# Contents



# For Loops

## Counter-controlled loops

- Previously, when we learned about loop vocabulary, we looked at counter-controlled **`while`** loops
- Although counter-controlled loops can perform many different kinds of actions in the body of the loop, they all use very similar code for managing the counter variable
- Two examples of counter-controlled **`while`** loops:

```
int i = 0;
while(i < 10)
{
    Console.WriteLine($"{i}");
    i++;
}
Console.WriteLine("Done");
```

```
int num = 1, total = 0;
while(num <= 25)
{
    total += num;
    num++;
}
Console.WriteLine($"The sum is {total}");
```

Notice that in both cases, we've written the same three pieces of code:

  - Initialize a counter variable (`i` or `num`) before the loop starts
  - Write a loop condition that will become false when the counter reaches a certain value (`i < 10` or `num <= 25`)
  - Increment the counter variable at the end of each loop iteration, as the last line of the body

## `for` loop example and syntax

- This **`for`** loop does the same thing as the first of the two **`while`** loops above:

```
for(int i = 0; i < 10; i++)
{
    Console.WriteLine($"{i}");
}
Console.WriteLine("Done");
```

  - The **`for`** statement actually contains 3 statements in 1 line; note that they are separated by semicolons
  - The code to initialize the counter variable has moved inside the **`for`** statement, and appears first
  - Next is the loop condition, `i < 10`
  - The third statement is the increment operation, `i++`, which no longer needs to be written at the end of the loop body

- In general, **`for`** loops have this syntax:

```
for(<initialization>; <condition>; <update>)
{
    <statements>
}
```

  - The initialization statement is executed once, when the program first reaches the loop. This is where you declare and initialize the counter variable.
  - The condition statement works exactly the same as a **`while`** loop's condition statement: Before executing the loop's body, the computer checks the condition, and skips the body (ending the loop) if it is false.
  - The update statement is code that will be executed each time the loop's body *ends*, before checking the condition again. You can imagine that it gets inserted right before the closing `}` of the loop body. This is where you increment the counter variable.

- Examining the example in detail
  - When the computer executes our example **`for`** loop, it first creates the variable `i` and initializes it to 0

- Then it evaluates the condition `i < 10`, which is true, so it executes the loop's body. The computer displays "0" in the console.
- At the end of the code block for the loop's body, the computer executes the update code, `i++`, and changes the value of `i` to 1.
- Then it returns to the beginning of the loop and evaluates the condition again. Since it is still true, it executes the loop body again.
- This process repeats several more times. On the last iteration, `i` is equal to 9. The computer displays "9" on the screen, then increments `i` to 10 at the end of the loop body.
- The computer returns to the **`for`** statement and evaluates the condition, but `i < 10` is false, so it skips the loop body and proceeds to the next line of code. It displays "Done" in the console.

## Limitations and Pitfalls of Using `for` Loops

### Scope of the `for` loop's variable

- When you declare a counter variable in the **`for`** statement, its scope is limited to *inside* the loop
- Just like method parameters, it is as if the variable declaration happened just inside the opening `{`, so it can only be accessed inside that code block
- This means you cannot use a counter variable after the end of the loop. This code will produce a compile error:

```
int total = 0;
for(int count = 0; count < 10; count++)
{
    total += count;
}
Console.WriteLine($"The average is {(double) total /
 ↪  count}");
```

- If you want to use the counter variable after the end of the loop, you must declare it *before* the loop
- This means your loop's initialization statement will need to assign the variable its starting value, but not declare it
- This code works correctly, since `count` is still in scope after the end of the loop:

```
int total = 0;
int count;
for(count = 0; count < 10; count++)
{
    total += count;
}
Console.WriteLine($"The average is {(double) total /
 ↪  count}");
```

### Accidentally re-declaring a variable

- If your **for** loop declares a new variable in its initialization statement, it cannot have the same name as a variable already in scope
- If you want your counter variable to still be in scope after the end of the loop, you cannot also declare it in the **for** loop. This is why we had to write **for**(`count = 0...` instead of **for**(`int count = 0...` in the previous example: the name `count` was already being used.
- Since counter variables often use short, common names (like `i` or `count`), it is more likely that you'll accidentally re-use one that's already in scope
- For example, you might have a program with many **for** loops, and in one of them you decide to declare the counter variable outside the loop because you need to use it after the end of the loop. This can cause an error in a different **for** loop much later in the program:

  ```
  int total = 0;
  int i;
  for(i = 0; i < 10; i++)
  {
      total += i;
  }
  Console.WriteLine($"The average is {(double) total /
   ↪  i}");
  // Many more lines of code
  // ...
  // Some time later:
  for(int i = 0; i < 10; i++)
  {
      Console.WriteLine($"{i}");
  }
  ```

  The compiler will produce an error on the second **for** loop, because the name "i" is already being used.

- On the other hand, if all of your **for** loops declare their variables inside the **for** statement, it is perfectly fine to reuse the same variable name. This code does not produce any errors:

```
int total = 0;
for(int i = 0; i < 10; i++)
{
    total += i;
}
Console.WriteLine($"The total is {total}");
// Some time later:
for(int i = 0; i < 10; i++)
{
    Console.WriteLine($"{i}");
}
```

### Accidentally double-incrementing the counter

- Now that you know about **for** loops, you may want to convert some of your counter-controlled **while** loops to **for** loops
- Remember that in a **while** loop the counter must be incremented in the loop body, but in a **for** loop the increment is part of the loop's header
- If you just convert the header of the loop and leave the body the same, you will end up incrementing the counter *twice* per iteration. For example, if you convert this **while** loop:

```
int i = 0;
while(i < 10)
{
    Console.WriteLine($"{i}");
    i++;
}
Console.WriteLine("Done");
```

to this **for** loop:

```
for(int i = 0; i < 10; i++)
{
    Console.WriteLine($"{i}");
    i++;
}
Console.WriteLine("Done");
```

it will not work correctly, because `i` will be incremented by both the loop body and the loop's update statement. The loop will seem to "skip" every other value of `i`.

## More Ways to use `for` Loops

### Complex condition statements

- The condition in a **`for`** loop can be any expression that results in a `bool` value
- If the condition compares the counter to a variable, the number of iterations depends on the variable. If the variable comes from user input, the loop is also user-controlled, like in this example:

```
Console.WriteLine("Enter a positive number.");
int numTimes = int.Parse(Console.ReadLine());
for(int c = 0; c < numTimes; c++)
{
    Console.WriteLine("**********");
}
```

- The condition can compare the counter to the result of a method call. In this case, the method will get called on every iteration of the loop, since the condition is re-evaluated every time the loop returns to the beginning. For example, in this loop:

```
for(int i = 1; i <= (int) myItem.GetPrice(); i++)
{
    Console.WriteLine($"${i}");
}
```

  the `GetPrice()` method of `myItem` will be called every time the condition is evaluated.

### Complex update statements

- The update statement can be anything, not just an increment operation
- For example, you can write a loop that only processes the even numbers like this:

```
for(int i = 0; i < 19; i += 2)
{
    Console.WriteLine($"{i}");
}
```

- You can write a loop that decreases the counter variable on every iteration, like this:

```
for(int t = 10; t > 0; t--)
{
    Console.Write($"{t}...");
```

```
}
Console.WriteLine("Liftoff!");
```

### Complex loop bodies

- The loop body can contain more complex statements, including other decision structures
- **`if`** statements can be nested inside **`for`** loops, and they will be evaluated again on every iteration
- For example, in this program:

```
for(int i = 0; i < 8; i++)
{
    if(i % 2 == 0)
    {
        Console.WriteLine("It's my turn");
    }
    else
    {
        Console.WriteLine("It's your turn");
    }
    Console.WriteLine("Switching players...");
}
```

  On even-numbered iterations, the computer will display "It's my turn" followed by "Switching players…", and on odd-numbered iterations the computer will display "It's your turn" followed by "Switching players…"

- **`for`** loops can contain other **`for`** loops. This means the "inner" loop will execute all of its iterations each time the "outer" loop executes one iteration.
- For example, this program prints a multiplication table:

```
for(int r = 0; r < 11; r++)
{
    for(int c = 0; c < 11; c++)
    {
        Console.Write($"{r} x {c} = {r * c} \t");
    }
    Console.Write("\n");
}
```

  The outer loop prints the rows of the table, while the inner loop prints the columns. On a single iteration of the outer **`for`** loop (i.e. when `r = 2`), the inner **`for`** loop executes its body 11 times,

using values of `c` from 0 to 10. Then the computer executes the `Console.Write("\n")` to print a newline before the next "row" iteration.

### Combining `for` and `while` loops

- **`while`** loops are good for sentinel-controlled loops or user-input validation, and **`for`** loops are good for counter-controlled loops
- This program asks the user to enter a number, then uses a **`for`** loop to print that number of asterisks on a single line:

```
string userInput;
do
{
    Console.WriteLine("Enter a positive number, or
↪   \"Q\" to stop");
    userInput = Console.ReadLine();
    int inputNum;
    int.TryParse(userInput, out inputNum);
    if(inputNum > 0)
    {
        for(int c = 0; c < inputNum; c++)
        {
            Console.Write("*");
        }
        Console.WriteLine();
    }
} while(userInput != "Q");
```

  - The sentinel value "Q" is used to end the program, so the outer **`while`** loop repeats until the user enters this value
  - Once the user enters a number, that number is used in the condition for a **`for`** loop that prints asterisks using `Console.Write()`. After the **`for`** loop ends, we use `Console.WriteLine()` with no argument to end the line (print a newline).
  - Since the user could enter either a letter or a number, we need to use `TryParse` to convert the user's input to a number
  - If `TryParse` fails (because the user entered a non-number), `inputNum` will be assigned the value 0. This is also an invalid value for the loop counter, so we do not need to check whether `TryParse` returned **`true`** or **`false`**. Instead, we simply check whether `inputNum` is valid (greater than 0) before executing the **`for`** loop, and skip the **`for`** loop entirely if

`inputNum` is negative or 0.