

Chapter 2. C# Language Basics

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

NOTE

Almost all of the code listings in this book 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 the samples, in LINQPad, click the Samples tab, and then click “Download more samples.” LINQPad is free—go to <http://www.linqpad.net>.

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*:

```
int x = 12 * 30;           // Statement 1
System.Console.WriteLine (x); // Statement 2
```

Our program consists of two *statements*. Statements in C# execute sequentially and are terminated by a semicolon. The first statement computes the *expression* `12 * 30` and stores the result in a *variable*, named `x`, whose type is a 32-bit integer (`int`). The second statement calls the `WriteLine` *method* on a *class* called `Console`, which is defined in a *namespace* called `System`. This prints the variable `x` to a text window on the screen.

A method performs a function; 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. A class is a kind of *type*, which we examine in “**Type Basics**”.

At the outermost level, types are organized into *namespaces*. Many commonly used types—including the `Console` class—reside in the `System` namespace. The .NET libraries are organized into nested namespaces. For example, the `System.Text` namespace contains types for handling text, and `System.IO` contain types for input/output.

Qualifying the `Console` class with the `System` namespace on every use adds clutter. The `using` directive lets you avoid this clutter by *importing* a namespace:

```
using System;           // Import the System namespace

int x = 12 * 30;
Console.WriteLine (x);  // No need to specify System.
```

A basic form of code reuse is to write higher-level functions that call lower-level functions. We can *refactor* our program with a reusable *method* called `FeetToInches` that multiplies an integer by 12, as follows:

```
using System;

Console.WriteLine (FeetToInches (30));    // 360
Console.WriteLine (FeetToInches (100));   // 1200

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

Our method contains a series of statements surrounded by a pair of braces. This is called a *statement block*.

A method can receive *input* data from the caller by specifying *parameters* and *output* data back to the caller by specifying a *return type*. Our

FeetToInches method has a parameter for inputting feet, and a return type for outputting inches:

```
int FeetToInches (int feet)
...
```

The *literals* 30 and 100 are the *arguments* passed to the FeetToInches method.

If a method doesn't receive input, use empty parentheses. If it doesn't return anything, use the `void` keyword:

```
using System;
SayHello();

void SayHello()
{
    Console.WriteLine ("Hello, world");
}
```

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*.

Compilation

The C# compiler compiles source code (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 an *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 itself is a set of libraries (as well as a runtime environment).

Each of the programs in the preceding section began directly with a series of statements (called *top-level statements*). The presence of top-level statements implicitly creates an entry point for a console or Windows

application. (Without top-level statements, a *Main method* denotes an application's entry point—see “Custom Types”.)

NOTE

Unlike .NET Framework, .NET 8 assemblies never have a *.exe* extension. The *.exe* that you see after building a .NET 8 application is a platform-specific native loader responsible for starting your application's *.dll* assembly.

.NET 8 also allows you to create a self-contained deployment that includes the loader, your assemblies, and the required portions of the .NET runtime—all in a single *.exe* file. .NET 8 also allows ahead-of-time (AOT) compilation, where the executable contains precompiled native code for faster startup and reduced memory consumption.

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 8 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* that prints “Hello world.”

To build and run your program, run the following 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 17](#).

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;

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



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:

```
int using = 123;    // Illegal
int @using = 123;   // Legal
```

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

Contextual keywords

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

add	dynamic	into	nuint	set
alias	equals	join	on	unman
and	file	let	or	value
ascending	from	managed	orderby	var
async	get	nameof	partial	with
await	global	nint	remove	when
by	group	not	required	where
descending	init	notnull	select	yield

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. An example is the semicolon, which terminates a statement. 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”](#).

Type Basics

A *type* defines the blueprint for a value. In this example, we use two literals of type `int` with values 12 and 30. We also declare a *variable* of type `int` whose name is `x`:

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

NOTE

Because most of the code listings in this book require types from the `System` namespace, we will omit `using System` from now on, unless we're illustrating a concept relating to namespaces.

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. You 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>. You 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 = 2022;  
message = message + x.ToString();  
Console.WriteLine (message);                 // Hello world2022
```

In this example, we called `x.ToString()` to obtain a string representation of the integer `x`. You can call `ToString()` on a variable of almost any type.

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 contains many important types that are not predefined by C# (e.g., DateTime).

Custom Types

Just as we can write our own methods, we can write our own types. In this next example, we define a custom type named `UnitConverter`—a class that serves as a blueprint for unit conversions:

```

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

public class UnitConverter
{
    int ratio;                                // Field

    public UnitConverter (int unitRatio)    // Constructor
    {
        ratio = unitRatio;
    }

    public int Convert (int unit)          // Method
    {
        return unit * ratio;
    }
}

```

```
}  
}
```

NOTE

In this example, our class definition appears in the same file as our top-level statements. This is legal—as long as the top-level statements appear first—and is acceptable when writing small test programs. With larger programs, the standard approach is to put the class definition in a separate file such as *UnitConverter.cs*.

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 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 can be marked as `static`. To refer to a static member from outside its type, you specify its *type* name rather than an *instance*. An example is the `WriteLine` method of the `Console` class. Because this is static, we call `Console.WriteLine()` and not `new Console().WriteLine()`.

(The `Console` class is actually declared as a *static class*, which means that *all* of its members are static and you can never create instances of a `Console`.)

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. We create two instances of the `Panda`, print their names, and then print the total population:

```
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

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
}
}

```

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 could not be accessed from outside the class. 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.

Defining namespaces

Particularly with larger programs, it makes sense to organize types into namespaces. Here’s how to define the `Panda` class inside a namespace called `Animals`:

```

using System;
using Animals;

Panda p = new Panda ("Pan Dee");
Console.WriteLine (p.Name);

namespace Animals
{
    public class Panda
    {
        ...
    }
}

```

```
}  
}
```

In this example, we also *imported* the `Animals` namespace so that our top-level statements could access its types without qualification. Without that import, we'd need to do this:

```
Animals.Panda p = new Animals.Panda ("Pan Dee");
```

We cover namespaces in detail at the end of this chapter (see [“Namespaces”](#)).

Defining a Main method

All of our examples, so far, have used top-level statements (a feature introduced in C# 9).

Without top-level statements, a simple console or Windows application looks like this:

```
using System;  
  
class Program  
{  
    static void Main()    // Program entry point  
    {  
        int x = 12 * 30;  
        Console.WriteLine (x);  
    }  
}
```

In the absence of top-level statements, C# looks for a static method called `Main`, which becomes the entry point. The `Main` method can be defined inside any class (and only one `Main` method can exist).

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](#).)

TOP-LEVEL STATEMENTS

Top-level statements (introduced in C# 9) let you avoid the baggage of a static `Main` method and a containing class. A file with top-level statements comprises three parts, in this order:

1. (Optionally) using directives
2. A series of statements, optionally mixed with method declarations
3. (Optionally) Type and namespace declarations

For example:

```
using System;                                // Part 1

Console.WriteLine ("Hello, world");          // Part 2
void SomeMethod1() { ... }                   // Part 2
Console.WriteLine ("Hello again!");          // Part 2
void SomeMethod2() { ... }                   // Part 2

class SomeClass { ... }                      // Part 3
namespace SomeNamespace { ... }              // Part 3
```

Because the CLR doesn't explicitly support top-level statements, the compiler translates your code into something like this:

```
using System;                                // Part 1

static class Program$ // Special compiler-generated name
{
    static void Main$ (string[] args) // Compiler-generated name
    {
        Console.WriteLine ("Hello, world");    // Part 2
        void SomeMethod1() { ... }              // Part 2
        Console.WriteLine ("Hello again!");    // Part 2
        void SomeMethod2() { ... }              // Part 2
    }
}

class SomeClass { ... }                      // Part 3
namespace SomeNamespace { ... }              // Part 3
```


Notice that everything in Part 2 is wrapped inside the main method. This means that `SomeMethod1` and `SomeMethod2` act as *local methods*. We discuss the full implications in “**Local methods**”, the most important being that local methods (unless declared as `static`) can access variables declared within the containing method:

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

void LocalMethod() { Console.WriteLine (x); }    // We can access x
```

Another consequence is that top-level methods cannot be accessed from other classes or types.

Top-level statements can optionally return an integer value to the caller and access a “magic” variable of type `string[]` called `args`, corresponding to command-line arguments passed by the caller.

As a program can have only one entry point, there can be at most one file with top-level statements in a C# project.

Types and 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 *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.¹

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](#)”.)

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](#)”). 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](#)” and pointer types in “[Unsafe Code and Pointers](#)”.

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

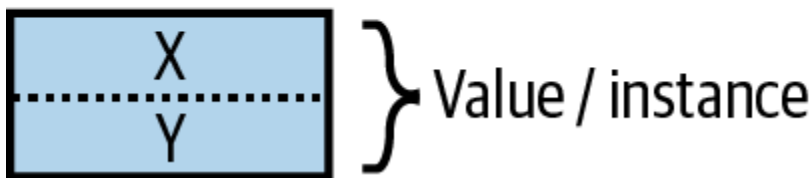


Figure 2-1. A value-type instance in memory

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

```
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.

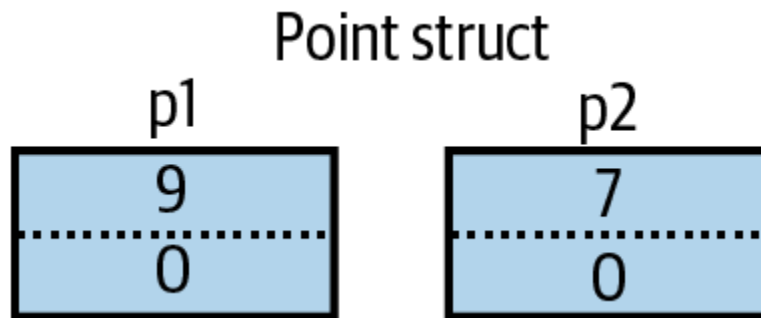


Figure 2-2. Assignment copies a value-type instance

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; }

```

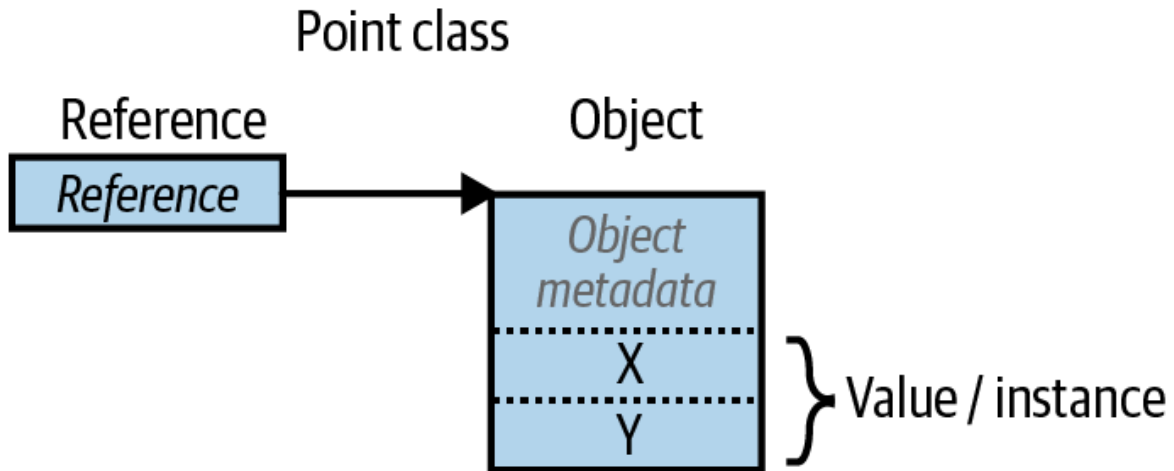


Figure 2-3. A reference-type instance in memory

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`:

```
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.

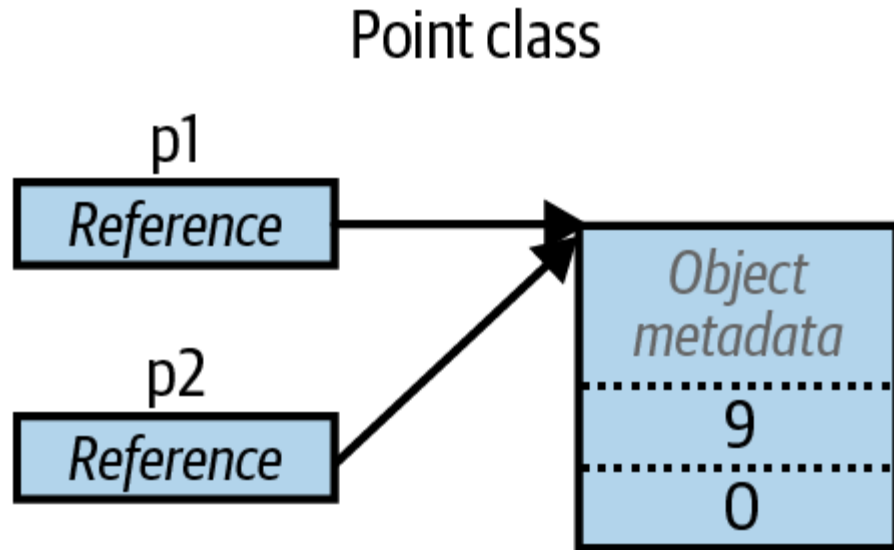


Figure 2-4. Assignment copies a reference

Null

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

```
Point p = null;
Console.WriteLine (p == null);    // True

// The following line generates a runtime error
// (a NullReferenceException is thrown):
Console.WriteLine (p.X);

class Point {...}
```

NOTE

In “**Nullable Reference Types**”, we describe a feature of C# that helps to reduce accidental `NullReferenceException` errors.

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

```
Point p = null;    // Compile-time error
int x = null;      // Compile-time error
```

```
struct Point {...}
```

NOTE

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

Storage overhead

Value-type instances occupy precisely the memory required to store their fields. In this example, `Point` takes 8 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 8 bytes). Thus, the following actually consumes 16 bytes of memory (with the 7 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”](#)).

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 8 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 4 or 8 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 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:


```
int i = 7;           // Underlying hexadecimal representation
bool b = true;       // 0x7
char c = 'A';        // 0x1
float f = 0.5f;       // 0x41
// uses IEEE floating-point encoding
```

The `System.IntPtr` and `System.UIntPtr` types are also primitive (see [Chapter 24](#)).

Numeric Types

C# has the predefined numeric types shown in [Table 2-1](#).

Table 2-1. Predefined numeric types in 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 $2^{63} - 1$
nint	IntPtr		32/64 bits	
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 $2^{64} - 1$
nuint	UIntPtr		32/64 bits	
Real				
float	Single	F	32 bits	$\pm (\sim 10^{38})$
double	Double	D	64 bits	$\pm (\sim 10^{308})$

C# type	System type	Suffix	Size	Range
decimal	Decimal	M	128 bits	$\pm (\sim 10^{28})$

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. The `nint` and `nuint` native-sized integer types are most useful when working with pointers, so we will describe these in a later chapter (see “[Native-Sized Integers](#)”).

Of the *real* number types, `float` and `double` are called *floating-point types*² 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.

NOTE

.NET supplements this list with several specialized numeric types, including `Int128` and `UInt128` for signed and unsigned 128-bit integers, `BigInteger` for arbitrarily large integers, and `Half` for 16-bit floating point numbers. `Half` is intended mainly for interoperating with graphics card processors and does not have native support in most CPUs, making `float` and `double` better choices for general use.

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;
```

You can insert an underscore anywhere within a numeric literal to make it more readable:

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

You can 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 `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 eliminate the remainder (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
~	Complement	~0xfU	0xffffffff0U
&	And	0xf0 & 0x33	0x30
	Or	0xf0 0x33	0xf3
^	Exclusive Or	0xff00 ^ 0x0ff0	0xf0f0
<<	Shift left	0x20 << 2	0x80
>>	Shift right	0x20 >> 1	0x10
>>>	Unsigned shift right	int.MinValue >>> 1	0x40000000

The shift-right operator (`>>`) replicates the high-order bit when operating on signed integers, whereas the unsigned shift-right operator (`>>>`) does not.

NOTE

Additional bitwise operations are exposed via a class called `BitOperations` in the `System.Numerics` namespace (see “[BitOperations](#)”).

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, you 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 at <http://www.ieee.org>.

double Versus decimal

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

Category	<code>double</code>	<code>decimal</code>
Internal representation	Base 2	Base 10
Decimal precision	15–16 significant figures	28–29 significant figures
Range	$\pm(\sim 10^{-324}$ to $\sim 10^{308})$	$\pm(\sim 10^{-28}$ to $\sim 10^{28})$
Special values	+0, -0, + ∞ , - ∞ , 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 x = 0.1f; // Not quite 0.1
Console.WriteLine (x + x + x + x + x + x + x + x + x + x); // 1.0000001
```

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; // 0.16666666666666666666666666667M
double d = 1.0 / 6.0; // 0.16666666666666666
```

This leads to accumulated rounding errors:

```
decimal notQuiteWholeM = m+m+m+m+m+m; // 1.000000000000000000000000000002M
double notQuiteWholeD = d+d+d+d+d+d; // 0.99999999999999998
```

which break 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.³ 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](#)):

```
Dude d1 = new Dude ("John");
Dude d2 = new Dude ("John");
Console.WriteLine (d1 == d2);         // False
Dude d3 = d1;
Console.WriteLine (d1 == d3);         // True

public class Dude
{
    public string Name;
    public Dude (string n) { Name = n; }
}
```

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”.

We explain the equality and comparison operators in greater detail in “Operator Overloading”, and in “Equality Comparison” and “Order Comparison”.

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; otherwise `b` is evaluated:

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

The conditional operator is particularly useful in Language-Integrated Query (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.

Table 2-2. Escape sequence characters

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	New line	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>";
```

Raw string literals (C# 11)

Wrapping a string in three or more quote characters (""") creates a *raw string literal*. Raw string literals can contain almost any character sequence, without escaping or doubling up:

```
string raw = """<file path="c:\temp\test.txt"></file>""";
```

Raw string literals make it easy to represent JSON, XML, and HTML literals, as well as regular expressions and source code. Should you need to include three (or more) quote characters in the string itself, you can do so by wrapping the string in four (or more) quote characters:

```
string raw = """"The "" sequence denotes raw string literals."""";
```

Multiline raw string literals are subject to special rules. We can represent the string "Line 1\r\nLine 2" as follows:

```
string multiLineRaw = """  
Line 1  
Line 2  
""";
```

Notice that the opening and closing quotes must be on separate lines to the string content. Additionally:

- Whitespace following the *opening* `"""` (on the same line) is ignored.
- Whitespace preceding the *closing* `"""` (on the same line) is treated as *common indentation* and is removed from every line in the string. This lets you include indentation for source-code readability without that indentation becoming part of the string.

Here's another example to illustrate the multiline raw string literal rules:

```
if (true)
    Console.WriteLine ("""
        {
            "Name" : "Joe"
        }
        """);
```

The output is as follows:

```
{
    "Name" : "Joe"
}
```

The compiler will generate an error if each line in a multiline raw string literal is not prefixed with the common indentation specified by the closing quotes.

Raw string literals can be interpolated, subject to special rules described in [“String interpolation”](#).

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”](#)):

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

Should you need to use a colon for another purpose (such as a ternary conditional operator, which we’ll cover later), you must wrap the entire expression in parentheses:

```
bool b = true;
Console.WriteLine($"The answer in binary is {(b ? 1 : 0)}");
```

From C# 10, interpolated strings can be constants, as long as the interpolated values are constants:

```
const string greeting = "Hello";  
const string message = $"{greeting}, world";
```

From C# 11, interpolated strings are permitted to span multiple lines (whether standard or verbatim):

```
string s = $"this interpolation spans {1 +  
1} lines";
```

Raw string literals (from C# 11) can also be interpolated:

```
string s = $"""The date and time is {DateTime.Now}""";
```

To include a brace literal in an interpolated string:

- With standard and verbatim string literals, repeat the desired brace character.
- With raw string literals, change the interpolation sequence by repeating the \$ prefix.

Using two (or more) \$ characters in a raw string literal prefix changes the interpolation sequence from one brace to two (or more) braces:

```
Console.WriteLine ($$"""{ "TimeStamp": "{ {DateTime.Now} }" }""");  
// Output: { "TimeStamp": "01/01/2024 12:13:25 PM" }
```

This preserves the ability to copy-and-paste text into a raw string literal without needing to modify the string.

String comparisons

To perform *equality* comparisons with strings, you can use the == operator (or one of string's Equals methods). For *order* comparison, you must use the string's CompareTo method; the < and > operators are unsupported. We describe equality and order comparison in detail in “[Comparing Strings](#)”.

UTF-8 Strings

From C# 11, you can use the `u8` suffix to create string literals encoded in UTF-8 rather than UTF-16. This feature is intended for advanced scenarios such as the low-level handling of JSON text in performance hotspots:

```
ReadOnlySpan<byte> utf8 = "ab→cd"u8; // Arrow symbol consumes 3 bytes
Console.WriteLine (utf8.Length);      // 7
```

The underlying type is `ReadOnlySpan<byte>`, which we cover in [Chapter 23](#). You can convert this to an array by calling the `ToArray()` method.

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. You 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' };
```

NOTE

From C# 12, you can use square brackets instead of curly braces:

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

This is called a *collection expression* and has the advantage of also working when calling methods:

```
Foo ([ 'a', 'e', 'i', 'o', 'u' ]);  
  
void Foo (char[] letters) { ... }
```

Collection expressions also work with other collection types such as lists and sets—see [“Collection Initializers and Collection Expressions”](#).

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”](#).

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:

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

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

```
Point[] a = new Point[1000];  
int x = a[500].X;                // Runtime error, NullReferenceException  
  
public class Point { public int X, Y; }
```

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

Indices and ranges (introduced in C# 8) 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 [Chapter 23](#)).

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

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}
};
```

(From C# 12, you can use square brackets instead of braces with single-dimensional arrays.)

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

```
var i = 3;           // i is implicitly of type int
var s = "sausage";   // s is implicitly of type string
```

The same principle can be applied to arrays, except that it can be taken one stage further. By omitting the type qualifier after the `new` keyword, the compiler infers the array type:

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

Here's how we can apply this to multidimensional arrays:

```
var rectMatrix = new[,]          // 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[] {0,1,2},
    new[] {3,4,5},
    new[] {6,7,8,9}
};
```

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[] {1,1000000000000L};    // 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](#)”).

Variables and Parameters

A variable represents a storage location that has a modifiable value. A variable can be a *local variable*, *parameter* (*value*, *ref*, or *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;

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 process ends.

Definite Assignment

C# enforces a definite assignment policy. In practice, this means that outside of an `unsafe` or `interop` context, you can't accidentally 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:

```
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:

```
int[] ints = new int[2];  
Console.WriteLine (ints[0]);    // 0
```

The following code outputs `0`, because fields are implicitly assigned a default value (whether instance or static):

```
Console.WriteLine (Test.X);    // 0

class Test { public static int X; }    // field
```

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
Reference types (and nullable value types)	null
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
```

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`:

```
Foo (8); // 8 is an argument
static void Foo (int p) {...} // p is a parameter
```

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:

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

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

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

```
StringBuilder sb = new StringBuilder();
Foo (sb);
Console.WriteLine (sb.ToString());    // test

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

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:

```
int x = 8;
Foo (ref x);           // Ask Foo to deal directly with x
Console.WriteLine (x);  // x is now 9

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

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.⁴ This makes it very clear what's going on.

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

```
string x = "Penn";
string y = "Teller";
Swap (ref x, ref y);
Console.WriteLine (x);    // Teller
Console.WriteLine (y);    // Penn

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

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:

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

void Split (string name, out string firstNames, out string lastName)
{
    int i = name.LastIndexOf (' ');
```

```

    firstNames = name.Substring (0, i);
    lastName = name.Substring (i + 1);
}

```

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

Out variables and discards

You can declare variables on the fly when calling methods with out parameters. We can replace the first two lines in our preceding example with this:

```

Split ("Stevie Ray Vaughan", out string a, out string b);

```

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 uninterested by using an underscore:

```

Split ("Stevie Ray Vaughan", out string a, out _); // Discard 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**”).

The `params` modifier

The `params` modifier, if applied to the last parameter of a method, allows the method to accept any number of arguments of a particular type. The parameter type must be declared as a (single-dimensional) array, as shown in the following example:

```
int total = Sum (1, 2, 3, 4);
Console.WriteLine (total);           // 10

// The call to Sum above is equivalent to:
int total2 = Sum (new int[] { 1, 2, 3, 4 });

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;
}
```

If there are zero arguments in the `params` position, a zero-length array is created.

You can also supply a `params` argument as an ordinary array. The first line in our example 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.

WARNING

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, a parameterless constructor of a value type, or a default expression. 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:

```
Foo (1);    // 1, 0
```

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

You can do the converse (pass a default value to x and an explicit value to y) by combining optional parameters with *named arguments*.

Named arguments

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

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

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

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, you 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) { ... }
```

You 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 24](#).

Ref Locals

A somewhat esoteric feature of C# is that you can define a local variable that *references* an element in an array or field in an object (from C# 7):

```
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 23](#) use `ref` returns to implement a highly efficient indexer. Outside such scenarios, `ref` returns are not commonly used, and you can consider them a micro-optimization feature.

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

```
class Program
{
    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 you 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**”), 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` 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 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**”, we will describe a scenario in which the use of var is mandatory.

Target-Typed new Expressions

Another way to reduce lexical repetition is with *target-typed new expressions* (from C# 9):

```
System.Text.StringBuilder sb1 = new();  
System.Text.StringBuilder sb2 = new ("Test");
```

This is precisely equivalent to:

```
System.Text.StringBuilder sb1 = new System.Text.StringBuilder();  
System.Text.StringBuilder sb2 = new System.Text.StringBuilder ("Test");
```

The principle is that you can call new without specifying a type name if the compiler is able to unambiguously infer it. Target-typed new expressions are particularly useful when the variable declaration and initialization are in different parts of your code. A common example is when you want to initialize a field in a constructor:

```
class Foo
{
    System.Text.StringBuilder sb;

    public Foo (string initialValue)
    {
        sb = new (initialValue);
    }
}
```

Target-typed new expressions are also helpful in the following scenario:

```
MyMethod (new ("test"));

void MyMethod (System.Text.StringBuilder sb) { ... }
```

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**”.

Table 2-3. C# operators (categories in order of precedence)

Category	Operator symbol	Operator name	Example	Useful over
Primary	.	Member access	x.y	No
	? . and ?[]	Null-conditional	x?.y or x?[0]	No
	! (postfix)	Null-forgiving	x!.y or x![0]	No
	-> (unsafe)	Pointer to struct	x->y	No
	()	Function call	x()	No
	[]	Array/index	a[x]	Via i
	++	Post-increment	x++	Yes
	--	Post-decrement	x--	Yes
	new	Create instance	new Foo()	No
	stackalloc	Stack allocation	stackalloc(10)	No
	typeof	Get type from identifier	typeof(int)	No

Category	Operator symbol	Operator name	Example	User over
	nameof	Get name of identifier	nameof(x)	No
	checked	Integral overflow check on	checked(x)	No
	unchecked	Integral overflow check off	unchecked(x)	No
	default	Default value	default(char)	No
Unary	await	Await	await myTask	No
	sizeof	Get size of struct	sizeof(int)	No
	+	Positive value of	+x	Yes
	-	Negative value of	-x	Yes
	!	Not	!x	Yes
	~	Bitwise complement	~x	Yes
	++	Pre-increment	++x	Yes
	--	Pre-decrement	--x	Yes

Category	Operator symbol	Operator name	Example	User over
	()	Cast	(int)x	No
	^	Index from end	array[^1]	No
	* (unsafe)	Value at address	*x	No
	& (unsafe)	Address of value	&x	No
Range	..	Range of indices	x..y	No
	..^		x..^y	
Switch & with	switch	Switch expression	num switch { 1 => true, _ => false }	No
	with	With expression	rec with { X = 123 }	No
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

Category	Operator symbol	Operator name	Example	User over
	>>	Shift right	x >> 1	Yes
	>>>	Unsigned 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 is y	No
	as	Type conversion	x as y	No
Equality	==	Equals	x == y	Yes
	!=	Not equals	x != y	Yes
Bitwise And	&	And	x & y	Yes
Bitwise Xor	^	Exclusive Or	x ^ y	Yes
Bitwise Or		Or	x y	Yes
Conditional And	&&	Conditional And	x && y	Via &

Category	Operator symbol	Operator name	Example	Useful over
Conditional Or	<code> </code>	Conditional Or	<code>x y</code>	Via <code> </code>
Null coalescing	<code>??</code>	Null coalescing	<code>x ?? y</code>	No
Conditional	<code>?:</code>	Conditional	<code>isTrue ? thenThis : elseThis</code>	No
Assignment and lambda	<code>=</code>	Assign	<code>x = y</code>	No
	<code>*=</code>	Multiply self by	<code>x *= 2</code>	Via <code>*</code>
	<code>/=</code>	Divide self by	<code>x /= 2</code>	Via <code>/</code>
	<code>%=</code>	Remainder & assign to self	<code>x %= 2</code>	
	<code>+=</code>	Add to self	<code>x += 2</code>	Via <code>+</code>
	<code>-=</code>	Subtract from self	<code>x -= 2</code>	Via <code>-</code>
	<code><<=</code>	Shift self left by	<code>x <<= 2</code>	Via <code><</code>
	<code>>>=</code>	Shift self right by	<code>x >>= 2</code>	Via <code>></code>
	<code>>>>=</code>	Unsigned shift self right by	<code>x >>>= 2</code>	Via <code>></code>

Category	Operator symbol	Operator name	Example	Use over
	&=	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](#)”).

Null-Coalescing Assignment Operator

The `??=` operator (introduced in C# 8) is the *null-coalescing assignment operator*. It says, “If the operand to the left is null, assign the right operand to the left operand.” Consider the following:

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

This is equivalent to:

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

The `??=` operator is particularly useful in implementing lazily calculated properties. We’ll cover this topic later, in “[Calculated Fields and Lazy Evaluation](#)”.

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());
```

Null-conditional expressions also work with indexers:

```
string[] words = null;  
string word = words?[1]; // word is null
```

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**”). 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 variable declaration introduces a new variable, optionally initializing it with an expression. 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**”):

```
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:

```

int x;
{
    int y;
    int x;           // Error - x already defined
}
{
    int y;           // OK - y not in scope
}
Console.Write (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:

```

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:

```

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

```

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

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 patterns have been introduced in recent versions of C#; see “[Patterns](#)” for a full discussion.

You can also switch on *types* (from C# 7):

```

TellMeTheType (12);
TellMeTheType ("hello");
TellMeTheType (true);

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;
        case DateTime:
            Console.WriteLine ("It's a DateTime");
            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](#)”).

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.

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`, it's 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

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 suite = "spades";

string cardName = (cardNumber, suite) switch
{
    (13, "spades") => "King of spades",
```

```
(13, "clubs") => "King of clubs",  
...  
};
```

Many more options are possible through the use of *patterns* (see **“Patterns”**).

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. For example, the following writes 012:

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

`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; used typically 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 .NET types 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);
```

Here's the output:

```
b
e
e
r
```

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

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](#)”). 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:

```
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](#)”):

```
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](#)” and “[IDisposable, Dispose, and Close](#)”).

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 *.dll* files that serve as units of deployment (described in [Chapter 17](#)).

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.

File-Scoped Namespaces

Often, you will want all the types in a file to be defined in one namespace:

```
namespace MyNamespace
{
    class Class1 {}
    class Class2 {}
}
```

From C# 10, you can accomplish this with a *file-scoped namespace*:

```
namespace MyNamespace; // Applies to everything that follows in the file.

class Class1 {}        // inside MyNamespace
class Class2 {}        // inside MyNamespace
```

File-scoped namespaces reduce clutter and eliminate an unnecessary level of indentation.

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;

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` (Windows Forms).

A `using` directive can be nested within a namespace itself to limit the scope of the directive.

The global using Directive

From C# 10, if you prefix a `using` directive with the `global` keyword, the directive will apply to all files in the project or compilation unit:

```
global using System;  
global using System.Collection.Generic;
```

This lets you centralize common imports and avoid repeating the same directives in every file.

`global using` directives must precede nonglobal directives and cannot appear inside namespace declarations. The global directive can be used with `using static`.

Implicit global usings

From .NET 6, project files allow for implicit `global using` directives. If the `ImplicitUsings` element is set to `true` in the project file (the default for new projects), the following namespaces are automatically imported:

```
System  
System.Collections.Generic  
System.IO  
System.Linq  
System.Net.Http  
System.Threading  
System.Threading.Tasks
```

Additional namespaces are imported, based on the project SDK (Web, Windows Forms, WPF, and so on).

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;

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 ([Chapter 3](#)), 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

Names declared in outer namespaces can be used 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; }
```

Alias any type (C# 12)

From C# 12, the `using` directive can alias any kind of type, including, for instance, arrays:

```
using NumberList = double[];  
NumberList numbers = { 2.5, 3.5 };
```

You can also alias tuples—we cover this in “[Aliasing Tuples \(C# 12\)](#)”.

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;  
  
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;

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;

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 25](#).