);
    void CopyTo (T[] array, int arrayIndex);
    bool IsReadOnly { get; }

    void Add(T item);
    bool Remove (T item);
    void Clear();
}
```

The nongeneric `ICollection` is similar in providing a countable collection, but it doesn't provide functionality for altering the list or checking for element membership:

```
public interface ICollection : IEnumerable
{
    int Count { get; }
    bool IsSynchronized { get; }
    object SyncRoot { get; }
    void CopyTo (Array array, int index);
}
```

The nongeneric interface also defines properties to assist with synchronization ([Chapter 14](#))—these were dumped in the generic version because thread safety is no longer considered intrinsic to the collection.

Both interfaces are fairly straightforward to implement. If implementing a read-only `ICollection<T>`, the `Add`, `Remove`, and `Clear` methods should throw a `NotSupportedException`.

These interfaces are usually implemented in conjunction with either the **IList** or the **IDictionary** interface.

## **IList<T>** and **IList**

**IList<T>** is the standard interface for collections indexable by position. In addition to the functionality inherited from **ICollection<T>** and **IEnumerable<T>**, it provides the ability to read or write an element by position (via an indexer) and insert/remove by position:

```
public interface IList<T> : ICollection<T>, IEnumerable<T>, IEnumerable
{
    T this [int index] { get; set; }
    int IndexOf (T item);
    void Insert (int index, T item);
    void RemoveAt (int index);
}
```

The **IndexOf** methods perform a linear search on the list, returning **-1** if the specified item is not found.

The nongeneric version of **IList** has more members because it inherits less from **ICollection**:

```
public interface IList : ICollection, IEnumerable
{
    object this [int index] { get; set }
    bool IsFixedSize { get; }
    bool IsReadOnly { get; }
    int Add (object value);
    void Clear();
```

```
    bool Contains (object value);
    int  IndexOf  (object value);
    void Insert   (int index, object value);
    void Remove   (object value);
    void RemoveAt (int index);
}
```

The `Add` method on the nongeneric `IList` interface returns an integer —this is the index of the newly added item. In contrast, the `Add` method on `ICollection<T>` has a `void` return type.

The general-purpose `List<T>` class is the quintessential implementation of both `IList<T>` and `IList`. C# arrays also implement both the generic and nongeneric `ILists` (although the methods that add or remove elements are hidden via explicit interface implementation and throw a `NotSupportedException` if called).

## NOTE

An `ArgumentException` is thrown if you try to access a multidimensional array via `IList`'s indexer. This is a trap when writing methods such as the following:

```
public object FirstOrDefault (IList list)
{
    if (list == null || list.Count == 0) return null;
    return list[0];
}
```

This might appear bulletproof, but it will throw an exception if called with a multidimensional array. You can test for a multidimensional array at runtime with this expression (more on this in [Chapter 19](#)):

```
list.GetType().IsArray && list.GetType().GetArrayRank()>1
```

## IReadOnlyCollection<T> and IReadOnlyList<T>

.NET Core also defines collection and list interfaces that expose just the members required for read-only operations:

```
public interface IReadOnlyCollection<out T> : IEnumerable<T>,
IEnumerable
{
    int Count { get; }
}

public interface IReadOnlyList<out T> : IReadOnlyCollection<T>,
IEnumerable<T>, IEnumerable
{
```

```
T this[int index] { get; }
```

Because the type parameter for these interfaces is used only in output positions, it's marked as *covariant*. This allows a list of cats, for instance, to be treated as a read-only list of animals. In contrast, T is not marked as covariant with `ICollection<T>` and `IList<T>`, because T is used in both input and output positions.

#### NOTE

These interfaces represent a read-only view of a collection or list; the underlying implementation might still be writable. Most of the writable (*mutable*) collections implement both the read-only and read/write interfaces.

In addition to letting you work with collections covariantly, the read-only interfaces allow a class to publicly expose a read-only view of a private writable collection. We demonstrate this—along with a better solution—in “[“ReadOnlyCollection<T>”](#)”.

`IReadOnlyList<T>` maps to the Windows Runtime type `IVectorView<T>`.

## The Array Class

The `Array` class is the implicit base class for all single and multidimensional arrays, and it is one of the most fundamental types implementing the standard collection interfaces. The `Array` class

provides type unification, so a common set of methods is available to all arrays, regardless of their declaration or underlying element type.

Because arrays are so fundamental, C# provides explicit syntax for their declaration and initialization, which we described in [Chapter 2](#) and [Chapter 3](#). When an array is declared using C#'s syntax, the CLR implicitly subtypes the `Array` class—synthesizing a *pseudotype* appropriate to the array's dimensions and element types. This pseudotype implements the typed generic collection interfaces, such as `IList<string>`.

The CLR also treats array types specially upon construction, assigning them a contiguous space in memory. This makes indexing into arrays highly efficient, but prevents them from being resized later on.

`Array` implements the collection interfaces up to `IList<T>` in both their generic and nongeneric forms. `IList<T>` itself is implemented explicitly, though, to keep `Array`'s public interface clean of methods such as `Add` or `Remove`, which throw an exception on fixed-length collections such as arrays. The `Array` class does actually offer a static `Resize` method, although this works by creating a new array and then copying over each element. As well as being inefficient, references to the array elsewhere in the program will still point to the original version. A better solution for resizable collections is to use the `List<T>` class (described in the following section).

An array can contain value-type or reference-type elements. Value-type elements are stored in place in the array, so an array of three

long integers (each 8 bytes) will occupy 24 bytes of contiguous memory. A reference-type element, however, occupies only as much space in the array as a reference (4 bytes in a 32-bit environment or 8 bytes in a 64-bit environment). Figure 7-2 illustrates the effect, in memory, of the following program:

```
StringBuilder[] builders = new StringBuilder [5];
builders [0] = new StringBuilder ("builder1");
builders [1] = new StringBuilder ("builder2");
builders [2] = new StringBuilder ("builder3");

long[] numbers = new long [3];
numbers [0] = 12345;
numbers [1] = 54321;
```

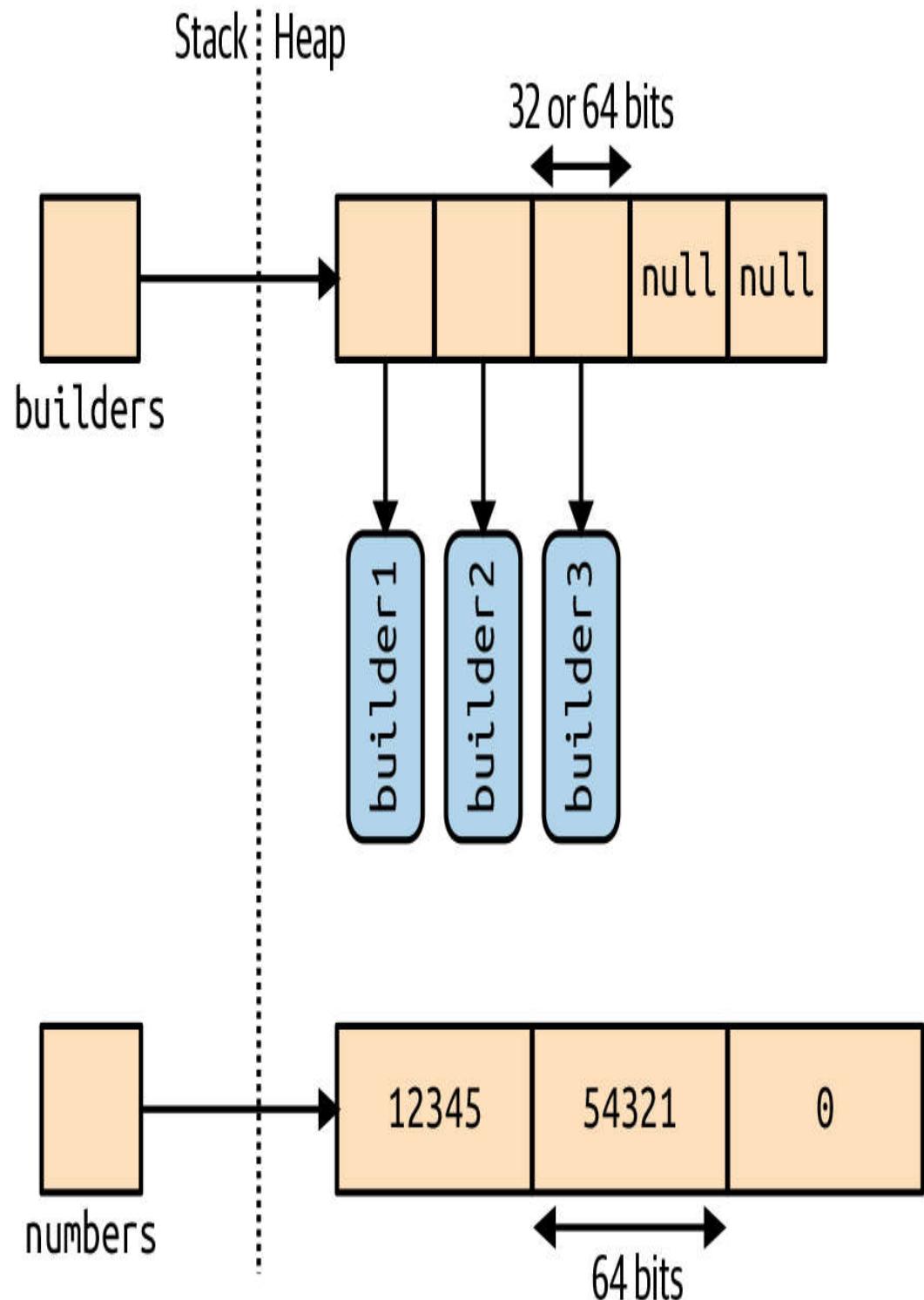


Figure 7-2. Arrays in memory

Because `Array` is a class, arrays are always (themselves) reference types—regardless of the array’s element type. This means that the statement `arrayB = arrayA` results in two variables that reference the same array. Similarly, two distinct arrays will always fail an equality test, unless you employ a *structural equality comparer*, which compares every element of the array:

```
object[] a1 = { "string", 123, true };
object[] a2 = { "string", 123, true };

Console.WriteLine (a1 == a2);                                // False
Console.WriteLine (a1.Equals (a2));                          // False

IStructuralEquatable se1 = a1;
Console.WriteLine (se1.Equals (a2,
    StructuralComparisons.StructuralEqualityComparer));    // True
```

Arrays can be duplicated by calling the `Clone` method: `arrayB = arrayA.Clone()`. However, this results in a shallow clone, meaning that only the memory represented by the array itself is copied. If the array contains value-type objects, the values themselves are copied; if the array contains reference-type objects, just the references are copied (resulting in two arrays whose members reference the same objects). Figure 7-3 demonstrates the effect of adding the following code to our example:

```
StringBuilder[] builders2 = builders;
StringBuilder[] shallowClone = (StringBuilder[]) builders.Clone();
```

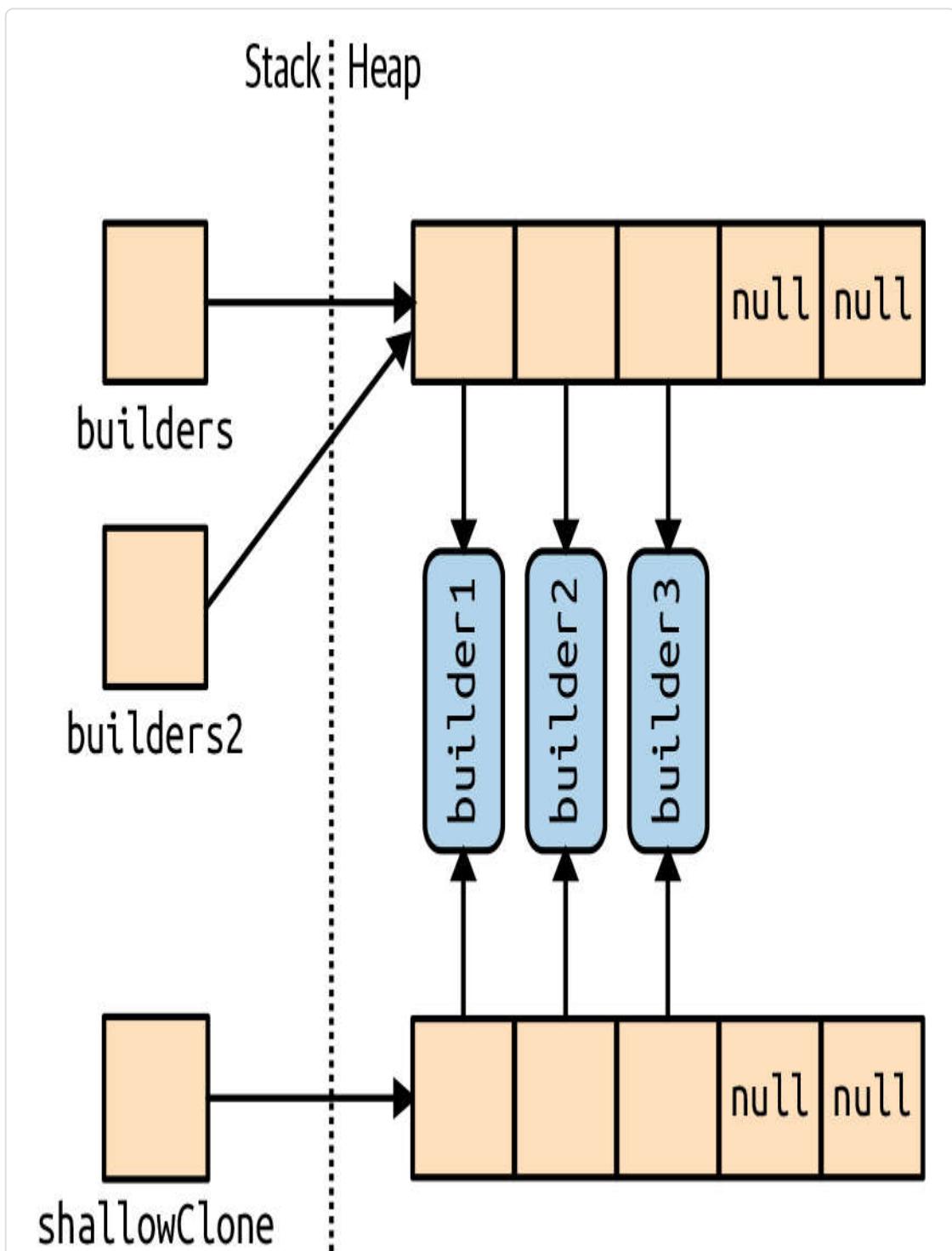


Figure 7-3. Shallow-cloning an array

To create a deep copy—for which reference type subobjects are duplicated—you must loop through the array and clone each element manually. The same rules apply to other .NET collection types.

Although `Array` is designed primarily for use with 32-bit indexers, it also has limited support for 64-bit indexers (allowing an array to theoretically address up to  $2^{64}$  elements) via several methods that accept both `Int32` and `Int64` parameters. These overloads are useless in practice because the CLR does not permit any object—including arrays—to exceed two gigabytes in size (whether running on a 32- or 64-bit environment).

#### NOTE

Many of the methods on the `Array` class that you expect to be instance methods are in fact static methods. This is an odd design decision, and means that you should check for both static and instance methods when looking for a method on `Array`.

## Construction and Indexing

The easiest way to create and index arrays is through C#'s language constructs:

```
int[] myArray = { 1, 2, 3 };
int first = myArray [0];
int last = myArray [myArray.Length - 1];
```

Alternatively, you can instantiate an array dynamically by calling `Array.CreateInstance`. This allows you to specify element type

and rank (number of dimensions) at runtime as well as allowing nonzero-based arrays through specifying a lower bound. Nonzero-based arrays are not compatible with the .NET Common Language Specification (CLS) and should not be exposed as public members in a library that might be consumed by a program written in F# or Visual Basic.

The `GetValue` and `SetValue` methods let you access elements in a dynamically created array (they also work on ordinary arrays):

```
// Create a string array 2 elements in length:  
Array a = Array.CreateInstance (typeof(string), 2);  
a.SetValue ("hi", 0); // → a[0] = "hi";  
a.SetValue ("there", 1); // → a[1] =  
"there";  
string s = (string) a.GetValue (0); // → s = a[0];  
  
// We can also cast to a C# array as follows:  
string[] cSharpArray = (string[]) a;  
string s2 = cSharpArray [0];
```

Zero-indexed arrays created dynamically can be cast to a C# array of a matching or compatible type (compatible by standard array-variance rules). For example, if `Apple` subclasses `Fruit`, `Apple[]` can be cast to `Fruit[]`. This leads to the issue of why `object[]` was not used as the unifying array type rather than the `Array` class. The answer is that `object[]` is incompatible with both multidimensional and value-type arrays (and non-zero-based arrays). An `int[]` array cannot be cast to `object[]`. Hence, we require the `Array` class for full type unification.

`GetValue` and `SetValue` also work on compiler-created arrays, and they are useful when writing methods that can deal with an array of any type and rank. For multidimensional arrays, they accept an *array* of indexers:

```
public object GetValue (params int[] indices)
public void    SetValue (object value, params int[] indices)
```

The following method prints the first element of any array, regardless of rank:

```
void WriteFirstValue (Array a)
{
    Console.Write (a.Rank + "-dimensional; ");

    // The indexers array will automatically initialize to all zeros, so
    // passing it into GetValue or SetValue will get/set the zero-based
    // (i.e., first) element in the array.

    int[] indexers = new int[a.Rank];
    Console.WriteLine ("First value is " + a.GetValue (indexers));
}

void Demo()
{
    int[] oneD = { 1, 2, 3 };
    int[,] twoD = { {5,6}, {8,9} };

    WriteFirstValue (oneD);    // 1-dimensional; first value is 1
    WriteFirstValue (twoD);   // 2-dimensional; first value is 5
}
```

## NOTE

For working with arrays of unknown type but known rank, generics provide an easier and more efficient solution:

```
void WriteFirstValue<T> (T[] array)  
  
{  
  
    Console.WriteLine (array[0]);  
  
}
```

**SetValue** throws an exception if the element is of an incompatible type for the array.

When an array is instantiated, whether via language syntax or **Array.CreateInstance**, its elements are automatically initialized. For arrays with reference-type elements, this means writing nulls; for arrays with value-type elements, this means calling the value-type's default constructor (effectively *zeroing* the members). The **Array** class also provides this functionality on demand via the **Clear** method:

```
public static void Clear (Array array, int index, int length);
```

This method doesn't affect the size of the array. This is in contrast to the usual use of `Clear` (such as in `ICollection<T>.Clear`) whereby the collection is reduced to zero elements.

## Enumeration

Arrays are easily enumerated with a `foreach` statement:

```
int[] myArray = { 1, 2, 3 };
foreach (int val in myArray)
    Console.WriteLine (val);
```

You can also enumerate using the static `Array.ForEach` method, defined as follows:

```
public static void ForEach<T> (T[] array, Action<T> action);
```

This uses an `Action` delegate, with this signature:

```
public delegate void Action<T> (T obj);
```

Here's the first example rewritten with `Array.ForEach`:

```
Array.ForEach (new[] { 1, 2, 3 }, Console.WriteLine);
```

## Length and Rank

`Array` provides the following methods and properties for querying length and rank:

```
public int GetLength      (int dimension);
public long GetLongLength (int dimension);

public int Length        { get; }
public long LongLength   { get; }

public int GetLowerBound (int dimension);
public int GetUpperBound (int dimension);

public int Rank { get; }    // Returns number of dimensions in array
```

`GetLength` and `GetLongLength` return the length for a given dimension (0 for a single-dimensional array), and `Length` and `LongLength` return the total number of elements in the array—all dimensions included.

`GetLowerBound` and `GetUpperBound` are useful with nonzero-indexed arrays. `GetUpperBound` returns the same result as adding `GetLowerBound` to `GetLength` for any given dimension.

## Searching

The `Array` class offers a range of methods for finding elements within a one-dimensional array:

### BinarySearch methods

For rapidly searching a sorted array for a particular item

### IndexOf/LastIndex methods

For searching unsorted arrays for a particular item

## `Find`/`FindLast`/`FindIndex`/`FindLastIndex`/`FindAll`/`Exists`/`TrueForAll`

For searching unsorted arrays for item(s) that satisfy a given `Predicate<T>`

None of the array-searching methods throws an exception if the specified value is not found. Instead, if an item is not found, methods returning an integer return `-1` (assuming a zero-indexed array), and methods returning a generic type return the type's default value (e.g., `0` for an `int`, or `null` for a `string`).

The binary search methods are fast, but they work only on sorted arrays and require that the elements be compared for *order* rather than simply *equality*. To this effect, the binary search methods can accept an `IComparer` or `IComparer<T>` object to arbitrate on ordering decisions (see “[Plugging in Equality and Order](#)”). This must be consistent with any comparer used in originally sorting the array. If no comparer is provided, the type's default ordering algorithm will be applied based on its implementation of `IComparable`/`IComparable<T>`.

The `IndexOf` and `LastIndexOf` methods perform a simple enumeration over the array, returning the position of the first (or last) element that matches the given value.

The predicate-based searching methods allow a method delegate or lambda expression to arbitrate on whether a given element is a *match*. A predicate is simply a delegate accepting an object and returning `true` or `false`:

```
public delegate bool Predicate<T> (T object);
```

In the following example, we search an array of strings for a name containing the letter “a”:

```
static void Main()
{
    string[] names = { "Rodney", "Jack", "Jill" };
    string match = Array.Find (names, ContainsA);
    Console.WriteLine (match);      // Jack
}
static bool ContainsA (string name) { return name.Contains ("a"); }
```

Here's the same code shortened with an anonymous method:

```
string[] names = { "Rodney", "Jack", "Jill" };
string match = Array.Find (names, delegate (string name)
    { return name.Contains ("a"); } );
```

A lambda expression shortens it further:

```
string[] names = { "Rodney", "Jack", "Jill" };
string match = Array.Find (names, n => n.Contains ("a"));      //
Jack
```

`FindAll` returns an array of all items satisfying the predicate. In fact, it's equivalent to `Enumerable.Where` in the `System.Linq` namespace, except that `FindAll` returns an array of matching items rather than an `IEnumerable<T>` of the same.

`Exists` returns `true` if any array member satisfies the given predicate, and is equivalent to `Any` in `System.Linq.Enumerable`.

`TrueForAll` returns `true` if all items satisfy the predicate, and is equivalent to `All` in `System.Linq.Enumerable`.

## Sorting

`Array` has the following built-in sorting methods:

```
// For sorting a single array:  
  
public static void Sort<T> (T[] array);  
public static void Sort (Array array);  
  
// For sorting a pair of arrays:  
  
public static void Sort<TKey,TValue> (TKey[] keys, TValue[] items);  
public static void Sort (Array keys, Array items);
```

Each of these methods is additionally overloaded to also accept the following:

```
int index          // Starting index at which to begin sorting  
int length        // Number of elements to sort  
IComparer<T> comparer // Object making ordering decisions  
Comparison<T> comparison // Delegate making ordering decisions
```

The following illustrates the simplest use of `Sort`:

```
int[] numbers = { 3, 2, 1 };  
Array.Sort (numbers);           // Array is now { 1, 2, 3 }
```

The methods accepting a pair of arrays work by rearranging the items of each array in tandem, basing the ordering decisions on the first array. In the next example, both the numbers and their corresponding words are sorted into numerical order:

```
int[] numbers = { 3, 2, 1 };
string[] words = { "three", "two", "one" };
Array.Sort (numbers, words);

// numbers array is now { 1, 2, 3 }
// words    array is now { "one", "two", "three" }
```

`Array.Sort` requires that the elements in the array implement `IComparable` (see “Order Comparison” in Chapter 6). This means that most built-in C# types (such as integers, as in the preceding example) can be sorted. If the elements are not intrinsically comparable or you want to override the default ordering, you must provide `Sort` with a custom `comparison` provider that reports on the relative position of two elements. There are ways to do this:

- Via a helper object that implements `IComparer/IComparer<T>` (see “Plugging in Equality and Order”)
- Via a `Comparison` delegate:

```
public delegate int Comparison<T> (T x, T y);
```

The `Comparison` delegate follows the same semantics as `IComparer<T>.CompareTo`: if `x` comes before `y`, a negative integer is

returned; if  $x$  comes after  $y$ , a positive integer is returned; if  $x$  and  $y$  have the same sorting position,  $0$  is returned.

In the following example, we sort an array of integers such that the odd numbers come first:

```
int[] numbers = { 1, 2, 3, 4, 5 };
Array.Sort (numbers, (x, y) => x % 2 == y % 2 ? 0 : x % 2 == 1 ? -1 :
1);

// numbers array is now { 1, 3, 5, 2, 4 }
```

### NOTE

As an alternative to calling `Sort`, you can use LINQ's `OrderBy` and `ThenBy` operators. Unlike `Array.Sort`, the LINQ operators don't alter the original array, instead emitting the sorted result in a fresh `IEnumerable<T>` sequence.

## Reversing Elements

The following `Array` methods reverse the order of all—or a portion of—elements in an array:

```
public static void Reverse (Array array);
public static void Reverse (Array array, int index, int length);
```

## Copying

`Array` provides four methods to perform shallow copying: `Clone`, `CopyTo`, `Copy`, and `ConstrainedCopy`. The former two are instance

methods; the latter two are static methods.

The `Clone` method returns a whole new (shallow-copied) array. The `CopyTo` and `Copy` methods copy a contiguous subset of the array. Copying a multidimensional rectangular array requires you to map the multidimensional index to a linear index. For example, the middle square (`position[1,1]`) in a  $3 \times 3$  array is represented with the index 4, from the calculation:  $1 \times 3 + 1$ . The source and destination ranges can overlap without causing a problem.

`ConstrainedCopy` performs an *atomic* operation: if all of the requested elements cannot be successfully copied (due to a type error, for instance), the operation is rolled back.

`Array` also provides an `AsReadOnly` method which returns a wrapper that prevents elements from being reassigned.

## Converting and Resizing

`Array.ConvertAll` creates and returns a new array of element type `TOutput`, calling the supplied `Converter` delegate to copy over the elements. `Converter` is defined as follows:

```
public delegate TOutput Converter<TInput,TOutput> (TInput input)
```

The following converts an array of floats to an array of integers:

```
float[] reals = { 1.3f, 1.5f, 1.8f };
int[] wholes = Array.ConvertAll (reals, r => Convert.ToInt32 (r));
```

```
// wholes array is { 1, 2, 2 }
```

The `Resize` method works by creating a new array and copying over the elements, returning the new array via the reference parameter. However, any references to the original array in other objects will remain unchanged.

#### NOTE

The `System.Linq` namespace offers an additional buffet of extension methods suitable for array conversion. These methods return an `IEnumerable<T>`, which you can convert back to an array via `Enumerable`'s `ToArray` method.

## Lists, Queues, Stacks, and Sets

.NET Core provides a basic set of concrete collection classes that implement the interfaces described in this chapter. This section concentrates on the *list-like* collections (versus the *dictionary-like* collections, which we cover in “[Dictionaries](#)”). As with the interfaces we discussed previously, you usually have a choice of generic or nongeneric versions of each type. In terms of flexibility and performance, the generic classes win, making their nongeneric counterparts redundant except for backward compatibility. This differs from the situation with collection interfaces, for which the nongeneric versions are still occasionally useful.

Of the classes described in this section, the generic `List` class is the most commonly used.

## List<T> and ArrayList

The generic `List` and nongeneric `ArrayList` classes provide a dynamically sized array of objects and are among the most commonly used of the collection classes. `ArrayList` implements `IList`, whereas `List<T>` implements both `IList` and `IList<T>` (and the read-only version, `IReadOnlyList<T>`). Unlike with arrays, all interfaces are implemented publicly, and methods such as `Add` and `Remove` are exposed and work as you would expect.

Internally, `List<T>` and `ArrayList` work by maintaining an internal array of objects, replaced with a larger array upon reaching capacity. Appending elements is efficient (because there is usually a free slot at the end), but inserting elements can be slow (because all elements after the insertion point must be shifted to make a free slot), as can removing elements (especially near the start). As with arrays, searching is efficient if the `BinarySearch` method is used on a list that has been sorted, but it is otherwise inefficient because each item must be individually checked.

### NOTE

`List<T>` is up to several times faster than `ArrayList` if `T` is a value type, because `List<T>` avoids the overhead of boxing and unboxing elements.

`List<T>` and `ArrayList` provide constructors that accept an existing collection of elements: these copy each element from the existing collection into the new `List<T>` or `ArrayList`:

```
public class List<T> : IList<T>, IReadOnlyList<T>
{
    public List ();
    public List (IEnumerable<T> collection);
    public List (int capacity);

    // Add+Insert
    public void Add      (T item);
    public void AddRange (IEnumerable<T> collection);
    public void Insert   (int index, T item);
    public void InsertRange (int index, IEnumerable<T> collection);

    // Remove
    public bool Remove   (T item);
    public void RemoveAt (int index);
    public void RemoveRange (int index, int count);
    public int RemoveAll (Predicate<T> match);

    // Indexing
    public T this [int index] { get; set; }
    public List<T> GetRange (int index, int count);
    public Enumerator<T> GetEnumerator();

    // Exporting, copying, and converting:
    public T[] ToArray();
    public void CopyTo (T[] array);
    public void CopyTo (T[] array, int arrayIndex);
    public void CopyTo (int index, T[] array, int arrayIndex, int count);
    public ReadOnlyCollection<T> AsReadOnly();
    public List<TOutput> ConvertAll<TOutput> (Converter <T,TOutput>
                                                converter);

    // Other:
    public void Reverse();           // Reverses order of elements in
list.
    public int Capacity { get; set; } // Forces expansion of internal
array.
    public void TrimExcess();        // Trims internal array back to
size.
```

```

    public void Clear();           // Removes all elements, so Count =
0.
}

public delegate TOutput Converter <TInput, TOutput> (TInput input);

```

In addition to these members, `List<T>` provides instance versions of all of `Array`'s searching and sorting methods.

The following code demonstrates `List`'s properties and methods (for examples of searching and sorting, see “[The Array Class](#)”):

```

var words = new List<string>();      // New string-typed list

words.Add ("melon");
words.Add ("avocado");
words.AddRange (new[] { "banana", "plum" } );
words.Insert (0, "lemon");           // Insert at start
words.InsertRange (0, new[] { "peach", "nashi" }); // Insert at start

words.Remove ("melon");
words.RemoveAt (3);                // Remove the 4th element
words.RemoveRange (0, 2);          // Remove first 2 elements

// Remove all strings starting in 'n':
words.RemoveAll (s => s.StartsWith ("n"));

Console.WriteLine (words [0]);        // first word
Console.WriteLine (words [words.Count - 1]); // last word
foreach (string s in words) Console.WriteLine (s); // all words
List<string> subset = words.GetRange (1, 2); // 2nd->3rd
words

string[] wordsArray = words.ToArray(); // Creates a new typed array

// Copy first two elements to the end of an existing array:

```

```
string[] existing = new string [1000];
words.CopyTo (0, existing, 998, 2);

List<string> upperCaseWords = words.ConvertAll (s => s.ToUpper());
List<int> lengths = words.ConvertAll (s => s.Length);
```

The nongeneric `ArrayList` class requires clumsy casts—as the following example demonstrates:

```
ArrayList al = new ArrayList();
al.Add ("hello");
string first = (string) al [0];
string[] strArr = (string[]) al.ToArray (typeof (string));
```

Such casts cannot be verified by the compiler; the following compiles successfully but then fails at runtime:

```
int first = (int) al [0];    // Runtime exception
```

### NOTE

An `ArrayList` is functionally similar to `List<object>`. Both are useful when you need a list of mixed-type elements that share no common base type (other than `object`). A possible advantage of choosing an `ArrayList`, in this case, would be if you need to deal with the list using reflection ([Chapter 19](#)). Reflection is easier with a nongeneric `ArrayList` than a `List<object>`.

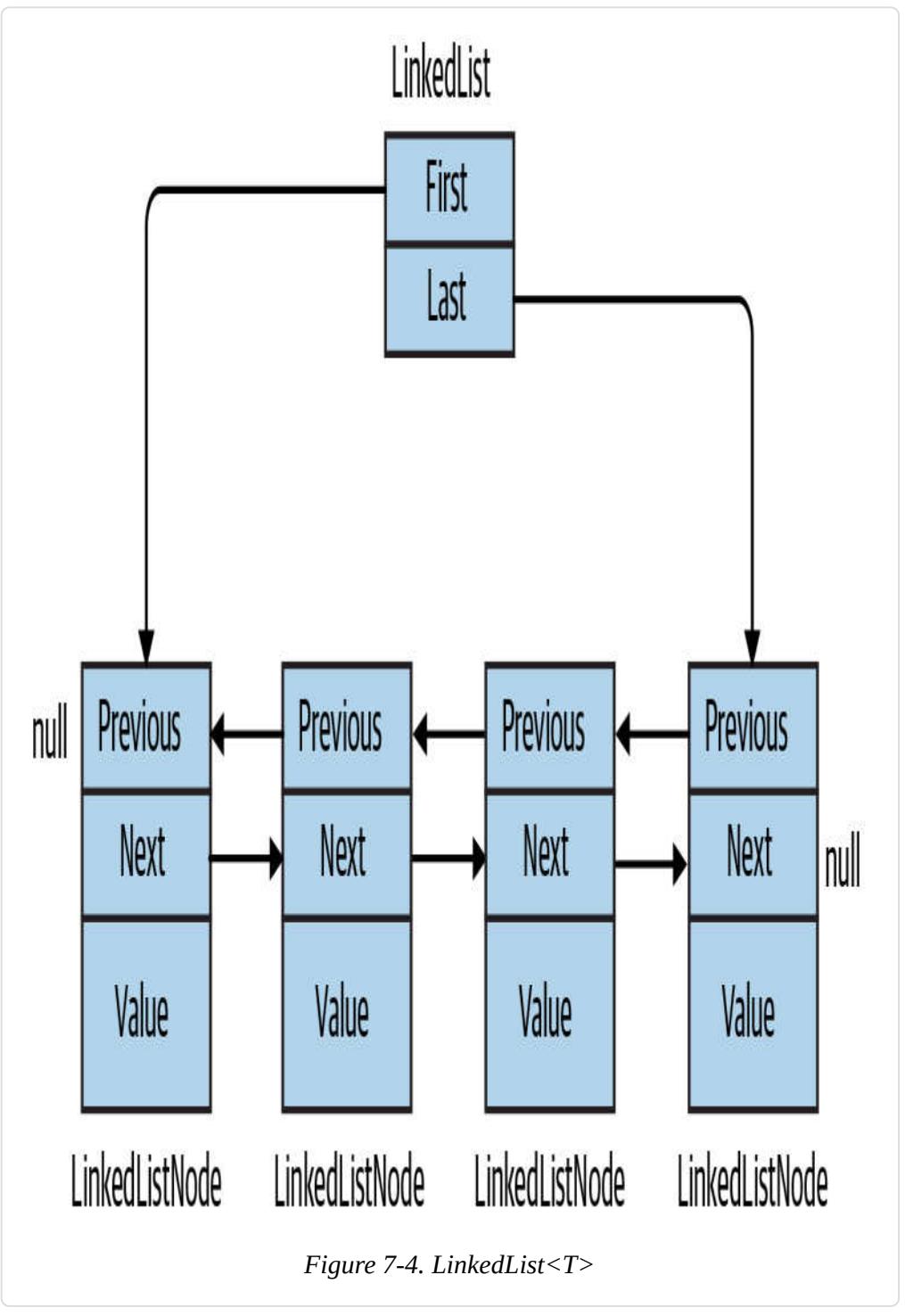
If you import the `System.Linq` namespace, you can convert an `ArrayList` to a generic `List` by calling `Cast` and then `ToList`:

```
ArrayList al = new ArrayList();
al.AddRange (new[] { 1, 5, 9 } );
List<int> list = al.Cast<int>().ToList();
```

`Cast` and `ToList` are extension methods in the `System.Linq.Enumerable` class.

## **LinkedList<T>**

`LinkedList<T>` is a generic doubly linked list (see [Figure 7-4](#)). A doubly linked list is a chain of nodes in which each references the node before, the node after, and the actual element. Its main benefit is that an element can always be inserted efficiently anywhere in the list because it just involves creating a new node and updating a few references. However, finding where to insert the node in the first place can be slow because there's no intrinsic mechanism to index directly into a linked list; each node must be traversed, and binary-chop searches are not possible.



`LinkedList<T>` implements `IEnumerable<T>` and `ICollection<T>` (and their nongeneric versions), but not `IList<T>` because access by index is not supported. List nodes are implemented via the following class:

```
public sealed class LinkedListNode<T>
{
    public LinkedList<T> List { get; }
    public LinkedListNode<T> Next { get; }
    public LinkedListNode<T> Previous { get; }
    public T Value { get; set; }
}
```

When adding a node, you can specify its position either relative to another node or at the start/end of the list. `LinkedList<T>` provides the following methods for this:

```
public void AddFirst(LinkedListNode<T> node);
public LinkedListNode<T> AddFirst (T value);

public void AddLast (LinkedListNode<T> node);
public LinkedListNode<T> AddLast (T value);

public void AddAfter (LinkedListNode<T> node, LinkedListNode<T>
newNode);
public LinkedListNode<T> AddAfter (LinkedListNode<T> node, T value);

public void AddBefore (LinkedListNode<T> node, LinkedListNode<T>
newNode);
public LinkedListNode<T> AddBefore (LinkedListNode<T> node, T value);
```

Similar methods are provided to remove elements:

```
public void Clear();

public void RemoveFirst();
public void RemoveLast();
```

```
public bool Remove (T value);
public void Remove (LinkedListNode<T> node);
```

`LinkedList<T>` has internal fields to keep track of the number of elements in the list as well as the head and tail of the list. These are exposed in the following public properties:

```
public int Count { get; }                      // Fast
public LinkedListNode<T> First { get; }        // Fast
public LinkedListNode<T> Last { get; }          // Fast
```

`LinkedList<T>` also supports the following searching methods (each requiring that the list be internally enumerated):

```
public bool Contains (T value);
public LinkedListNode<T> Find (T value);
public LinkedListNode<T> FindLast (T value);
```

Finally, `LinkedList<T>` supports copying to an array for indexed processing and obtaining an enumerator to support the `foreach` statement:

```
public void CopyTo (T[] array, int index);
public Enumerator<T> GetEnumerator();
```

Here's a demonstration on the use of `LinkedList<string>`:

```
var tune = new LinkedList<string>();
tune.AddFirst ("do");                                // do
tune.AddLast ("so");                                 // do - so
```

```

tune.AddAfter (tune.First, "re");           // do - re - so
tune.AddAfter (tune.First.Next, "mi");       // do - re - mi - so
tune.AddBefore (tune.Last, "fa");           // do - re - mi - fa -
so

tune.RemoveFirst();                         // re - mi - fa - so
tune.RemoveLast();                          // re - mi - fa

LinkedListNode<string> miNode = tune.Find ("mi");
tune.Remove (miNode);                      // re - fa
tune.AddFirst (miNode);                    // mi - re - fa

foreach (string s in tune) Console.WriteLine (s);

```

## Queue<T> and Queue

`Queue<T>` and `Queue` are first-in, first-out (FIFO) data structures, providing methods to `Enqueue` (add an item to the tail of the queue) and `Dequeue` (retrieve and remove the item at the head of the queue). A `Peek` method is also provided to return the element at the head of the queue without removing it, and there is a `Count` property (useful in checking that elements are present before dequeuing).

Although queues are enumerable, they do not implement `IList<T>/IList`, because members cannot be accessed directly by index. A `ToArray` method is provided, however, for copying the elements to an array from which they can be randomly accessed:

```

public class Queue<T> : IEnumerable<T>, ICollection, IEnumerable
{
    public Queue();
    public Queue (IEnumerable<T> collection); // Copies existing

```

```
elements
    public Queue (int capacity);           // To lessen auto-resizing
    public void Clear();
    public bool Contains (T item);
    public void CopyTo (T[] array, int arrayIndex);
    public int Count { get; }
    public T Dequeue();
    public void Enqueue (T item);
    public Enumerator<T> GetEnumerator();      // To support foreach
    public T Peek();
    public T[] ToArray();
    public void TrimExcess();
}
```

The following is an example of using `Queue<int>`:

```
var q = new Queue<int>();
q.Enqueue (10);
q.Enqueue (20);
int[] data = q.ToArray();           // Exports to an array
Console.WriteLine (q.Count);        // "2"
Console.WriteLine (q.Peek());        // "10"
Console.WriteLine (q.Dequeue());     // "10"
Console.WriteLine (q.Dequeue());     // "20"
Console.WriteLine (q.Dequeue());     // Throws an exception (queue empty)
```

Queues are implemented internally using an array that's resized as required—much like the generic `List` class. The queue maintains indexes that point directly to the head and tail elements; therefore, enqueueing and dequeuing are extremely quick operations (except when an internal resize is required).

## Stack<T> and Stack

`Stack<T>` and `Stack` are last-in, first-out (LIFO) data structures, providing methods to `Push` (add an item to the top of the stack) and `Pop` (retrieve and remove an element from the top of the stack). A nondestructive `Peek` method is also provided, as is a `Count` property and a `ToArrayList` method for exporting the data for random access:

```
public class Stack<T> : IEnumerable<T>, ICollection, IEnumerable
{
    public Stack();
    public Stack (IEnumerable<T> collection); // Copies existing
elements
    public Stack (int capacity); // Lessens auto-resizing
    public void Clear();
    public bool Contains (T item);
    public void CopyTo (T[] array, int arrayIndex);
    public int Count { get; }
    public Enumerator<T> GetEnumerator(); // To support foreach
    public T Peek();
    public T Pop();
    public void Push (T item);
    public T[] ToArrayList();
    public void TrimExcess();
}
```

The following example demonstrates `Stack<int>`:

```
var s = new Stack<int>();
s.Push (1); // Stack = 1
s.Push (2); // Stack = 1,2
s.Push (3); // Stack = 1,2,3
Console.WriteLine (s.Count); // Prints 3
Console.WriteLine (s.Peek()); // Prints 3, Stack = 1,2,3
Console.WriteLine (s.Pop()); // Prints 3, Stack = 1,2
Console.WriteLine (s.Pop()); // Prints 2, Stack = 1
```

```
Console.WriteLine (s.Pop());      // Prints 1, Stack = <empty>
Console.WriteLine (s.Pop());      // Throws exception
```

Stacks are implemented internally with an array that's resized as required, as with `Queue<T>` and `List<T>`.

## BitArray

A `BitArray` is a dynamically sized collection of compacted `bool` values. It is more memory efficient than both a simple array of `bool` and a generic `List` of `bool` because it uses only one bit for each value, whereas the `bool` type otherwise occupies one byte for each value.

`BitArray`'s indexer reads and writes individual bits:

```
var bits = new BitArray(2);
bits[1] = true;
```

There are four bitwise operator methods (`And`, `Or`, `Xor`, and `Not`). All but the last accept another `BitArray`:

```
bits.Xor (bits);           // Bitwise exclusive-OR bits with itself
Console.WriteLine (bits[1]); // False
```

## HashSet<T> and SortedSet<T>

`HashSet<T>` and `SortedSet<T>` are generic collections new to Framework 3.5 and 4.0, respectively. Both have the following distinguishing features:

- Their `Contains` methods execute quickly using a hash-based lookup.
- They do not store duplicate elements and silently ignore requests to add duplicates.
- You cannot access an element by position.

`SortedSet<T>` keeps elements in order, whereas `HashSet<T>` does not.

#### NOTE

The commonality of these types is captured by the interface `ISet<T>`.

For historical reasons, `HashSet<T>` resides in `System.Core.dll` (whereas `SortedSet<T>` and `ISet<T>` reside in `System.dll`).

`HashSet<T>` is implemented with a hashtable that stores just keys; `SortedSet<T>` is implemented with a red/black tree.

Both collections implement `ICollection<T>` and offer methods that you would expect, such as `Contains`, `Add`, and `Remove`. In addition, there's a predicate-based removal method called `RemoveWhere`.

The following constructs a `HashSet<char>` from an existing collection, tests for membership, and then enumerates the collection (notice the absence of duplicates):

```
var letters = new HashSet<char> ("the quick brown fox");
```

```
Console.WriteLine (letters.Contains ('t'));      // true
Console.WriteLine (letters.Contains ('j'));      // false

foreach (char c in letters) Console.Write (c);    // the quickbrownfx
```

(The reason we can pass a `string` into `HashSet<char>`'s constructor is because `string` implements `IEnumerable<char>`.)

The really interesting methods are the set operations. The following set operations are *destructive* in that they modify the set:

```
public void UnionWith          (IEnumerable<T> other);    // Adds
public void IntersectWith      (IEnumerable<T> other);    // Removes
public void ExceptWith         (IEnumerable<T> other);    // Removes
public void SymmetricExceptWith (IEnumerable<T> other);    // Removes
```

whereas the following methods simply query the set and so are nondestructive:

```
public bool IsSubsetOf        (IEnumerable<T> other);
public bool IsProperSubsetOf  (IEnumerable<T> other);
public bool IsSupersetOf      (IEnumerable<T> other);
public bool IsProperSupersetOf (IEnumerable<T> other);
public bool Overlaps          (IEnumerable<T> other);
public bool SetEquals         (IEnumerable<T> other);
```

`UnionWith` adds all the elements in the second set to the original set (excluding duplicates). `IntersectWith` removes the elements that are not in both sets. We can extract all of the vowels from our set of characters as follows:

```
var letters = new HashSet<char> ("the quick brown fox");
letters.IntersectWith ("aeiou");
foreach (char c in letters) Console.Write (c);      // eui
```

**ExceptWith** removes the specified elements from the source set.  
Here, we strip all vowels from the set:

```
var letters = new HashSet<char> ("the quick brown fox");
letters.ExceptWith ("aeiou");
foreach (char c in letters) Console.Write (c);      // th qckbrwnfx
```

**SymmetricExceptWith** removes all but the elements that are unique to one set or the other:

```
var letters = new HashSet<char> ("the quick brown fox");
letters.SymmetricExceptWith ("the lazy brown fox");
foreach (char c in letters) Console.Write (c);      // quicklazy
```

Note that because `HashSet<T>` and `SortedSet<T>` implement `IEnumerable<T>`, you can use another type of set (or collection) as the argument to any of the set operation methods.

`SortedSet<T>` offers all the members of `HashSet<T>`, plus the following:

```
public virtual SortedSet<T> GetViewBetween (T lowerValue, T upperValue)
public IEnumerable<T> Reverse()
public T Min { get; }
public T Max { get; }
```

`SortedSet<T>` also accepts an optional `IComparer<T>` in its constructor (rather than an *equality comparer*).

Here's an example of loading the same letters into a `SortedSet<char>`:

```
var letters = new SortedSet<char> ("the quick brown fox");
foreach (char c in letters) Console.Write (c);    // bcefhi knoqr tuwx
```

Following on from this, we can obtain the letters in the set between *f* and *j* as follows:

```
foreach (char c in letters.GetViewBetween ('f', 'j'))
    Console.Write (c);                                // fhi
```

## Dictionaries

A dictionary is a collection in which each element is a key/value pair. Dictionaries are most commonly used for lookups and sorted lists.

The Framework defines a standard protocol for dictionaries, via the interfaces `IDictionary` and `IDictionary <TKey, TValue>` as well as a set of general-purpose dictionary classes. The classes each differ in the following regard:

- Whether or not items are stored in sorted sequence
- Whether or not items can be accessed by position (index) as well as by key

- Whether generic or nongeneric
- Whether it's fast or slow to retrieve items by key from a large dictionary

Table 7-1 summarizes each of the dictionary classes and how they differ in these respects. The performance times are in milliseconds and based on performing 50,000 operations on a dictionary with integer keys and values on a 1.5 GHz PC. (The differences in performance between generic and nongeneric counterparts using the same underlying collection structure are due to boxing, and show up only with value-type elements.)

*T*  
*a*  
*b*  
*l*  
*e*

*7*  
-  
*1*  
.   
*D*  
*i*  
*c*  
*t*  
*i*  
*o*  
*n*  
*a*  
*r*  
*y*

c  
l  
a  
s  
s  
e  
s

Ty pe	Inter nal struc ture	Retrie ve by index ?	Memory overhead (avg. bytes per item)	Speed: random insertio n	Speed: sequential insertion	Speed: retrieval by key
<b>Unsorted</b>						
Dictionary <K,V>	Hashtable	No	2	30	30	20
Hashtable	Hashtable	No	3	50	50	30
ListDictionary	Linked list	No	3 6	50,00 0	50,00 0	50,00 0
OrderedDictionary	Hashtable + array	Yes	5 9	70	70	40
<b>Sorted</b>						
SortedDictionary <K,V>	Red/black tree	No	2 0	130	100	120
SortedList <K,V>	2xArray	Yes	2	3,300	30	40
SortedList	2xArray	Yes	2 7	4,500	100	180

In Big-O notation, retrieval time by key is as follows:

- O(1) for `Hashtable`, `Dictionary`, and `OrderedDictionary`
- O(log n) for `SortedDictionary` and `SortedList`
- O(n) for `ListDictionary` (and nondictionary types such as `List<T>`)

*n* is the number of elements in the collection.

## **IDictionary< TKey, TValue >**

`IDictionary< TKey, TValue >` defines the standard protocol for all key/value-based collections. It extends `ICollection< T >` by adding methods and properties to access elements based on a key of arbitrary type:

```
public interface IDictionary < TKey, TValue > :  
    ICollection < KeyValuePair < TKey, TValue > >, IEnumerable  
{  
    bool ContainsKey ( TKey key );  
    bool TryGetValue ( TKey key, out TValue value );  
    void Add ( TKey key, TValue value );  
    bool Remove ( TKey key );  
  
    TValue this [ TKey key ] { get; set; } // Main indexer - by key  
    ICollection < TKey > Keys { get; } // Returns just keys  
    ICollection < TValue > Values { get; } // Returns just values  
}
```

## NOTE

From Framework 4.5, there's also an interface called `IReadOnlyDictionary<TKey, TValue>`, which defines the read-only subset of dictionary members. This maps to the Windows Runtime type `IMapView<K,V>` and was introduced primarily for that reason.

To add an item to a dictionary, you either call `Add` or use the index's set accessor—the latter adds an item to the dictionary if the key is not already present (or updates the item if it is present). Duplicate keys are forbidden in all dictionary implementations, so calling `Add` twice with the same key throws an exception.

To retrieve an item from a dictionary, use either the indexer or the `TryGetValue` method. If the key doesn't exist, the indexer throws an exception, whereas `TryGetValue` returns `false`. You can test for membership explicitly by calling `ContainsKey`; however, this incurs the cost of two lookups if you subsequently retrieve the item.

Enumerating directly over an `IDictionary<TKey, TValue>` returns a sequence of `KeyValuePair` structs:

```
public struct KeyValuePair <TKey, TValue>
{
    public TKey Key { get; }
    public TValue Value { get; }
}
```

You can enumerate over just the keys or values via the dictionary's `Keys/Values` properties.

We demonstrate the use of this interface with the generic **Dictionary** class in the following section.

## IDictionary

The nongeneric **IDictionary** interface is the same in principle as **IDictionary<TKey, TValue>**, apart from two important functional differences. It's important to be aware of these differences, because **IDictionary** appears in legacy code (including the .NET Framework itself in places):

- Retrieving a nonexistent key via the indexer returns null (rather than throwing an exception).
- **Contains** tests for membership rather than **ContainsKey**.

Enumerating over a nongeneric **IDictionary** returns a sequence of **DictionaryEntry** structs:

```
public struct DictionaryEntry
{
    public object Key { get; set; }
    public object Value { get; set; }
}
```

## Dictionary<TKey,TValue> and Hashtable

The generic **Dictionary** class is one of the most commonly used collections (along with the **List<T>** collection). It uses a hashtable data structure to store keys and values, and it is fast and efficient.

## NOTE

The nongeneric version of `Dictionary<TKey, TValue>` is called `Hashtable`; there is no nongeneric class called `Dictionary`. When we refer simply to `Dictionary`, we mean the generic `Dictionary<TKey, TValue>` class.

`Dictionary` implements both the generic and nongeneric `IDictionary` interfaces, the generic `IDictionary` being exposed publicly. `Dictionary` is, in fact, a “textbook” implementation of the generic `IDictionary`.

Here's how to use it:

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

d.Add("One", 1);
d["Two"] = 2;      // adds to dictionary because "two" not already
present
d["Two"] = 22;    // updates dictionary because "two" is now present
d["Three"] = 3;

Console.WriteLine (d["Two"]);           // Prints "22"
Console.WriteLine (d.ContainsKey ("One")); // true (fast operation)
Console.WriteLine (d.ContainsValue (3));   // true (slow operation)
int val = 0;
if (!d.TryGetValue ("onE", out val))
    Console.WriteLine ("No val");        // "No val" (case
sensitive)

// Three different ways to enumerate the dictionary:

foreach (KeyValuePair<string, int> kv in d)          // One; 1
    Console.WriteLine (kv.Key + " ; " + kv.Value);       // Two; 22
```

```
// Three; 3

foreach (string s in d.Keys) Console.Write (s);      // OneTwoThree
Console.WriteLine();
foreach (int i in d.Values) Console.Write (i);      // 1223
```

Its underlying hashtable works by converting each element's key into an integer hashcode—a pseudo-unique value—and then applying an algorithm to convert the hashcode into a hash key. This hash key is used internally to determine which “bucket” an entry belongs to. If the bucket contains more than one value, a linear search is performed on the bucket. A good hash function does not strive to return strictly unique hashcodes (which would usually be impossible); it strives to return hashcodes that are evenly distributed across the 32-bit integer space. This avoids the scenario of ending up with a few very large (and inefficient) buckets.

A dictionary can work with keys of any type, provided it's able to determine equality between keys and obtain hashcodes. By default, equality is determined via the key's `object.Equals` method, and the pseudo-unique hashcode is obtained via the key's `GetHashCode` method. You can change this behavior either by overriding these methods or by providing an `IEqualityComparer` object when constructing the dictionary. A common application of this is to specify a case-insensitive equality comparer when using string keys:

```
var d = new Dictionary<string, int> (StringComparer.OrdinalIgnoreCase);
```

We discuss this further in “[Plugging in Equality and Order](#)”.

As with many other types of collections, you can improve the performance of a dictionary slightly by specifying the collection's expected size in the constructor, avoiding or lessening the need for internal resizing operations.

The nongeneric version is named `Hashtable` and is functionally similar, apart from differences stemming from it exposing the nongeneric `IDictionary` interface discussed previously.

The downside to `Dictionary` and `Hashtable` is that the items are not sorted. Furthermore, the original order in which the items were added is not retained. As with all dictionaries, duplicate keys are not allowed.

#### NOTE

When the generic collections were introduced in Framework 2.0, the CLR team chose to name them according to what they represent (`Dictionary`, `List`) rather than how they are internally implemented (`Hashtable`, `ArrayList`). Although this is good because it gives them the freedom to later change the implementation, it also means that the *performance contract* (often the most important criterion in choosing one kind of collection over another) is no longer captured in the name.

## OrderedDictionary

An `OrderedDictionary` is a nongeneric dictionary that maintains elements in the same order that they were added. With an `OrderedDictionary`, you can access elements both by index and by key.

## NOTE

An `OrderedDictionary` is not a *sorted* dictionary.

An `OrderedDictionary` is a combination of a `Hashtable` and an `ArrayList`. This means that it has all the functionality of a `Hashtable`, plus functions such as `RemoveAt`, and an integer indexer. It also exposes `Keys` and `Values` properties that return elements in their original order.

This class was introduced in .NET 2.0, yet peculiarly, there's no generic version.

## ListDictionary and HybridDictionary

`ListDictionary` uses a singly linked list to store the underlying data. It doesn't provide sorting, although it does preserve the original entry order of the items. `ListDictionary` is extremely slow with large lists. Its only real "claim to fame" is its efficiency with very small lists (fewer than 10 items).

`HybridDictionary` is a `ListDictionary` that automatically converts to a `Hashtable` upon reaching a certain size, to address `ListDictionary`'s problems with performance. The idea is to get a low memory footprint when the dictionary is small, and good performance when the dictionary is large. However, given the overhead in converting from one to the other—and the fact that a `Dictionary` is not excessively heavy or slow in either scenario—you wouldn't suffer unreasonably by using a `Dictionary` to begin with.

Both classes come only in nongeneric form.

## Sorted Dictionaries

The Framework provides two dictionary classes internally structured such that their content is always sorted by key:

- `SortedDictionary< TKey , TValue >`
- `SortedList< TKey , TValue >`<sup>1</sup>

(In this section, we abbreviate `< TKey , TValue >` to `< , >`.)

`SortedDictionary< , >` uses a red/black tree: a data structure designed to perform consistently well in any insertion or retrieval scenario.

`SortedList< , >` is implemented internally with an ordered array pair, providing fast retrieval (via a binary-chop search) but poor insertion performance (because existing values need to be shifted to make room for a new entry).

`SortedDictionary< , >` is much faster than `SortedList< , >` at inserting elements in a random sequence (particularly with large lists). `SortedList< , >`, however, has an extra ability: to access items by index as well as by key. With a sorted list, you can go directly to the *n*th element in the sorting sequence (via the indexer on the `Keys`/`Values` properties). To do the same with a `SortedDictionary< , >`, you must manually enumerate over *n* items. (Alternatively, you could write a class that combines a sorted dictionary with a list class.)

None of the three collections allows duplicate keys (as is the case with all dictionaries).

The following example uses reflection to load all of the methods defined in `System.Object` into a sorted list keyed by name, and then enumerates their keys and values:

```
// MethodInfo is in the System.Reflection namespace

var sorted = new SortedList <string, MethodInfo>();

foreach (MethodInfo m in typeof (object).GetMethods())
    sorted [m.Name] = m;

foreach (string name in sorted.Keys)
    Console.WriteLine (name);

foreach (MethodInfo m in sorted.Values)
    Console.WriteLine (m.Name + " returns a " + m.ReturnType);
```

Here's the result of the first enumeration:

```
Equals
GetHashCode
GetType
ReferenceEquals
ToString
```

Here's the result of the second enumeration:

```
Equals returns a System.Boolean
GetHashCode returns a System.Int32
```

```
GetType returns a System.Type  
ReferenceEquals returns a System.Boolean  
ToString returns a System.String
```

Notice that we populated the dictionary through its indexer. If we instead used the `Add` method, it would throw an exception because the `object` class upon which we're reflecting overloads the `Equals` method, and you can't add the same key twice to a dictionary. By using the indexer, the later entry overwrites the earlier entry, preventing this error.

#### NOTE

You can store multiple members of the same key by making each value element a list:

```
SortedList <string, List<MethodInfo>>
```

Extending our example, the following retrieves the `MethodInfo` whose key is "GetHashCode", just as with an ordinary dictionary:

```
Console.WriteLine (sorted ["GetHashCode"]); // Int32 GetHashCode()
```

So far, everything we've done would also work with a `SortedDictionary<, >`. The following two lines, however, which retrieve the last key and value, work only with a sorted list:

```
Console.WriteLine (sorted.Keys [sorted.Count - 1]); //
```

```
ToString  
Console.WriteLine (sorted.Values[sorted.Count - 1].IsVirtual); // True
```

## Customizable Collections and Proxies

The collection classes discussed in previous sections are convenient in that you can directly instantiate them, but they don't allow you to control what happens when an item is added to or removed from the collection. With strongly typed collections in an application, you sometimes need this control; for instance:

- To fire an event when an item is added or removed
- To update properties because of the added or removed item
- To detect an “illegal” add/remove operation and throw an exception (for example, if the operation violates a business rule)

The .NET Framework provides collection classes for this exact purpose, in the `System.Collections.ObjectModel` namespace. These are essentially proxies or wrappers that implement `IList<T>` or `IDictionary<,>` by forwarding the methods through to an underlying collection. Each `Add`, `Remove`, or `Clear` operation is routed via a virtual method that acts as a “gateway” when overridden.

Customizable collection classes are commonly used for publicly exposed collections; for instance, a collection of controls exposed publicly on a `System.Windows.Form` class.

### `Collection<T>` and `CollectionBase`

The `Collection<T>` class is a customizable wrapper for `List<T>`.

As well as implementing `IList<T>` and `IList`, it defines four additional virtual methods and a protected property, as follows:

```
public class Collection<T> :  
    IList<T>, ICollection<T>, IEnumerable<T>, IList, ICollection,  
    IEnumerable  
{  
    // ...  
  
    protected virtual void ClearItems();  
    protected virtual void InsertItem (int index, T item);  
    protected virtual void RemoveItem (int index);  
    protected virtual void SetItem (int index, T item);  
  
    protected IList<T> Items { get; }  
}
```

The virtual methods provide the gateway by which you can “hook in” to change or enhance the list’s normal behavior. The protected `Items` property allows the implementer to directly access the “inner list”—this is used to make changes internally without the virtual methods firing.

The virtual methods need not be overridden; they can be left alone until there’s a requirement to alter the list’s default behavior. The following example demonstrates the typical “skeleton” use of `Collection<T>`:

```
public class Animal  
{
```

```

public string Name;
public int Popularity;

public Animal (string name, int popularity)
{
    Name = name; Popularity = popularity;
}
}

public class AnimalCollection : Collection <Animal>
{
    // AnimalCollection is already a fully functioning list of animals.
    // No extra code is required.
}

public class Zoo    // The class that will expose AnimalCollection.
{
    // This would typically have additional members.

    public readonly AnimalCollection Animals = new AnimalCollection();
}

class Program
{
    static void Main()
    {
        Zoo zoo = new Zoo();
        zoo.Animals.Add (new Animal ("Kangaroo", 10));
        zoo.Animals.Add (new Animal ("Mr Sea Lion", 20));
        foreach (Animal a in zoo.Animals) Console.WriteLine (a.Name);
    }
}

```

As it stands, `AnimalCollection` is no more functional than a simple `List<Animal>`; its role is to provide a base for future extension. To illustrate, let's now add a `Zoo` property to `Animal` so that it can reference the `Zoo` in which it lives and override each of the virtual

methods in `Collection<Animal>` to maintain that property automatically:

```
public class Animal
{
    public string Name;
    public int Popularity;
    public Zoo Zoo { get; internal set; }
    public Animal(string name, int popularity)
    {
        Name = name; Popularity = popularity;
    }
}

public class AnimalCollection : Collection <Animal>
{
    Zoo zoo;
    public AnimalCollection (Zoo zoo) { this.zoo = zoo; }

    protected override void InsertItem (int index, Animal item)
    {
        base.InsertItem (index, item);
        item.Zoo = zoo;
    }
    protected override void SetItem (int index, Animal item)
    {
        base.SetItem (index, item);
        item.Zoo = zoo;
    }
    protected override void RemoveItem (int index)
    {
        this [index].Zoo = null;
        base.RemoveItem (index);
    }
    protected override void ClearItems()
    {
        foreach (Animal a in this) a.Zoo = null;
    }
}
```

```
        base.ClearItems();
    }
}

public class Zoo
{
    public readonly AnimalCollection Animals;
    public Zoo() { Animals = new AnimalCollection (this); }
}
```

`Collection<T>` also has a constructor accepting an existing `IList<T>`. Unlike with other collection classes, the supplied list is *proxied* rather than *copied*, meaning that subsequent changes will be reflected in the wrapping `Collection<T>` (although *without* `Collection<T>`'s virtual methods firing). Conversely, changes made via the `Collection<T>` will change the underlying list.

## COLLECTIONBASE

`CollectionBase` is the nongeneric version of `Collection<T>` introduced in Framework 1.0. This provides most of the same features as `Collection<T>`, but is clumsier to use. Instead of the template methods `InsertItem`, `RemoveItem`, `SetItem`, and `ClearItem`, `CollectionBase` has “hook” methods that double the number of methods required: `OnInsert`, `OnInsertComplete`, `OnSet`, `OnSetComplete`, `OnRemove`, `OnRemoveComplete`, `OnClear`, and `OnClearComplete`. Because `CollectionBase` is nongeneric, you must also implement typed methods when subclassing it—at a minimum, a typed indexer and `Add` method.

## KeyedCollection<TKey,TItem> and DictionaryBase

`KeyedCollection< TKey , TItem >` subclasses `Collection< TItem >`. It both adds and subtracts functionality. What it adds is the ability to access items by key, much like with a dictionary. What it subtracts is the ability to proxy your own inner list.

A keyed collection has some resemblance to an `OrderedDictionary` in that it combines a linear list with a hashtable. However, unlike `OrderedDictionary`, it doesn't implement `IDictionary` and doesn't support the concept of a key/value *pair*. Keys are obtained instead from the items themselves, via the abstract `GetKeyForItem` method. This means enumerating a keyed collection is just like enumerating an ordinary list.

You can best think of `KeyedCollection< TKey , TItem >` as `Collection< TItem >` plus fast lookup by key.

Because it subclasses `Collection<>`, a keyed collection inherits all of `Collection<>`'s functionality, except for the ability to specify an existing list in construction. The additional members it defines are as follows:

```
public abstract class KeyedCollection < TKey , TItem > : Collection < TItem >

// ...

protected abstract TKey GetKeyForItem(TItem item);
protected void ChangeItemKey(TItem item, TKey newKey);

// Fast lookup by key - this is in addition to lookup by index.
public TItem this[TKey key] { get; }
```

```
    protected IDictionary<TKey, TItem> Dictionary { get; }  
}
```

`GetKeyForItem` is what the implementer overrides to obtain an item's key from the underlying object. The `ChangeItemKey` method must be called if the item's key property changes, in order to update the internal dictionary. The `Dictionary` property returns the internal dictionary used to implement the lookup, which is created when the first item is added. This behavior can be changed by specifying a creation threshold in the constructor, delaying the internal dictionary from being created until the threshold is reached (in the interim, a linear search is performed if an item is requested by key). A good reason not to specify a creation threshold is that having a valid dictionary can be useful in obtaining an `ICollection<>` of keys, via the `Dictionary`'s `Keys` property. This collection can then be passed on to a public property.

The most common use for `KeyedCollection<,>` is in providing a collection of items accessible both by index and by name. To demonstrate this, let's revisit the zoo, this time implementing `AnimalCollection` as a `KeyedCollection<string,Animal>`:

```
public class Animal  
{  
    string name;  
    public string Name  
    {  
        get { return name; }  
        set {  
            if (Zoo != null) Zoo.Animals.NotifyNameChange (this, value);  
            name = value;  
        }  
    }  
}
```

```

        }
    }

    public int Popularity;
    public Zoo Zoo { get; internal set; }

    public Animal (string name, int popularity)
    {
        Name = name; Popularity = popularity;
    }
}

public class AnimalCollection : KeyedCollection <string, Animal>
{
    Zoo zoo;
    public AnimalCollection (Zoo zoo) { this.zoo = zoo; }

    internal void NotifyNameChange (Animal a, string newName) =>
        this.ChangeItemKey (a, newName);

    protected override string GetKeyForItem (Animal item) => item.Name;

    // The following methods would be implemented as in the previous
    example
    protected override void InsertItem (int index, Animal item)... 
    protected override void SetItem (int index, Animal item)... 
    protected override void RemoveItem (int index)... 
    protected override void ClearItems()...
}

public class Zoo
{
    public readonly AnimalCollection Animals;
    public Zoo() { Animals = new AnimalCollection (this); }
}

```

The following code demonstrates its use:

```
Zoo zoo = new Zoo();
zoo.Animals.Add (new Animal ("Kangaroo", 10));
zoo.Animals.Add (new Animal ("Mr Sea Lion", 20));
Console.WriteLine (zoo.Animals [0].Popularity);           // 10
Console.WriteLine (zoo.Animals ["Mr Sea Lion"].Popularity); // 20
zoo.Animals ["Kangaroo"].Name = "Mr Roo";
Console.WriteLine (zoo.Animals ["Mr Roo"].Popularity);    // 10
```

## DICTIONARYBASE

The nongeneric version of `KeyedCollection` is called `DictionaryBase`. This legacy class takes a very different approach in that it implements `IDictionary` and uses clumsy hook methods like `CollectionBase: OnInsert, OnInsertComplete, OnSet, OnSetComplete, OnRemove, OnRemoveComplete, OnClear, and OnClearComplete` (and additionally, `OnGet`). The primary advantage of implementing `IDictionary` over taking the `KeyedCollection` approach is that you don't need to subclass it in order to obtain keys. But since the very purpose of `DictionaryBase` is to be subclassed, it's no advantage at all. The improved model in `KeyedCollection` is almost certainly due to the fact that it was written some years later, with the benefit of hindsight. `DictionaryBase` is best considered useful for backward compatibility.

## ReadOnlyCollection<T>

`ReadOnlyCollection<T>` is a wrapper, or *proxy*, that provides a read-only view of a collection. This is useful in allowing a class to publicly expose read-only access to a collection that the class can still update internally.

A read-only collection accepts the input collection in its constructor, to which it maintains a permanent reference. It doesn't take a static copy of the input collection, so subsequent changes to the input collection are visible through the read-only wrapper.

To illustrate, suppose that your class wants to provide read-only public access to a list of strings called **Names**. We could do this as follows:

```
public class Test
{
    List<string> names = new List<string>();
    public IReadonlyList<string> Names => names;
}
```

Although **Names** returns a read-only interface, the consumer can still downcast at runtime to **List<string>** or **IList<string>** and then call **Add**, **Remove**, or **Clear** on the list. **ReadOnlyCollection<T>** provides a more robust solution:

```
public class Test
{
    List<string> names = new List<string>();
    public ReadOnlyCollection<string> Names { get; private set; }

    public Test() => Names = new ReadOnlyCollection<string> (names);

    public void AddInternally() => names.Add ("test");
}
```

Now, only members within the **Test** class can alter the list of names:

```
Test t = new Test();

Console.WriteLine (t.Names.Count);          // 0
t.AddInternally();
Console.WriteLine (t.Names.Count);          // 1

t.Names.Add ("test");                     // Compiler error
((IList<string>) t.Names).Add ("test"); // NotSupportedException
```

## Immutable Collections

We just described how `ReadOnlyCollection<T>` creates a read-only view of a collection. Restricting the ability to write (*mutate*) a collection—or any other object—simplifies software and reduces bugs.

The *immutable collections* extend this principle, by providing collections that cannot be modified at all after initialization. Should you need to add an item to an immutable collection, you must instantiate a new collection, leaving the old one untouched.

Immutability is a hallmark of *functional programming* and has the following benefits:

- It eliminates a large class of bugs associated with changing state.
- It vastly simplifies parallelism and multithreading, by avoiding most of the thread-safety problems that we describe in Chapters 14, 22, and 23.
- It makes code easier to reason about.

The disadvantage of immutability is that when you need to make a change, you must create a whole new object. This incurs a performance hit, although there are mitigating strategies that we discuss in this section, including the ability to reuse portions of the original structure.

The immutable collections are built into .NET Core (in .NET Framework, they are available via the *System.Collections.Immutable* NuGet package). All collections are defined in the *System.Collections.Immutable* namespace:

Type	Internal structure
<code>ImmutableArray&lt;T&gt;</code>	Array
<code>ImmutableList&lt;T&gt;</code>	AVL tree
<code>ImmutableDictionary&lt;K,V&gt;</code>	AVL tree
<code>ImmutableHashSet&lt;T&gt;</code>	AVL tree
<code>ImmutableSortedDictionary&lt;K,V&gt;</code>	AVL tree
<code>ImmutableSortedSet&lt;T&gt;</code>	AVL tree
<code>ImmutableStack&lt;T&gt;</code>	Linked list
<code>ImmutableQueue&lt;T&gt;</code>	Linked list

The `ImmutableArray<T>` and `ImmutableList<T>` types are both immutable versions of `List<T>`. Both do the same job but with

different performance characteristics that we discuss in “[Immutable Collections and Performance](#)”.

The immutable collections expose a public interface similar to their mutable counterparts. The key difference is that the methods that appear to alter the collection (such as `Add` or `Remove`) don’t alter the original collection; instead they return a new collection with the requested item added or removed.

#### NOTE

Immutable collections prevent the adding and removing of items; they don’t prevent the items *themselves* from being mutated. To get the full benefits of immutability, you need to ensure that only immutable items end up in an immutable collection.

## Creating Immutable Collections

Each immutable collection type offers a `Create<T>()` method, which accepts optional initial values and returns an initialized immutable collection:

```
ImmutableArray<int> array = ImmutableArray.Create<int> (1, 2, 3);
```

Each collection also offers a `CreateRange<T>` method, which does the same job as `Create<T>`; the difference is that its parameter type is `IEnumerable<T>` instead of `params T[]`.

You can also create an immutable collection from an existing `IEnumerable<T>`, using appropriate extension methods (`ToImmutableArray`, `ToImmutableList`, `ToImmutableDictionary`, and so on):

```
var list = new[] { 1, 2, 3 }.ToImmutableList();
```

## Manipulating Immutable Collections

The `Add` method returns a new collection containing the existing elements plus the new one:

```
var oldList = ImmutableList.Create<int> (1, 2, 3);

ImmutableList<int> newList = oldList.Add (4);

Console.WriteLine (oldList.Count);      // 3  (unaltered)
Console.WriteLine (newList.Count);      // 4
```

The `Remove` method operates in the same fashion, returning a new collection with the item removed.

Repeatedly adding or removing elements in this manner is inefficient, because a new immutable collection is created for each add or remove operation. A better solution is to call `AddRange` (or `RemoveRange`), which accepts an `IEnumerable<T>` of items, which are all added or removed in one go:

```
var anotherList = oldList.AddRange (new[] { 4, 5, 6 });
```

The immutable list and array also define `Insert` and `InsertRange` methods to insert elements at a particular index, a `RemoveAt` method to remove at an index, and `RemoveAll`, which removes based on a predicate.

## Builders

For more complex initialization needs, each immutable collection class defines a *builder* counterpart. Builders are classes that are functionally equivalent to a mutable collection, with similar performance characteristics. After the data is initialized, calling `.ToImmutable()` on a builder returns an immutable collection:

```
ImmutableArray<int>.Builder builder = ImmutableArray.CreateBuilder<int>();
builder.Add (1);
builder.Add (2);
builder.Add (3);
builder.RemoveAt (0);
ImmutableArray<int> myImmutable = builder.ToImmutable();
```

You also can use builders to *batch* multiple updates to an existing immutable collection:

```
var builder2 = myImmutable.ToBuilder();
builder2.Add (4);      // Efficient
builder2.Remove (2);  // Efficient
...
// More changes to builder...
// Return a new immutable collection with all the changes applied:
ImmutableArray<int> myImmutable2 = builder2.ToImmutable();
```

## Immutable Collections and Performance

Most of the immutable collections use an *AVL tree* internally, which allows the add/remove operations to reuse portions of the original internal structure rather than having to re-create the entire thing from scratch. This reduces the overhead of add/remove operations from potentially *huge* (with large collections) to just *moderately large*, but it comes at the cost of making read operations slower. The end result is that most immutable collections are slower than their mutable counterparts for both reading and writing.

The most seriously affected is `ImmutableList<T>`, which for both read and add operations is 10 to 200 times slower than `List<T>` (depending on the size of the list). This is why `ImmutableArray<T>` exists: by using an array inside, it avoids the overhead for read operations (for which it's comparable in performance to an ordinary mutable array). The flip side is that it's *much* slower than (even) `ImmutableList<T>` for add operations because none of the original structure can be reused.

Hence, `ImmutableArray<T>` is desirable when you want unimpeded *read*-performance and don't expect many subsequent calls to `Add` or `Remove` (without using a builder):

Type	Read performance	Add performance
<code>ImmutableList&lt;T&gt;</code>	Slow	Slow
<code>ImmutableArray&lt;T&gt;</code>	Very fast	Very slow

## NOTE

Calling `Remove` on an `ImmutableArray` is more expensive than calling `Remove` on a `List<T>`—even in the worst-case scenario of removing the first element—because allocating the new collection places additional load on the garbage collector.

Although the immutable collections as a whole incur a potentially significant performance cost, it's important to keep the overall magnitude in perspective. An `Add` operation on an `ImmutableList` with a million elements is still likely to occur in less than a microsecond on a typical laptop, and a read operation, in less than 100 nanoseconds. And, if you need to perform write operations in a loop, you can avoid the accumulated cost with a builder.

The following factors also work to mitigate the costs:

- Immutability allows for easy concurrency and parallelization ([Chapter 23](#)), so you can employ all available cores. Parallelizing with mutable state easily leads to errors, and requires the use of locks or concurrent collections, both of which hurt performance.
- With immutability, you don't need to “defensively copy” collections or data structures to guard against unexpected change. This was a factor in favoring the use of immutable collections in writing recent portions of Visual Studio.
- In most typical programs, few collections have enough items for the difference to matter.

In addition to Visual Studio, the well-performing Microsoft Roslyn toolchain was built with immutable collections, demonstrating how the benefits can outweigh the costs.

## Plugging in Equality and Order

In the sections “Equality Comparison” and “Order Comparison” in Chapter 6, we described the standard .NET protocols that make a type equatable, hashable, and comparable. A type that implements these protocols can function correctly in a dictionary or sorted list “out of the box.” More specifically:

- A type for which `Equals` and `GetHashCode` return meaningful results can be used as a key in a `Dictionary` or `Hashtable`.
- A type that implements `IComparable`/`IComparable<T>` can be used as a key in any of the *sorted* dictionaries or lists.

A type’s default equating or comparison implementation typically reflects what is most “natural” for that type. Sometimes, however, the default behavior is not what you want. You might need a dictionary whose `string` type key is treated without respect to case. Or you might want a sorted list of customers, sorted by each customer’s postcode. For this reason, the .NET Framework also defines a matching set of “plug-in” protocols. The plug-in protocols achieve two things:

- They allow you to switch in alternative equating or comparison behavior.

- They allow you to use a dictionary or sorted collection with a key type that's not intrinsically equatable or comparable.

The plug-in protocols consist of the following interfaces:

#### **IEqualityComparer** and **IEqualityComparer<T>**

- Performs plug-in *equality comparison and hashing*
- Recognized by **Hashtable** and **Dictionary**

#### **IComparer** and **IComparer<T>**

- Performs plug-in *order comparison*
- Recognized by the sorted dictionaries and collections; also, **Array.Sort**

Each interface comes in both generic and nongeneric forms. The **IEqualityComparer** interfaces also have a default implementation in a class called **EqualityComparer**.

In addition, in Framework 4.0 we got two new interfaces called **IStructuralEquatable** and **IStructuralComparable** which allow for the option of structural comparisons on classes and arrays.

## **IEqualityComparer and EqualityComparer**

An equality comparer switches in nondefault equality and hashing behavior, primarily for the **Dictionary** and **Hashtable** classes.

Recall the requirements of a hashtable-based dictionary. It needs answers to two questions for any given key:

- Is it the same as another?
- What is its integer hashcode?

An equality comparer answers these questions by implementing the `IEqualityComparer` interfaces:

```
public interface IEqualityComparer<T>
{
    bool Equals (T x, T y);
    int GetHashCode (T obj);
}

public interface IEqualityComparer      // Nongeneric version
{
    bool Equals (object x, object y);
    int GetHashCode (object obj);
}
```

To write a custom comparer, you implement one or both of these interfaces (implementing both gives maximum interoperability). Because this is somewhat tedious, an alternative is to subclass the abstract `EqualityComparer` class, defined as follows:

```
public abstract class EqualityComparer<T> : IEqualityComparer,
                                              IEqualityComparer<T>
{
    public abstract bool Equals (T x, T y);
    public abstract int GetHashCode (T obj);
```

```
    bool IEqualityComparer.Equals (object x, object y);  
    int IEqualityComparer.GetHashCode (object obj);  
  
    public static EqualityComparer<T> Default { get; }  
}
```

`EqualityComparer` implements both interfaces; your job is simply to override the two abstract methods.

The semantics for `Equals` and `GetHashCode` follow the same rules as those for `object.Equals` and `object.GetHashCode`, described in Chapter 6. In the following example, we define a `Customer` class with two fields, and then write an equality comparer that matches both the first and last names:

```
public class Customer  
{  
    public string LastName;  
    public string FirstName;  
  
    public Customer (string last, string first)  
    {  
        LastName = last;  
        FirstName = first;  
    }  
}  
public class LastFirstEqComparer : EqualityComparer <Customer>  
{  
    public override bool Equals (Customer x, Customer y)  
        => x.LastName == y.LastName && x.FirstName == y.FirstName;  
  
    public override int GetHashCode (Customer obj)  
        => (obj.LastName + ";" + obj.FirstName).GetHashCode();  
}
```

To illustrate how this works, let's create two customers:

```
Customer c1 = new Customer ("Bloggs", "Joe");
Customer c2 = new Customer ("Bloggs", "Joe");
```

Because we've not overridden `object.Equals`, normal reference-type equality semantics apply:

```
Console.WriteLine (c1 == c2);           // False
Console.WriteLine (c1.Equals (c2));      // False
```

The same default equality semantics apply when using these customers in a `Dictionary` without specifying an equality comparer:

```
var d = new Dictionary<Customer, string>();
d [c1] = "Joe";
Console.WriteLine (d.ContainsKey (c2));    // False
```

Now, with the custom equality comparer:

```
var eqComparer = new LastFirstEqComparer();
var d = new Dictionary<Customer, string> (eqComparer);
d [c1] = "Joe";
Console.WriteLine (d.ContainsKey (c2));    // True
```

In this example, we would have to be careful not to change the customer's `FirstName` or `LastName` while it was in use in the dictionary; otherwise, its hashcode would change and the `Dictionary` would break.

## EQUALITYCOMPARER<T>.DEFAULT

Calling `EqualityComparer<T>.Default` returns a general-purpose equality comparer that you can use as an alternative to the static `object.Equals` method. The advantage is that it first checks whether `T` implements `IEquatable<T>`, and if so, calls that implementation instead, avoiding the boxing overhead. This is particularly useful in generic methods:

```
static bool Foo<T> (T x, T y)
{
    bool same = EqualityComparer<T>.Default.Equals (x, y);
    ...
}
```

## IComparer and Comparer

Comparers are used to switch in custom ordering logic for sorted dictionaries and collections.

Note that a comparer is useless to the unsorted dictionaries such as `Dictionary` and `Hashtable`—these require an `IEqualityComparer` to get hashcodes. Similarly, an equality comparer is useless for sorted dictionaries and collections.

Here are the `IComparer` interface definitions:

```
public interface IComparer
{
    int Compare(object x, object y);
}
public interface IComparer <in T>
```

```
{  
    int Compare(T x, T y);  
}
```

As with equality comparers, there's an abstract class that you can subtype instead of implementing the interfaces:

```
public abstract class Comparer<T> : IComparer, IComparer<T>  
{  
    public static Comparer<T> Default { get; }  
  
    public abstract int Compare (T x, T y); // Implemented  
    by you  
    int IComparer.Compare (object x, object y); // Implemented for you  
}
```

The following example illustrates a class that describes a wish as well as a comparer that sorts wishes by priority:

```
class Wish  
{  
    public string Name;  
    public int Priority;  
  
    public Wish (string name, int priority)  
    {  
        Name = name;  
        Priority = priority;  
    }  
}  
  
class PriorityComparer : Comparer <Wish>  
{  
    public override int Compare (Wish x, Wish y)  
    {
```

```

        if (object.Equals (x, y)) return 0;           // Fail-safe check
        return x.Priority.CompareTo (y.Priority);
    }
}

```

The `object.Equals` check ensures that we can never contradict the `Equals` method. Calling the static `object.Equals` method in this case is better than calling `x.Equals` because it still works if `x` is null!

Here's how our `PriorityComparer` is used to sort a `List`:

```

var wishList = new List<Wish>();
wishList.Add (new Wish ("Peace", 2));
wishList.Add (new Wish ("Wealth", 3));
wishList.Add (new Wish ("Love", 2));
wishList.Add (new Wish ("3 more wishes", 1));

wishList.Sort (new PriorityComparer());
foreach (Wish w in wishList) Console.Write (w.Name + " | ");

// OUTPUT: 3 more wishes | Love | Peace | Wealth |

```

In the next example, `SurnameComparer` allows you to sort surname strings in an order suitable for a phone book listing:

```

class SurnameComparer : Comparer <string>
{
    string Normalize (string s)
    {
        s = s.Trim().ToUpper();
        if (s.StartsWith ("MC")) s = "MAC" + s.Substring (2);
        return s;
    }
}

```

```
    public override int Compare (string x, string y)
        => Normalize (x).CompareTo (Normalize (y));
}
```

Here's **SurnameComparer** in use in a sorted dictionary:

```
var dic = new SortedDictionary<string,string> (new SurnameComparer());
dic.Add ("MacPhail", "second!");
dic.Add ("MacWilliam", "third!");
dic.Add ("McDonald", "first!");

foreach (string s in dic.Values)
    Console.Write (s + " ");           // first! second! third!
```

## StringComparer

**StringComparer** is a predefined plug-in class for equating and comparing strings, allowing you to specify language and case sensitivity. **StringComparer** implements both **IEqualityComparer** and **IComparer** (and their generic versions), so you can use it with any type of dictionary or sorted collection.

Because **StringComparer** is abstract, you obtain instances via its static properties. **StringComparer.Ordinal** mirrors the default behavior for string equality comparison and **StringComparer.CurrentCulture** for order comparison. Here are all of its static members:

```
public static StringComparer CurrentCulture { get; }
public static StringComparer CurrentCultureIgnoreCase { get; }
public static StringComparer InvariantCulture { get; }
public static StringComparer InvariantCultureIgnoreCase { get; }
```

```
public static StringComparer Ordinal { get; }
public static StringComparer OrdinalIgnoreCase { get; }
public static StringComparer Create (CultureInfo culture,
                                    bool ignoreCase);
```

In the following example, an ordinal case-insensitive dictionary is created such that `dict["Joe"]` and `dict["JOE"]` mean the same thing:

```
var dict = new Dictionary<string, int>
(StringComparer.OrdinalIgnoreCase);
```

In the next example, an array of names is sorted, using Australian English:

```
string[] names = { "Tom", "HARRY", "sheila" };
CultureInfo ci = new CultureInfo ("en-AU");
Array.Sort<string> (names, StringComparer.Create (ci, false));
```

The final example is a culture-aware version of the `SurnameComparer` we wrote in the previous section (to compare names suitable for a phone book listing):

```
class SurnameComparer : Comparer<string>
{
    StringComparer strCmp;

    public SurnameComparer (CultureInfo ci)
    {
        // Create a case-sensitive, culture-sensitive string comparer
        strCmp = StringComparer.Create (ci, false);
    }
}
```

```

string Normalize (string s)
{
    s = s.Trim();
    if (s.ToUpper().StartsWith ("MC")) s = "MAC" + s.Substring (2);
    return s;
}

public override int Compare (string x, string y)
{
    // Directly call Compare on our culture-aware StringComparer
    return strCmp.Compare (Normalize (x), Normalize (y));
}
}

```

## IStructuralEquatable and IStructuralComparable

As we discussed in [Chapter 6](#), structs implement *structural comparison* by default: two structs are equal if all of their fields are equal. Sometimes, however, structural equality and order comparison are useful as plug-in options on other types, as well—such as arrays. Framework 4.0 introduced two new interfaces to help with this:

```

public interface IStructuralEquatable
{
    bool Equals (object other, IEqualityComparer comparer);
    int GetHashCode (IEqualityComparer comparer);
}

public interface IStructuralComparable
{
    int CompareTo (object other, IComparer comparer);
}

```

The `IEqualityComparer`/`IComparer` that you pass in are applied to each individual element in the composite object. We can demonstrate this by using arrays. In the following example, we compare two arrays for equality, first using the default `Equals` method, then using `IStructuralEquatable`'s version:

```
int[] a1 = { 1, 2, 3 };
int[] a2 = { 1, 2, 3 };
IStructuralEquatable se1 = a1;
Console.WriteLine (a1.Equals (a2)); // False
Console.WriteLine (se1.Equals (a2, EqualityComparer<int>.Default)); // True
```

Here's another example:

```
string[] a1 = "the quick brown fox".Split();
string[] a2 = "THE QUICK BROWN FOX".Split();
IStructuralEquatable se1 = a1;
bool isTrue = se1.Equals (a2,
    StringComparer.InvariantCultureIgnoreCase);
```

---

<sup>1</sup> There's also a functionally identical nongeneric version of this called `SortedList`.