C# Flow Control

Topics Covered

- Flow-Control Structures
  - Sequential
  - Selection
  - Iteration
- Structured Programming
- if / else Selection Statements
- The Conditional Operator
- Switch Statements
- while Loops
- for Loops
- do ... while Loops
- break and continue Statements
- foreach Loops
  - The IEnumerable Interface

Flow-Control Structures

C# is a sequential language. Normally, statements execute one after the other in the order that they were written, a process called *sequential execution*.

C# provides language structures to specify situations where the next line of code is not the next to execute. These are called flow-control structures.

The C# flow-control statements fall into just three categories:

- Sequential statements – this is the default behavior
- Selection statements – the next line of code is selected based on a test
- Iteration statements – the execution loops where the # repetitions is based on a test

Any C# application is built by chaining (or nesting) these control structures together.

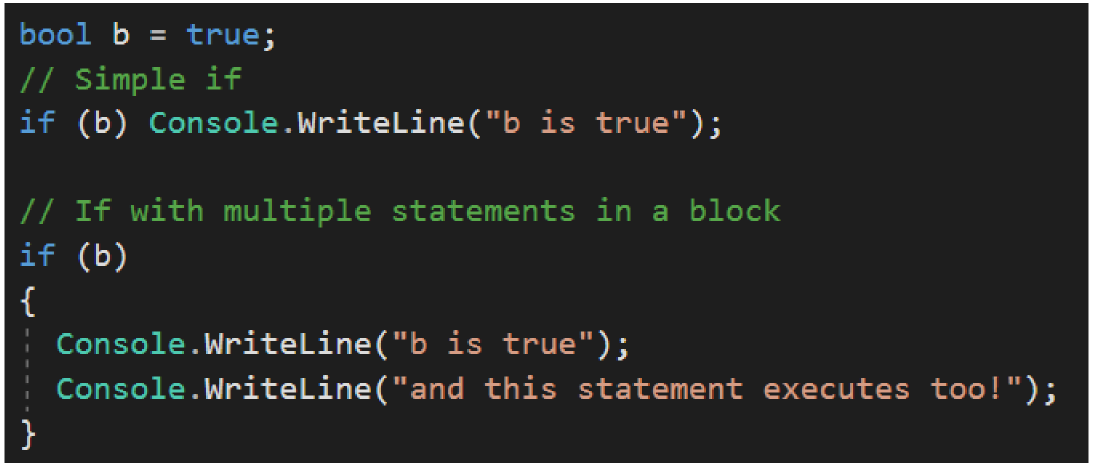
The Nefarious *goto*

Early programming languages made frequent use of *goto* statements, which allowed transfer of control to a wide range of destinations within a program. Research during the 1960s found that this was a source of difficulty in writing and maintaining software, so programmers were challenged to begin using **structured programming**, where goto is replaced by the flow-control statements we find in modern languages like C#. This resulted in impressive improvements in the craft – shorter development times – because structured programs are clearer and easier to comprehend, debug, and modify.

Notably, C# still has a *goto* statement, but its usage is quite limited compared to the *goto* of past decades.

if

if is the simplest selection statement. It performs (selects) an action if a condition is true, or skips the action if the condition is false.



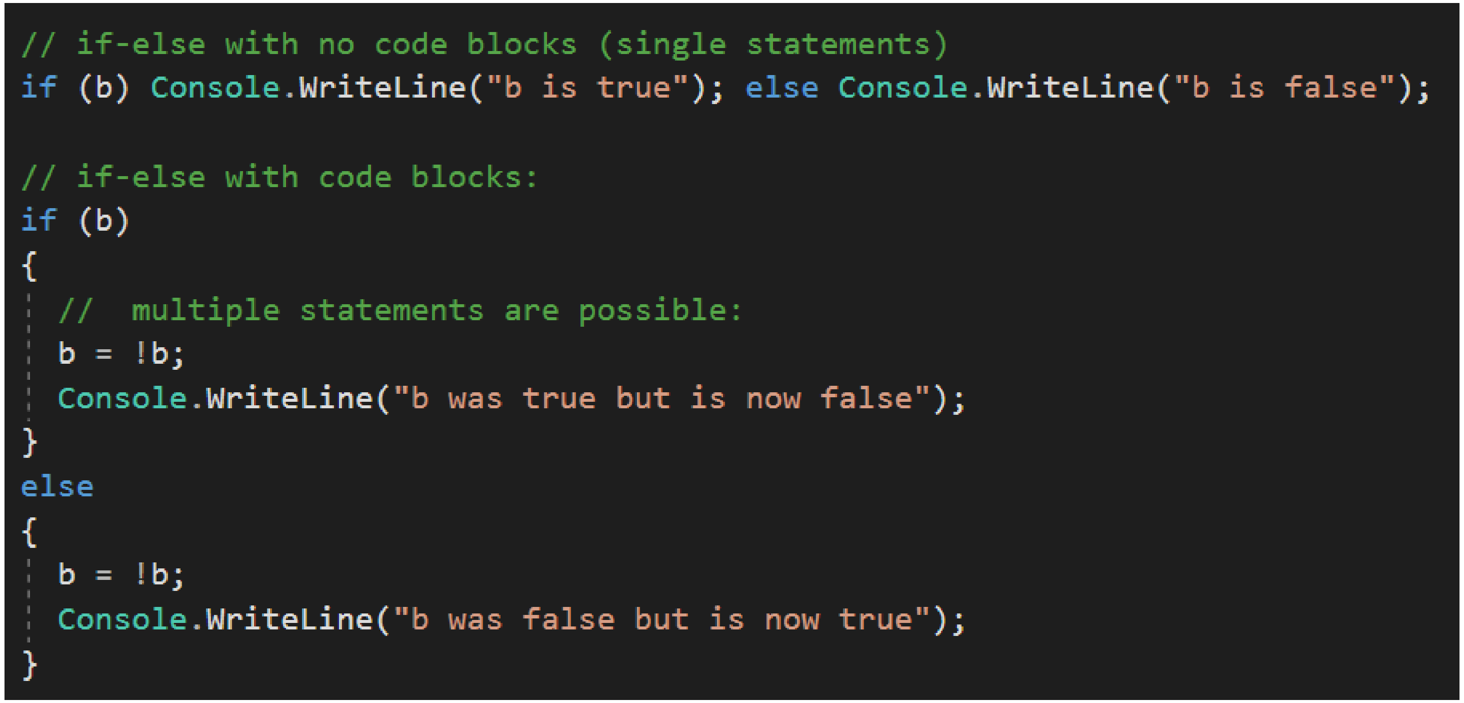
```
bool b = true;
// Simple if
if (b) Console.WriteLine("b is true");

// If with multiple statements in a block
if (b)
{
    Console.WriteLine("b is true");
    Console.WriteLine("and this statement executes too!");
}
```

The “action” may in fact be multiple lines of code contained within a code block.

Note the absence of “then”. The boolean expression is <u>always</u> within parentheses.

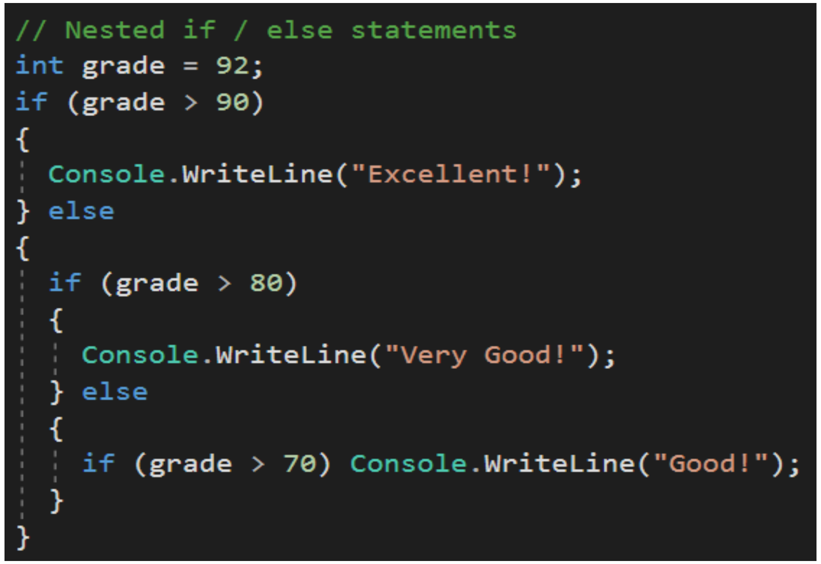
if / else

if / **else** allows us to select one action if the condition is true, and a different action when the condition is false:

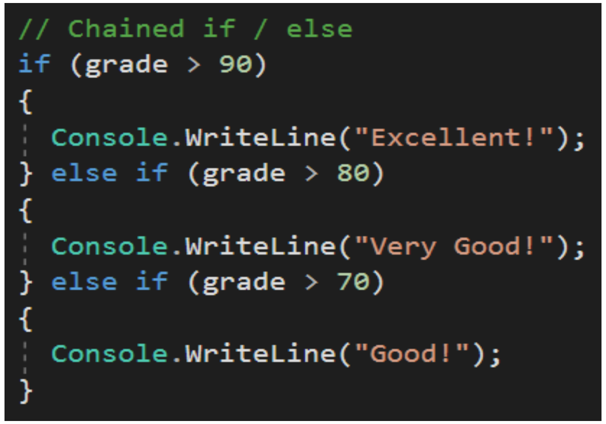


```
// if-else with no code blocks (single statements)
if (b) Console.WriteLine("b is true"); else Console.WriteLine("b is false");

// if-else with code blocks:
if (b)
{
  //  multiple statements are possible:
  b = !b;
  Console.WriteLine("b was true but is now false");
}
else
{
  b = !b;
  Console.WriteLine("b was false but is now true");
}
```

Nested and Chained if / else



```
// Nested if / else statements
int grade = 92;
if (grade > 90)
{
    Console.WriteLine("Excellent!");
} else
{
    if (grade > 80)
    {
        Console.WriteLine("Very Good!");
    } else
    {
        if (grade > 70) Console.WriteLine("Good!");
    }
}
```

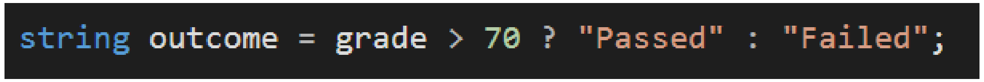


```
// Chained if / else
if (grade > 90)
{
    Console.WriteLine("Excellent!");
} else if (grade > 80)
{
    Console.WriteLine("Very Good!");
} else if (grade > 70)
{
    Console.WriteLine("Good!");
}
```

These two code samples behave identically. The chained if/else is more concise and easier to comprehend.

The Conditional Operator ?:

C# has a conditional operator. It is a ternary operator – it takes 3 operands.



```
string outcome = grade > 70 ? "Passed" : "Failed";
```

Operand: 1 2 3

The 1st operand is a boolean expression. The 2nd operand is the result of the expression if the 1st operand is true. The 3d operand is the result of the expression if the 1st operand is false.

switch

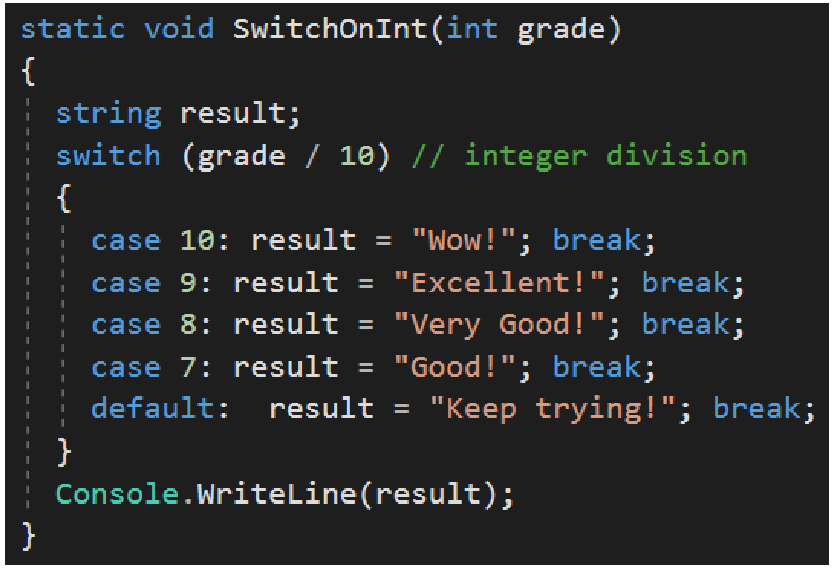
if is a single-selection statement. *if/else* is a double-selection statement. *switch* is a multiple-selection statement – it allows you to select among multiple actions based on a value.

When control reaches the switch statement, the program evaluates (grade / 10) (the *switch expression*).

It then attempts to match the switch expression's value to one of the case labels. If a match is found, the statement(s) following that case label are executed.

If no match is found, then the default statement(s) (if there is a default case) will execute.

Every case ends with a break statement. This transfers control to the first statement after the switch structure. break is *required*.*



```
static void SwitchOnInt(int grade)
{
    string result;
    switch (grade / 10) // integer division
    {
        case 10: result = "Wow!"; break;
        case 9: result = "Excellent!"; break;
        case 8: result = "Very Good!"; break;
        case 7: result = "Good!"; break;
        default:  result = "Keep trying!"; break;
    }
    Console.WriteLine(result);
}
```

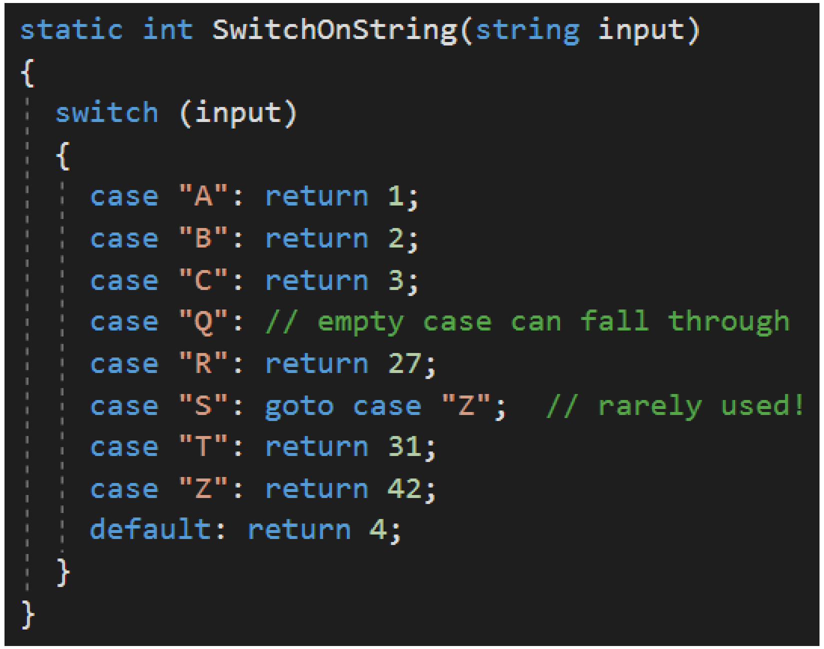
case labels

A few things to notice about switch statements:

The switch expression is evaluated at run-time, but the case labels are always constants or literals. They cannot be runtime expressions.

Every case must end with either a return statement or a break statement, with two exceptions:

- An empty case statement can fall through to the next statement.
- A case statement can goto another case label.



```
static int SwitchOnString(string input)
{
    switch (input)
    {
        case "A": return 1;
        case "B": return 2;
        case "C": return 3;
        case "Q": // empty case can fall through
        case "R": return 27;
        case "S": goto case "Z";  // rarely used!
        case "T": return 31;
        case "Z": return 42;
        default: return 4;
    }
}
```

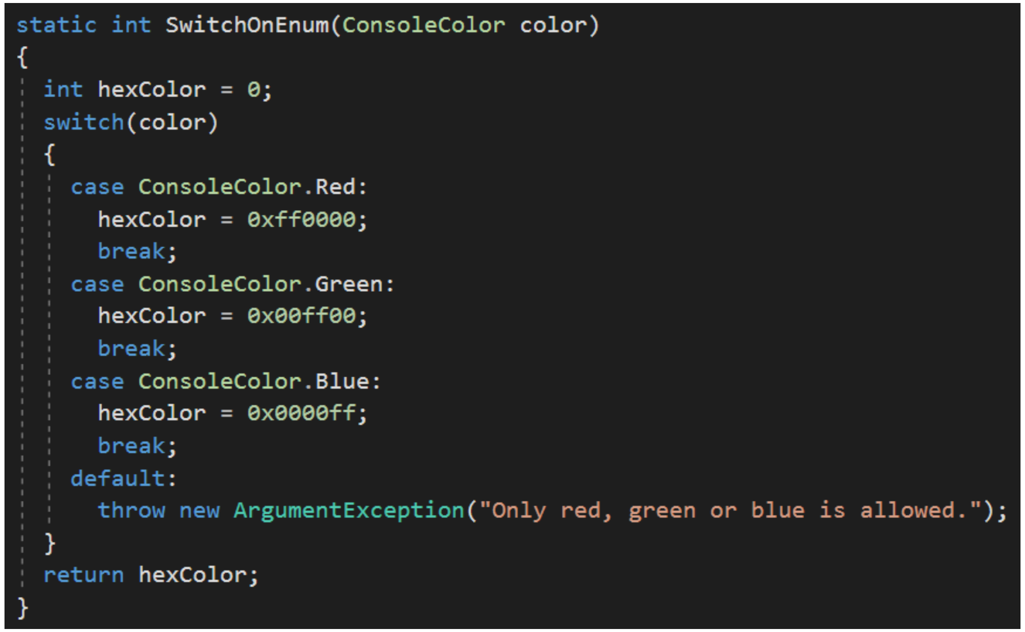
Unintentional fall-through in switch statements is a common source of bugs in the C languages, so C# places these restrictions on the switch statement.

Switching on Enums

switch statements can work with strings, all integer types, char, and enums.

Under the hood, C# enums are based on integer types, so ….

The example here uses the ConsoleColor enum.

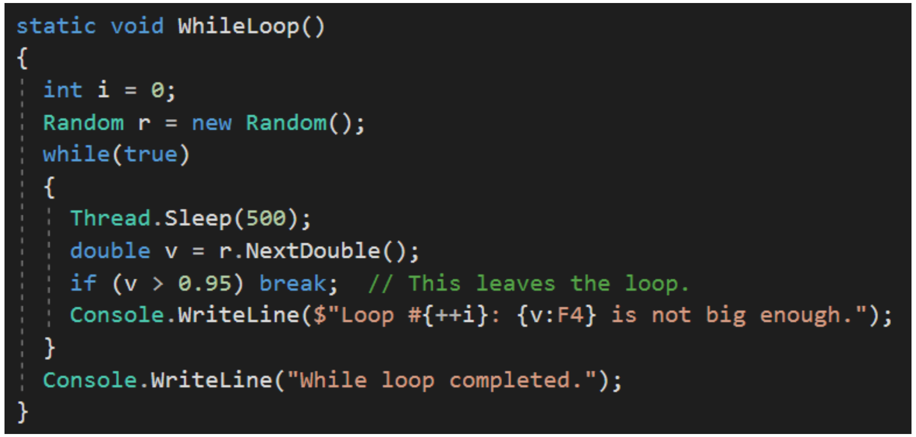


```
static int SwitchOnEnum(ConsoleColor color)
{
  int hexColor = 0;
  switch(color)
  {
    case ConsoleColor.Red:
      hexColor = 0xff0000;
      break;
    case ConsoleColor.Green:
      hexColor = 0x00ff00;
      break;
    case ConsoleColor.Blue:
      hexColor = 0x0000ff;
      break;
    default:
      throw new ArgumentException("Only red, green or blue is allowed.");
  }
  return hexColor;
}
```

while loops

while loops are the first of the 4 C# iteration statements that we will look at.

```
while(condition)
{
    // loop statements
}
```



```
static void WhileLoop()
{
    int i = 0;
    Random r = new Random();
    while(true)
    {
        Thread.Sleep(500);
        double v = r.NextDouble();
        if (v > 0.95) break;  // This leaves the loop.
        Console.WriteLine($"Loop #{++i}: {v:F4} is not big enough.");
    }
    Console.WriteLine("While loop completed.");
}
```

The first line of the while loop evaluates the loop *condition*. If *condition* is true, the loop is entered.

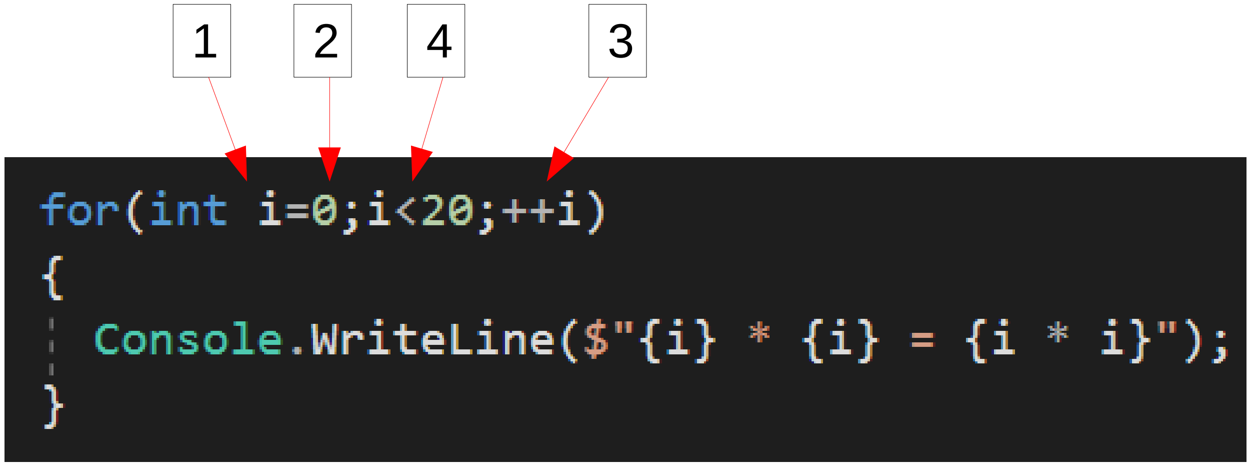
The loop statements execute, and the loop *condition* is evaluated again. Looping will continue until either:

1) *condition* evaluates false
2) the loop body executes a break statement.

Counter-controlled Iteration: for loops

for loops have 4 essential elements:

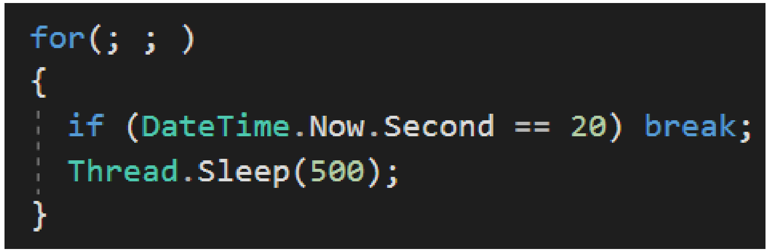
1. A control variable
2. The control variable’s initial value
3. The control variable’s increment applied during each loop iteration
4. The loop-continuation condition that determines if looping should continue.



```
for(int i=0;i<20;++i)
{
    Console.WriteLine($"{i} * {i} = {i * i}");
}
```

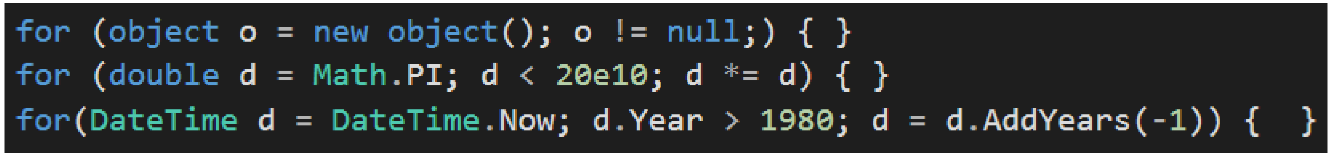
The for Loop Header

All parts of a for loop's header are optional. The for loop shown on the right is perfectly valid. The loop body provides a break condition – otherwise it would be an infinite loop.



```
for(; ; )
{
    if (DateTime.Now.Second == 20) break;
    Thread.Sleep(500);
}
```

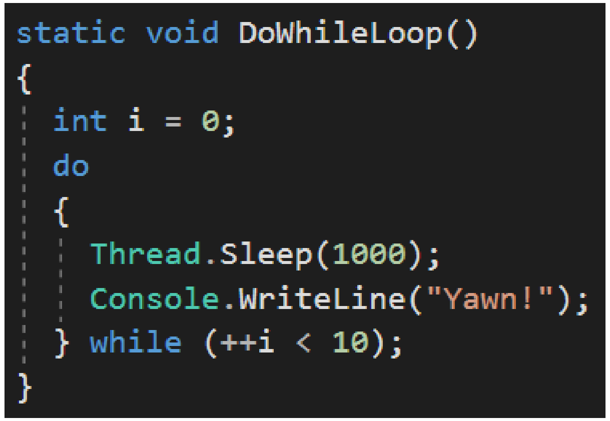
The for loop header is remarkably flexible. The examples below are all valid for loops.



```
for (object o = new object(); o != null;) { }
for (double d = Math.PI; d < 20e10; d *= d) { }
for(DateTime d = DateTime.Now; d.Year > 1980; d = d.AddYears(-1)) {  }
```

do ... while Loops

The do … while loop is much like the while loop, except the while condition is evaluated at the bottom of the loop instead of at the top. This change guarantees that the loop will be entered at least once.

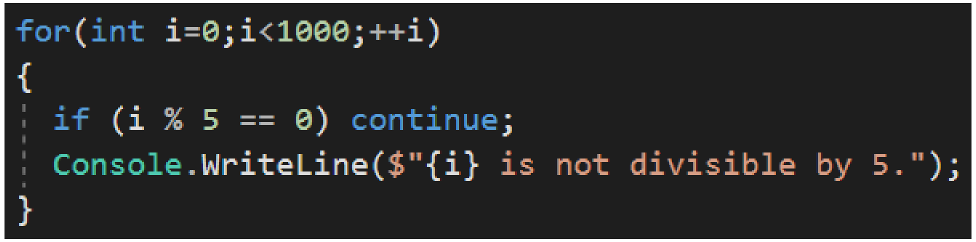


```
static void DoWhileLoop()
{
    int i = 0;
    do
    {
        Thread.Sleep(1000);
        Console.WriteLine("Yawn!");
    } while (++i < 10);
}
```

break and continue statements

We’ve already seen examples where the break statement is used to leave a loop and transfer control to the first statement after the loop.

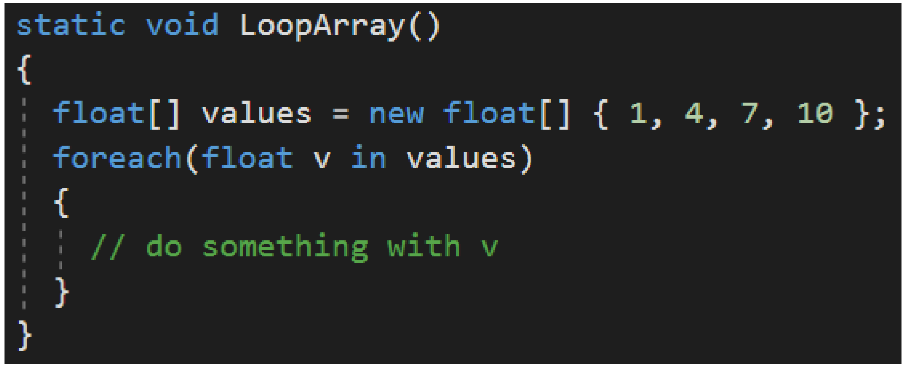
In contrast, the continue statement skips any remaining statements in the loop body and transfers control to the top of the loop.



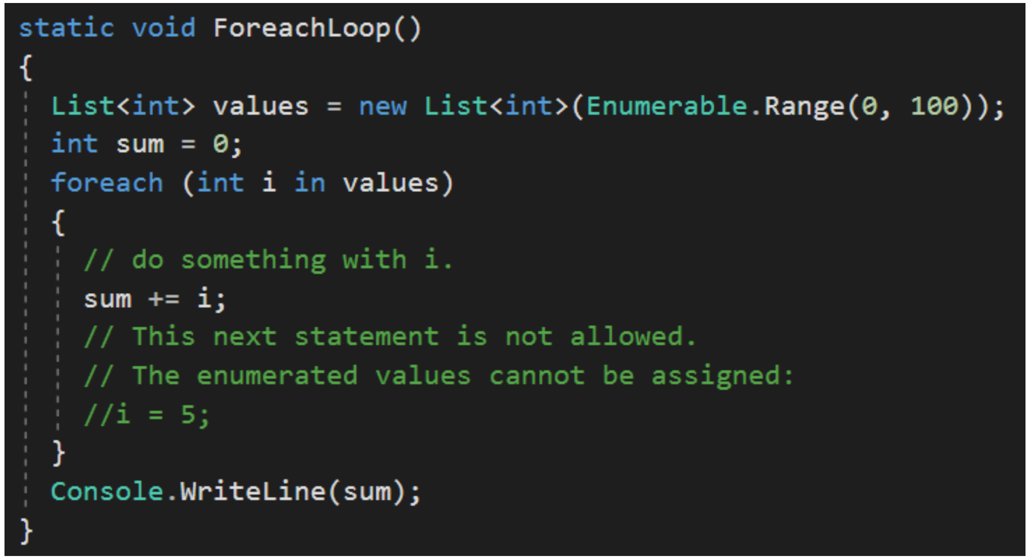
```
for(int i=0;i<1000;++i)
{
    if (i % 5 == 0) continue;
    Console.WriteLine($"{i} is not divisible by 5.");
}
```

foreach loops

The fourth C# iteration statement is foreach. With foreach, we can loop through members of the various collection classes, including Lists, Arrays, Dictionaries, and more.



```csharp
static void LoopArray()
{
    float[] values = new float[] { 1, 4, 7, 10 };
    foreach(float v in values)
    {
        // do something with v
    }
}
```



```csharp
static void ForeachLoop()
{
    List<int> values = new List<int>(Enumerable.Range(0, 100));
    int sum = 0;
    foreach (int i in values)
    {
        // do something with i.
        sum += i;
        // This next statement is not allowed.
        // The enumerated values cannot be assigned:
        //i = 5;
    }
    Console.WriteLine(sum);
}
```

IEnumerable Interface

The collection classes that let us use foreach all have something in common – they implement the IEnumerable interface.

We can look for example at the Array class definition. From there, we have to follow the interface inheritance down three levels, but we eventually find IEnumerable:

Array ==> IList ==> ICollection ==> IEnumerable

```
public abstract class Array : ICloneable, System.Collections.IList,
System.Collections.IStructuralComparable, System.Collections.IStructuralEquatable
```

```
public interface IList : System.Collections.ICollection
```

```
public interface ICollection : System.Collections.IEnumerable
```

This same mechanism allows us to create our own classes that implement IEnumerable. Instances of those classes can then be used in foreach loops.

Topics Covered

- Flow-Control Structures
  - Sequential
  - Selection
  - Iteration
- Structured Programming
- if / else Selection Statements
- The Conditional Operator
- Switch Statements
- while Loops
- for Loops
- do ... while Loops
- break and continue Statements
- foreach Loops
  - The IEnumerable Interface

Exercises