# Passing arguments by value

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

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

```
static void Main()
{
  int x = 8;
```

```
    int x = 8;
    Foo (x);
    Console.WriteLine (x);
  }
}

// Make a copy of x
// x will still be 8
```

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

Passing a reference-type argument by value copies the *reference*, but not the object. In the following example, Foo sees the same StringBuilder object that Main instantiated, but has an independent *reference* to it. In other words, sb and fooSB are separate variables that reference the same StringBuilder object:

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

```
    }

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

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

# The ref modifier

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

```
class Test
```

```
class Test
{
```



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

  static void Main()
  {
    int x = 8;
    Foo (ref  x);
    Console.WriteLine (x);
```

```
    Foo (ref x);
    Console.WriteLine (x);
  }
}

// Ask Foo to deal directly with x
// x is now 9
```

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

The `ref` modifier is essential in implementing a swap method (later, in "Generics" on page 101 in Chapter 3, we will show how to write a swap method that works with any type):

```
class Test
{
```

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

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

```
    Console.WriteLine (x);
    Console.WriteLine (y);
  }
}

// Teller
// Penn
```

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 it:

An `out` argument is like a `ref` argument, except it:

- Need not be assigned before going into the function

§ An exception to this rule is when calling COM methods. We discuss this in Chapter 25.



- Must be assigned before it comes *out* of the function

The `out` modifier is most commonly used to get multiple return values back from a method. For example:

```
class Test
{
  static void Split (string name, out string firstNames,
```

```
static void Split (string name, out string firstNames,
                   out string lastName)
{
   int i = name.LastIndexOf (' ');
   firstNames = name.Substring (0, i);
   lastName   = name.Substring (i + 1);
}
```

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

```
// Stevie Ray
// Vaughn
```

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

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

### 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);
```

```
    Console.WriteLine (x);
    y = 1;
    Console.WriteLine (x);
  }
}

// x is 0
// Mutate y
// x is 1
```

## The params modifier

The **params** parameter modifier may be specified on the last parameter of a method so that the method accepts any number of parameters of a particular type. The

The **params** parameter modifier may be specified on the last parameter of a method so that the method accepts any number of parameters of a particular type. The parameter type must be declared as an array. For example:

```
class Test
{
  static int Sum (params int[] ints)
  {
    int sum = 0;
```



```
    for (int i = 0; i < ints.Length; i++)
      sum += ints[i];
    return sum;
  }
```

```
}

// Increase sum by ints[i]
// 10

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

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

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

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

### Optional parameters (C# 4.0)

From C# 4.0, 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); }
```

Optional parameters may be omitted when calling the method:

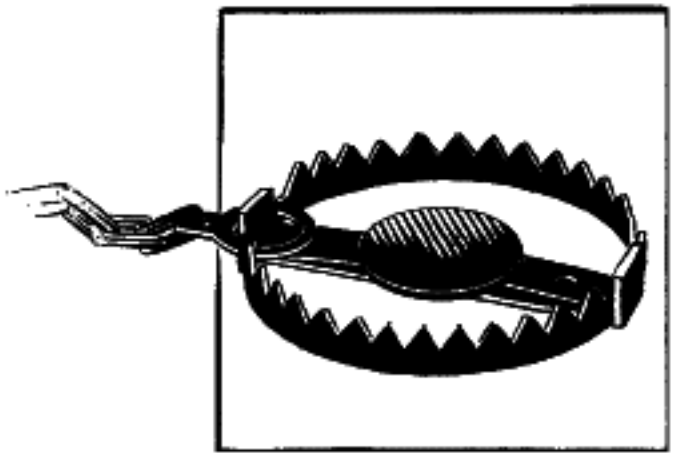
```
Foo();      // 23
```

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.

wherever it is used.



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

The default value of an optional parameter must be specified by a constant expression, or a parameterless constructor of a value type. Optional parameters cannot be marked with `ref` or `out`.

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

and the default value of 0 is passed to y:

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

```
void Test()
{
  Foo(1);
}

// 1, 0
```



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

### Named arguments (C# 4.0)

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

## Named arguments (x: 1)

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

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

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

// 1, 2
```

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

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

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`:

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 parameters:

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

However, there is a restriction: positional parameters must come before named arguments. So we couldn't call `Foo` like this:

```
Foo (x:1, 2);          // Compile-time error
```

```
Foo (x:1, 2);          // Compile-time error
```

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

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

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

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

This is particularly useful when calling COM APIs, and is discussed in detail in Chapter 25.



# var—Implicitly Typed Local Variables

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

var (introduced in C# 3.0) in place of the type declaration. For example:

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

This is precisely equivalent to:

```
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";
```

```
x = "hello";

// Compile-time error; x is of type int
```

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

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

What type is x?

In "Anonymous Methods" on page 134 in Chapter 4, we will describe a scenario where the use of `var` is mandatory.

# Expressions and Operators

# 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
```

Complex expressions can be built because an operand may itself be an expression, such as the operand (12 * 30) in the following example:

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

Operators in C# are 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, where 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. For example:

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

A void expression, since it has no value, cannot be used as an operand to build more complex expressions:

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

```
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 actually carries the assignment value, 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)
```

This style of expression can be used to initialize multiple values:

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

The *compound assignment operators* are syntactic shortcuts that combine assign

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

```
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 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

# 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 due to left associativity:

is evaluated as follows due to left associativity:

```
( 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*, lambda, null coalescing, and conditional operator 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

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" on page 153 in Chapter 4.

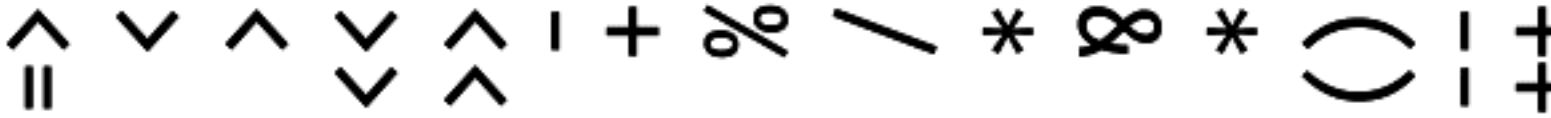
*Table 2-3. C# operators (categories in order of precedence)*

| Category | Operator symbol | Operator name | Example | User-overloadable |
|---|---|---|---|---|
| Primary | `()` | Grouping | `while(x)` | No |
| | `.` | Member access | `x.y` | No |



| Category | Operator symbol | Operator name | Example | User-overloadable |
|---|---|---|---|---|
| | `->` | Pointer to struct (unsafe) | `x->y` | No |

```
[]
++
--
new
stackalloc
typeof
checked
unchecked
sizeof
+
-
!
~
++
--
```



<=

Function call

Array/index

Post-increment

Post-decrement

Create instance

Unsafe stack allocation

Get type from identifier

Integral overflow

Integral overflow check on
Integral overflow check off
Get size of struct
Positive value of
Negative value of
Not
Bitwise complement
Pre-increment
Pre-decrement
Cast

Cast
Value at address (unsafe)
Address of value (unsafe)
Multiply
Divide
Remainder
Add
Subtract
Shift left
Shift right

Shift left
Shift right
Less than
Greater than
Less than or equal to

```
x()
a[x]
```

```
x++
x--
new Foo()
stackalloc(10)
```

```
stackalloc(10)
typeof(int)
checked(x)
unchecked(x)
sizeof(int)
+x
-x
!x
~x
++x
--x
(int)x
*x
```

```
*x
&x
x * y
x / y
x % y
x + y
x - y
x >> 1
x << 1
x < y
x > y
x <= y
```

```
x <= y
```

- No
- Via indexer
- Yes
- Yes
  No
  No
  No
  No
  No
  No

No
Yes
Yes
Yes
Yes
Yes
Yes
No
No
No
Yes
Yes

Yes
Yes
Yes
Yes
Yes
Yes
Yes
Yes
Yes
Yes

C#

# Basics

Unary

Multiplicative

Additive

Shift

Relational

| Category | Operator symbol | Operator name | Example | User-overloadable |
|---|---|---|---|---|
| | >= | Greater than or equal to | x >= y | Yes |
| Equality | | | | |
| Logical And | | | | |
| Logical Xor | | | | |
| Logical Or | | | | |
| Conditional And | | | | |
| Conditional Or | | | | |
| Nullness | | | | |

Null coalescing

Conditional

```
is
as
==
!=
&
^
|
&&
||
??
```

??
?:
=
*=
/=
+=
-=
<<=
>>=
&=
^=

Type is or is subclass of

Type conversion

Type conversion

Equals

Not equals

And

Exclusive Or

Or

Conditional And

Conditional Or

Null coalescing

Conditional

```
x is y
```

```
x is y
x as y
x == y
x != y
x & y
x ^ y
x | y
x && y
x || y
x ?? y
isTrue ? thenThis
```

```
isTrue ? thenThis
Value : elseThis
Value
x = y
x *= 2
x /= 2
x += 2
x -= 2
x <<= 2
x >>= 2
x &= 2
x ^= 2
```

No

No
No
Yes
Yes
Yes
Yes
Yes
Via &
Via |
No
No
No

No

Via *

Via /

Via +

Via -

Via <<

Via >>

Via &

Via ^

Assignment

Assign

Multiply self by

Multiply self by

Divide self by

Add to self

Subtract from self

Shift self left by

Shift self right by

And self by

Exclusive-Or self by

|= Or self by x |= 2 Via |

| | | | | |
|---|---|---|---|---|
| | \|= | Or self by | x \|= 2 | Via \| |
| Lambda | => | Lambda | x => x + 1 | No |

# Statements

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

# Declaration Statements

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



```
string someWord = "rosebud";
```

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

A constant declaration is like a variable declaration, except that the variable cannot be changed after it has been declared, and the initialization must occur with the declaration:

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

# Local variables

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

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

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

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.

# 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. The possible expression statements are:

- 
- 
-

- 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;
x++;
```

```
x++;
y = Math.Max (x, 5);
Console.WriteLine (y);
sb = new StringBuilder();
new StringBuilder();
```

```
// Assignment expression
// Increment expression
// Assignment expression
// Method call expression
// Assignment expression
// Object instantiation expression
```

```
// 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 (`?:`)

Selection statements (`if`, `switch`)

Conditional operator (`?:`)

Loop statements (`while`, `do..while`, `for`, `foreach`)

This section covers the simplest two constructs: the `if-else` statement and the `switch` statement.

# The if statement

An `if` statement executes a body of code depending on whether a `bool` expression is true. For example:

```
if (5 < 2 * 3)
{
  Console.WriteLine ("true");        // True
}
```

If the body of code is a single statement, you can optionally omit the braces:

```
if (5 < 2 * 3)
  Console.WriteLine ("true");        // True
```

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

# The else clause

An `if` statement is optionally followed by 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)
```

```
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. For example:

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

This is semantically identical to:

```
if (true)
```

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

We can change the execution flow by moving the braces:

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

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

```
static void TellMeWhatICanDo (int age)
{
  if (age >= 35)
```

```
{
  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 "elsif" 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` statement.



# The switch statement

`switch` statements let you branch program execution based on a selection of possible values that a variable may have. `switch` statements may result in cleaner code than

switch statements let you branch program execution based on a selection of possible values that a variable may have. switch statements may result in cleaner code than multiple if statements, since switch statements require an expression to be evaluated only once. For instance:

```
static void ShowCard(int cardNumber)
{
  switch (cardNumber)
  {
    case 13:
      Console.WriteLine ("King");
      break;
    case 12:
      Console.WriteLine ("Queen");
      break;
    case 11:
      Console.WriteLine ("Jack");
      break;
    case -1:                          // Joker is -1
      goto case 12;                   // In this game joker counts as queen
    default:                          // Executes for any other cardNumber
      Console.WriteLine (cardNumber);
      break;
  }
}
```

You can only switch on an expression of a type that can be statically evaluated, which

You can only switch on an expression of a type that can be statically evaluated, which restricts it to the built-in integral types, `string` type, and `enum` types.

At the end of each `case` clause, you must say explicitly where execution is to go next, with some kind of jump statement. 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 `case`s sequentially:

```
switch (cardNumber)
{
```

```
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.

# 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:



```
int i = 0;
```

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

```
OUTPUT:
0
1
2
```

`do-while` loops differ in functionality from `while` loops only in that they test the expression *after* the statement block has executed (ensuring that the block is always

`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
```

*Initialization clause*

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

*Condition clause*

*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 (where 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 may be omitted. One can implement an infinite loop such as the following (though `while(true)` may be used instead):

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

# foreach loops

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

```
foreach (char c in "beer")   // c is the iteration variable
  Console.WriteLine (c);
```

```
OUTPUT:
b
e
```

e
e
r

We define enumerable objects in "Enumeration and Iterators" on page 143 in Chapter 4.

# Jump Statements

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

Jump statements obey the reliability rules of `try` statements (see "try Statements and Exceptions" on page 134 in Chapter 4). This means that:

- A jump out of a `try` block always executes the `try`'s `finally` block before reaching the target of the jump.

finally block before reaching the target of the jump.

- A jump cannot be made from the inside to the outside of a finally block.



# 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

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 the statement block. The form is as follows:

```
goto statement-label;
```

```
goto statement-label;
```

Or, when used within a `switch` statement:

```
goto case case-constant;
```

A label statement is just a placeholder in a code block, 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;
```

```
    i++;
    goto startLoop;
  }
```

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

The goto *case-constant* transfers execution to another case in a switch block (see "The switch statement" on page 52).



# The return statement

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

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

```
    decimal p = d * 100m;
    return p;              // Return to the calling method with value
  }
```

A return statement can appear anywhere in a method.

## The throw statement

The throw statement throws an exception to indicate an error has occurred (see "try Statements and Exceptions" on page 134 in Chapter 4):

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

# Miscellaneous Statements

The lock statement is a syntactic shortcut for calling the Enter and Exit methods of the Monitor class (see Chapter 19).

The using statement provides an elegant syntax for calling Dispose on objects that implement IDisposable, within a finally block (see "try Statements and Exceptions" on page 134 in Chapter 4 and "IDisposable, Dispose, and Close" on page 475 in Chapter 12).

tions" on page 134 in Chapter 4 and "IDisposable, Dispose, and Close" on page 475 in Chapter 12).

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

# Namespaces

A namespace is a domain within which type names must be unique. Types are typically organized into hierarchical namespaces—both to avoid naming conflicts and to make type names easier to find. 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:

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();
```



Namespaces are independent of assemblies, which are units of deployment such as an *.exe* or *.dll* (described in Chapter 16).

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

The `namepace` 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

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

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

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

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

# The using Directive

# The using Directive

The `using` directive *imports* a namespace. This is a convenient way to refer to types without their fully qualified names. This example is semantically identical to our previous example:

```
using Outer.Middle.Inner;

class Test
{
  static void Main()
  {
    Class1 c;
  }
```

```
    }
  }
```



# Rules Within a Namespace

## Name scoping

Names declared in outer namespaces can be used unqualified within inner namespaces. Here, the names `Middle` and `Class1` are implicitly imported into `Inner`:

```
namespace Outer
{
  namespace Middle
```

```
    namespace Middle
    {
      class Class1 {}

      namespace Inner
      {
        class Class2 : Class1
      }
    }
  }

  {}
```

If you want to refer to a type in a different branch of your namespace hierarchy, you

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.Foo f2;
    }
  }
}

// = Outer.Inner.Foo
// = Outer.Foo
```

```
// = Outer.Foo
```



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 {}
```

```
  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
```

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

### Nested using directive

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 {}
}
```

```
  class class3 : class1 {}
}

// Compile-time error
```

# Aliasing Types and Namespaces

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

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

An entire namespace can be aliased, as follows:

```
using R = System.Reflection;
```

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

# Advanced Namespace Features

## Extern

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

Library 1:

```
// csc target:library /out:Widgets1.dll widgetsv1.cs

namespace Widgets
{
```

```
{
  public class Widget {}
}
```

Library 2:

```
// csc target:library /out:Widgets2.dll widgetsv2.cs
```

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

Application:

```
// csc /r:Widgets1.dll /r:Widgets2.dll application.cs
```

```
// csc /r:Widgets1.dll /r:Widgets2.dll application.cs

using Widgets;
class Test
{
  static void Main()
```



```
  {
    Widget w = new Widget();
  }
}
```