# Conditionals and Repetition

## Introduction

In this section, we are going to learn how to control the flow of a program using what are called *control flow statements*. Control flow statements include *if*, *switch*, *for*, *foreach*, *while*, and *do while*. We will also learn how to handle exceptions using *try-catch-finally*, which is the control flow statement that helps us deal with adversity when an error occurs in our program.

## Conditional statements: if

Often we must choose among multiple courses of action based on the result of a comparison. The most popular choice for comparing two values and then deciding on an action to take is the if statement.

An if statement looks like the following:

```
if (true-false-condition) {
    what to do if the condition is true
}
```

Often, there is more than just one course of action that could be taken. If our logic is such that we either do one action if the condition is true or another action if it is false, then the syntax looks like this instead:

```
if (true-false-condition) {
   what to do if the condition is true
} else {
   what to do if the condition is false
}
```

Or, we might have even more options on what to do, based on the outcome of multiple comparisons. If that is the case, we would use *else if* to test those additional comparisons.

```
if (true-false-condition-1) {
   what to do if condition 1 is true
} else if (true-false-condition-2) {
   what to do if condition 2 is true
} else if (true-false-condition-3) {
   what to do if condition 3 is true
} else {
   what to do if none of the conditions were true
}
```

Note that we leave the structure of the if statement as soon as we find one condition to be true and do its corresponding action. Also note that the final else statement, which stipulates what to do if nothing was true, doesn't actually have to be there if there is no default action to take.

The conditions themselves can contain a mix of

- comparison operators, like >, <, <=, >=, ==, !=
- logical operators, like && (and) and || (or)
- The values *true* and *false*

## Example: if statements

Write a program that screens a person by their age to determine if they are eligible to attend a 21-and-over concert unaccompanied. If the user enters an age less than 0 or greater than 110, we will tell them they entered an invalid age. If they enter an age less than 18, we will tell them that no minors are allowed. If they enter an age between 18 and 21, we will tell them that they must be accompanied by an adult. Otherwise, we will congratulate them on being able to purchase a ticket.

```
int age;
Console.Write("Enter your age: ");
age = int.Parse(Console.ReadLine());
if (age < 0 || age > 110)
{
    Console.WriteLine("That age is invalid.");
} else if (age < 18)
{
    Console.WriteLine("Children are not allowed.");
} else if (age < 21)
{
    Console.WriteLine("You must be accompanied by an adult.");
} else
{
    Console.WriteLine("Congratulations. You may purchase a ticket.");
}
```

## Inline if statements

An if statement that merely assigns a value to a variable based on the result of a true-false condition can be written either as a full-blown if statement or as an inline if. Here is an example. Suppose we wish to determine if a number is even or odd. We can use the mod operator – the symbol % – to check.

```
string numType;
if (num % 2) == 0 {
       numType = "even";
} else {
       numType = "odd";
}
```

We can write it much more compactly like so:

```
string numType = num % 2 == 0 ? "even" : "odd";
```

These kinds of statements can be particularly helpful when trying to output a conditionally determined value to the screen because they can be embedded in a WriteLine. For example, in this program, we determine if a value is even or odd and print it as such.

```
namespace InLineConditional
{
    internal class Program
    {
        static void Main(string[] args)
        {
            int number;
            Console.Write("Enter a number and I will tell you if it is even or odd: ");
            number = int.Parse(Console.ReadLine());
            Console.WriteLine($"The number {number} is {(number % 2 == 0 ? "even" : "odd")}.");
        }
```

```
    }
}
```

A bit of trivia: the parentheses around the in-line conditional are required when using formatted strings (i.e. strings expressed with $"")

## Switch statement

A switch statement is another kind of conditional, but it is one that targets specific values rather than ranges of values. The switch statement tests the values of a *control variable* to see if they are any of multiple discrete values. When a matching value is found, the code that corresponds to that discrete value is performed. Each of the discrete values is called a *case*. The response to each case usually includes a *break* statement at the end to skip out of the switch statement after completing that one matching case. If there are no matching cases, you can include a case called default that executes as a last resort.

Here is an example. Suppose we want to print words or encouragement or derision based on a person's letter grade. Here's the code that could do that.

```
string letter;
Console.Write("Enter your letter grade: ");
letter = Console.ReadLine();
switch (letter)
{
    case "A+": case "A":
        Console.WriteLine("You're pretty smart.");
        break;
    case "B":
        Console.WriteLine("Well-done");
        break;
    case "C":
        Console.WriteLine("C is for cookie, that's good enough for me.");
        break;
    case "D":
        Console.WriteLine("D is for diploma");
        break;
    default:
        Console.WriteLine("F this, amiright?");
        break;
}
```

## Repetition

Programming languages perform tasks repeatedly using loops. There are two kinds of loops:

- sentinel-controlled, which repeat a number of times that can't be predicted ahead of time but instead continue until some condition is no longer met
- counter-controlled, which repeat a number of times that can be predicted ahead of time, either because we know we want to repeat something x number of times, or because we need to perform the same operations on a fixed set of data.

C# provides both kinds of loops.

- Sentinel-controlled loops are implemented as `while` and as `do-while` loops
- Counter-controlled loops are implemented as `for` and `foreach` loops

## for loop

A `for` loop begins with a somewhat complicated looking first line

```
for (declare and initialize the counter;  test the counter; adjust the counter) {
      instructions to repeat as long as the test on the counter is still true;
}
```

For example, here is a program that asks the user to enter 5 numbers. At the end, it reports to them the average of the numbers.

```
int sum = 0;
int num;
for (int i = 0; i < 5; i++)
{
    Console.Write("Enter an integer: ");
    num = int.Parse(Console.ReadLine());
    sum = sum + num;
}
double avg = sum / 5.0;
Console.WriteLine("The average of the 5 numbers is {0:F2}.", avg);
```

## foreach loop

The `foreach` loop is a counter-controlled loop that performs the same tasks on every entry in a list. We haven't studied lists yet, so this might be jumping ahead a bit, but I think this example is easy enough for us to understand. Suppose I create a list of numbers called nums like so:

```
int[] nums = {5, 7, 3, 2, 6, 8};
```

I can then use a foreach loop to visit each number and add them up:

```
int[] nums = { 5, 7, 3, 2, 6, 8 };
int sum = 0;
foreach(int num in nums) {
    sum = sum + num;
}
Console.WriteLine("The sum is {0}.", sum);
```

Notice how the top of the foreach loop declares the counter as being of the same type as the values in the list we are going to use it to traverse. We use the keyword *in* to indicate that num is going to take on the full range of values stored in the list *nums*.

## while loop

The `while` loop is a sentinel-controlled loop. It performs a set of instructions as long as the condition at the top of the while loop is true. As soon as it becomes false, we leave the while loop and continue with the statement immediately after it.

While loops test their condition at the very top, before any statement inside is executed. This means that we must initialize any variables involved in the while loop's test before we arrive at the while loop. It also means that we must adjust the values of those variables inside the loop if we are ever going to be able to escape the loop. Otherwise, if we don't adjust the variables that control the while loop, the condition will always be true, and we'll be stuck in an infinite loop.

Here is the general syntax:

```
declare and initialize the while loop's condition's variables
while (condition is true) {
   statements to do
   adjust the variables on which the condition depends
}
```

## while loop example

In this example, we write a program that keeps asking the person to enter a number until they indicate they wish to stop. It then shows them the average of the numbers they entered.

```
int sum = 0;
int count = 0;
int num;
string another = "y";
while (another == "y")
{
    Console.Write("Enter a number: ");
    num = int.Parse(Console.ReadLine());
    count = count + 1;
    sum = sum + num;
    Console.Write("Do you want to enter another? ");
    another = Console.ReadLine().ToLower();
}
double avg = (double)sum / count;
Console.WriteLine($"The average is {avg,0:F2}.");
```

## do while loop

The `do while` loop is another sentinel-controlled loop. In other words, like the while loop, the do while loop allows us to repeat a set of statements as long as a condition is true. Unlike a `while` loop, which tests its condition at the top, the `do while` loop has no gatekeeper, as it performs its test at the bottom of the loop, after the statements inside it have been performed at least once. Sometimes the `do while` loop proves more convenient than the while loop because it does not require you to initialize the loop's condition's variables prior to executing its statements. This is particularly useful when you know that the statements should be performed at least once.

As an example, rewrite the previous example, but this time using a do while loop instead of a while loop.

```
int sum = 0;
int count = 0;
int num;
string another;
do
{
    Console.Write("Enter a number: ");
    num = int.Parse(Console.ReadLine());
    count = count + 1;
    sum = sum + num;
    Console.Write("Do you want to enter another? ");
    another = Console.ReadLine().ToLower();
} while (another == "y");
double avg = (double)sum / count;
Console.WriteLine($"The average is {avg,0:F2}.");
```

## Jump statements

It is possible to break out of a for loop completely using *break*, or to resume the for loop at the top of its next iteration (and skip the rest of the current iteration) using *continue*. Just include *break;* as an instruction if you need to leave a for loop prematurely for some reason, and include *continue;* if you want to resume the for loop at the beginning of its next iteration.

For example, here we skip the rest of a for loop if the counter is 7. Specifically, this code prints the values of the counter 0 through 9 except for the number 7.

```
static void Main(string[] args)
{
   for (int i = 0; i < 10; i++)
    {
        if (i == 7)
        {
            continue;
        }
        Console.WriteLine(i);
    }
}
```

## Exception Handling

An exception is a problem that occurs as a program runs that, if not handled, will cause the program to crash. To enable the program to respond more gracefully to an exception than just having the program crash in a flurry of ugly error messages posted to the terminal window, we can use `try .. catch` blocks.

A `try catch` block consists of two parts:

- try, in which we *try* to do some code that could raise an exception
- catch, in which we *catch* the exception that did occur so that we can handle it gracefully.

Optionally, there can be a third part called *finally* that contains code that must be done regardless of whether an exception happened or not. Often you'll put required cleanup code (such as closing a file or network connection) inside the finally part.

Here is the syntax of try..catch..finally:

```
try {
    code to try that could raise an exception
} catch (Exception ex) {
    handle the exception, perhaps by displaying ex.Message
} finally {
    what to do always, regardless of whether an exception happened.
}
```

You could actually trap and respond specifically to particular kinds of exceptions. If you do this, you must arrange the catch statements such that the least specific type of excpetion – just plain ol' Exception – is caught last, like so:

```
try {
} catch (IndexOutOfRangeException) {

} catch (FormatException) {

} catch (Exception ex) {
```

```
} finally {

}
```

In this example, we ask the user to enter a number and tell them if they entered an odd or an even number. If they enter something that isn't an integer, we will display an error message.

```
int num;
try
{
    Console.Write("Enter an integer: ");
    num = int.Parse(Console.ReadLine());
    if (num % 2 == 0)
    {
        Console.WriteLine("Even");
    } else
    {
        Console.WriteLine("Odd");
    }
} catch (Exception ex)
{
    Console.WriteLine("You didn't enter a valid integer.");
    Console.WriteLine("Here is the error message: ");
    Console.WriteLine(ex.Message);
} finally
{
    Console.WriteLine("Have a nice life.");
}
```

## Example:

In this example, we will create a tool for calculating the total payroll for a company. You will ask the user to enter the hours worked and hourly pay rate for each employee. A sliding tax scale will be used to determine the tax rate for each person. As each person's paycheck is processed, we will print their gross pay, net pay, and taxes to the screen. After processing an employee, you will ask the user if they have another employee to process. At the end of the program, we will show the total amount paid to the employees and the average pay per employee.

```
using System;

namespace Payroll20220507
{
    internal class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Welcome to Payroll");
            double hours, rate, gross, net, taxes, taxRate;
            string doAgain = "y";
            do
            {
                try
                {
                    Console.Write("Enter hours worked: ");
                    hours = double.Parse(Console.ReadLine());
                    Console.Write("Enter hourly pay rate: ");
                    rate = double.Parse(Console.ReadLine());
                    gross = hours * rate;
```

```
                    if (gross >= 2000)
                    {
                        taxRate = 0.2;
                    }
                    else if (gross >= 1000)
                    {
                        taxRate = 0.15;
                    }
                    else if (gross >= 500)
                    {
                        taxRate = 0.1;
                    }
                    else
                    {
                        taxRate = 0;
                    }
                    taxes = taxRate * gross;
                    net = gross - taxes;
                    Console.WriteLine("Gross Pay: {0,15:F2}", gross);
                    Console.WriteLine("Taxes:     {0,15:F2}", taxes);
                    Console.WriteLine("Net Pay:   {0,15:F2}", net);
                    Console.Write("Do you have another employee? ");
                    doAgain = Console.ReadLine().ToLower();
                }
                catch (Exception ex)
                {
                    Console.WriteLine("You entered invalid values.");
                }
            } while (doAgain == “y”);
            Console.WriteLine("Thank you for using this program.");
        }
    }
}
```

## Another Example:

Create a C# solution called HousePainting. The program will enable the user to specify the dimensions of the rooms of the house they want to paint. For each room, the program will ask the user for the length, width, and height of the room, whether or not they want to use primer, and whether or not they want to paint the ceiling. At the end, the program will show them the total square footage they will be painting and the total number of gallons of paint and primer required. You can assume that paint covers 400 square feet per gallon and primer covers 250 square feet per gallon.

Here's how your program should behave:

```
****************************************
          HOUSE PAINTER V1.0
****************************************

How many rooms do you plan to paint? 4
For Room #1 ...
        Enter length: 12.5
        Enter width: 13.6
        Enter height: 8
        Paint the ceiling? y
        Use primer? y
For Room #2 ...
        Enter length: 10.5
        Enter width: 11.5
        Enter height: 8
        Paint the ceiling? n
        Use primer? n
For Room #3 ...
        Enter length: 14.6
        Enter width: 16.5
        Enter height: 10
        Paint the ceiling? y
        Use primer? y
For Room #4 ...
        Enter length: 10
        Enter width: 10.2
        Enter height: 8
        Paint the ceiling? n
        Use primer? y

To paint 2125.70 square feet,
        you need 5.31 gallons of paint.
To prime 1773.70 square feet,
        you need 7.09 gallons of primer.

Thank you for using this program.
```