Module 02(Making decisions and performing Iterations)

Introduction

C# decision structures provide logic in your application code that allows the execution of different sections of code depending on the state of data in the application. You might ask users whether they wish to save any changes to a file that is open in the application. The decision structure permits you to code behavior to execute based on the answer provided by the user. Visual C# uses conditional statements to achieve this functionality.

The primary conditional statement in Visual C# is the *if* statement. An alternative to the *if* statement is a *switch* statement. As you will see in the section on the *switch* statement, you might want to use it for more complex decisions.

C# if Statements

In C#, if statements are concerned with Boolean logic. If the statement is **true**, the block of code associated with the if statement is executed. If the statement is **false**, control either falls through to the line after the if statement, or after the closing curly brace of an if statement block.

The following code sample demonstrates an if statement to determine if a response contains a value of **yes**.

if statement

```
string response = "Yes";
if (response == "Yes")
{
   // statements that will execute if the value of the response variable is
   // yes, will be placed here.
}
```

Note the use of curly braces in the code sample. You can eliminate the curly braces if your statement to execute is a single line statement. C# understands that if no curly braces are used, the line immediately after the if(condition) will be executed if the condition is **true**. Otherwise, it is not. To avoid confusion as to which lines will execute for a true condition, a recommended practice is to always use curly braces for your if statement.

In C#, if statements can also have associated else clauses. The else clause executes when the ifstatement is **false**.

The following code example shows how to use an if else statement to execute code when a condition is **false**.

if else Statements

```
string response;
if (response == "connection_failed")
{
   // Block of code executes if the value of the response variable is "connection_failed".
}
else
{
   // Block of code executes if the value of the response variable is not "connection_failed".
}
```

if statements can also have associated else if clauses. The clauses are tested in the order that they appear in the code after the if statement. If any of the clauses returns **true**, the block of code associated with that statement is executed and control leaves the block of code associated with the entire if construct.

The following code example shows how to use an if statement with an else if clause.

else if Statements

```
string response;
if (response == "connection_failed")
{
   // Block of code executes if the value of the response variable is "connection_failed".
}
else if (response == "connection_error")
{
   // Block of code executes if the value of the response variable is "connection_error".
}
else
{
   // Block of code executes if the value of the response variable is neither above responses.
}
```

You can create as many else if blocks as necessary for your logic, or until you become completely lost from too many else if clauses. If you require any more than five else if clauses, you might want to consider the switch statement, presented next.

The switch statement

If there are too many else if statements, code can become messy and difficult to follow. In this scenario, a better solution is to use a switch statement. The switch statement simply replaces multiple else ifstatements.

The following sample shows how you can use a switch statement to replace a collection of else if clauses.

switch Statement

```
string response;
switch (response)
{
  case "connection_failed":
    // Block of code executes if the value of response is "connection_failed".
    break;
  case "connection_success":
    // Block of code executes if the value of response is "connection_success".
    break;
  case "connection_error":
    // Block of code executes if the value of response is "connection_error".
    break;
  default:
    // Block executes if none of the above conditions are met.
    break;
}
```

Notice that there is a block labeled default:. This block of code will execute when none of the other blocks match.

In each case statement, notice the break keyword. This causes control to jump to the end of the switch after processing the block of code. If you omit the break keyword, your code will not compile. If you want to handle multiple cases with the same code segment, you can use a fall-through setup similar to this sample code.

```
string response;
switch (response)
{
  case "connection_success":
    // Block of code executes if the value of response is "connection_success".
    break;
  case "connection_failed":
  case "connection_error":
    // Block of code executes if the value of response is "connection_failed"
    // or "connection_error;
    break;
  default:
    // Block executes if none of the above conditions are met.
    break;
}
```

If you are coming from another programming language, such as C, that also uses the switch statement, you might notice that in the C# language, you can use string values in your switch statements and don't have to use integers or enumerated types. C# switch statements support the following data types as expressions:

- sbyte
- byte
- short
- ushort
- int
- uint
- long
- ulong
- char
- string
- enumerations

Introducing Repetition

Visual C# provides a number of standard constructs known as loops that you can use to implement iteration logic. If you are coming from other programming languages, you might recognize for loops, while loops, and do loops. C# supports all three of these iteration statements.

The for Loops

The for loop executes a block of code repeatedly until the specified expression evaluates to false. You can define a for loop as follows.

```
for ([initializers]; [condition]; [iterator])
{
  // code to repeat goes here
}
```

The [initializers] portion is used to initialize a value as a counter for the loop. On each iteration, the loop checks that the value of the counter is within the range to execute the for loop, specified in the [condition]portion., and if so, execute the body of the loop. At then end of each loop iteration, the [iterator] section is responsible for incrementing the loop counter.

The following code example shows how to use a for loop to execute a code block 10 times.

for Loop

```
for (int i = 0 ; i < 10; i++)
{
   // Code to execute.
}
```

In this example, i = 0; is the initializer, i < 10; is the condition, and i++ is the iterator.

For Each Loops

While a for loop is easy to use, it can present some challenges depending on the situation. As an example, consider iterating over a collection or an array of values. You would need to know how many elements are in the collection or array. In many cases you will know this, but sometimes you may have collections or arrays that are dynamic and are not sized at compile-time. If the size of the collection or array changes during runtime, it might be better option to use a foreach loop.

The following code example shows how to use a foreach loop to iterate a string array.

foreach Loop

```
string[] names = new string[10];
// Process each name in the array.
foreach (string name in names)
{
   // Code to execute.
}
```

C# handles determining how many items are in the array and will stop executing the loop when the end is reached. The use of foreach loops can help prevent index out of bounds errors on arrays.

The while Loop

A while loop enables you to execute a block of code while a given condition is **true**. For example, you can use a while loop to process user input until the user indicates that they have no more data to enter. The loop can continue to prompt the user until they decide to end the interaction by entering a sentinel value. The sentinel value is responsible for ending the loop.

The following code example shows how to use a while loop.

while Loop

```
string response = PromptUser();
while (response != "Quit")
{
   // Process the data.
   response = PromptUser();
}
```

It's imperative to include the response = PromptUser(); inside the loop braces. Failure to put this into the loop body will result in an infinite loop because the sentinel value can never be changed.

The do Loop

A do loop, sometimes also referred to as a do...while loop, is very similar to a while loop, with the exception that a do loop will always execute the body of the loop at least once. In a while loop, if the condition is false from the start, the body of the loop will never execute.

You might want to use a do loop if you know that the code will only execute in response to a user prompt for data. In this scenario, you know that the application will need to process at least one piece of data, and can therefore use a do loop.

The following code example shows the use of a do loop.

do Loop

```
do
{
   // Process the data.
   response = PromptUser();
} while (response != "Quit");
```

Module Two Assignment

For this assignment, you will create the pattern of a chess board that is 8 x 8. Use X and O to represent the squares.

1. Create the appropriate nested looping structure to output the characters in an 8 x 8 grid on the screen using Console.Write() or Console.WriteLine() as appropriate.
2. Include a decision structure to ensure that alternate rows start with opposite characters as a real chess board alternates the colors among rows.

   This is what your output should look like.

```
XOXOXOXO
OXOXOXOX
XOXOXOXO
OXOXOXOX
XOXOXOXO
OXOXOXOX
XOXOXOXO
OXOXOXOX
```

Grading Criteria:

- Used a nested loop
- Used a decision structure to flip row output
- Output is correct per above image