

عنوان مضمون

Visual Programming-I

توسط : صفری

خزان 1397

C# decision making

Flow Control

- All the C# code you've seen the program execution has proceeded from one line to the next in top-to-bottom order, missing nothing.
- This chapter describes two methods for controlling program flow that is, the order of execution of lines of C# code:
 1. *Branching:*

Branching executes code conditionally, depending on the outcome of an evaluation, such as "Execute this code only if the variable myVal is less than 10."
 2. *Looping:*

Looping repeatedly executes the same statements, either a certain number of times or until a test condition has been reached.

Both techniques involve the use of ***Boolean logic***.

BOOLEAN LOGIC

- The **bool** type can hold one of only two values: **true** or **false**.
- In particular, bool types are used to store the result of a *comparison*.

OPERATOR	CATEGORY	EXAMPLE EXPRESSION	RESULT
<code>==</code>	Binary	<code>var1 = var2 == var3;</code>	var1 is assigned the value true if var2 is equal to var3, or false otherwise.
<code>!=</code>	Binary	<code>var1 = var2 != var3;</code>	var1 is assigned the value true if var2 is not equal to var3, or false otherwise.
<code><</code>	Binary	<code>var1 = var2 < var3;</code>	var1 is assigned the value true if var2 is less than var3, or false otherwise.
<code>></code>	Binary	<code>var1 = var2 > var3;</code>	var1 is assigned the value true if var2 is greater than var3, or false otherwise.
<code><=</code>	Binary	<code>var1 = var2 <= var3;</code>	var1 is assigned the value true if var2 is less than or equal to var3, or false otherwise.
<code>>=</code>	Binary	<code>var1 = var2 >= var3;</code>	var1 is assigned the value true if var2 is greater than or equal to var3, or false otherwise.

BOOLEAN LOGIC

- You might use operators such as these on numeric values in code:

```
bool isLessThan10;
```

```
isLessThan10 = myVal < 10;
```

- The **&** and **|** operators also have two similar operators, known as ***conditional Boolean operators***

OPERATOR	CATEGORY	EXAMPLE EXPRESSION	RESULT
&&	Binary	var1 = var2 && var3;	var1 is assigned the value true if var2 and var3 are both true, or false otherwise. (Logical AND)
	Binary	var1 = var2 var3;	var1 is assigned the value true if either var2 or var3 (or both) is true, or false otherwise. (Logical OR)

Boolean Bitwise and Assignment Operators

- **Boolean comparisons** can be combined with **assignments** by combining **Boolean bitwise and assignment operators**.
- When expressions use both the assignment (=) and bitwise operators (&, |, and ^), the binary representation of the compared quantities are used to compute the outcome, instead of the integer, string, or similar values.

OPERATOR	CATEGORY	EXAMPLE EXPRESSION	RESULT
&=	Binary	<code>var1 &= var2;</code>	var1 is assigned the value that is the result of <code>var1 & var2</code> .
=	Binary	<code>var1 = var2;</code>	var1 is assigned the value that is the result of <code>var1 var2</code> .
^=	Binary	<code>var1 ^= var2;</code>	var1 is assigned the value that is the result of <code>var1 ^ var2</code> .

- the equation `var1 ^= var2` is similar to `var1 = var1 ^ var2`.

```
using static System.Console;
using static System.Convert;
namespace ConsoleApplication6
{
    class Program
    {
        static void Main(string[] args)
        {
            WriteLine("Enter an integer:");
            int myInt =.ToInt32(ReadLine());
            bool isLessThan10 = myInt < 10;
            bool isBetween0And5 = (0 <= myInt) && (myInt <= 5);
            WriteLine($"Integer less than 10? {isLessThan10}");
            WriteLine($"Integer between 0 and 5? {isBetween0And5}");
            WriteLine($"Exactly one of the above is true? " +
    $"{isLessThan10 ^ isBetween0And5}");
            ReadKey();
        }
    }
}
```

BRANCHING

- Branching is the act of controlling which line of code should be executed next.
- This section describes three branching techniques available in C#:
 - The ternary operator
 - The if statement
 - The switch statement

BRANCHING

- The Ternary Operator:

You've already seen unary operators that work on one operand, and binary operators that work on two operands, so it won't come as a surprise that this operator works on three operands. The syntax is as follows:

<test> ? <resultIfTrue>: <resultIfFalse>

- You might use this as follows to test the value of an int variable called myInteger:

```
string resultString = (myInteger < 10) ? "Less than 10" : "Greater than or  
equal to 10";
```

BRANCHING

- The if Statement:

The simplest use of an if statement is as follows, where *<test>* is evaluated (it must evaluate to a Boolean value for the code to compile) and the line of code that follows the statement is executed if *<test>* evaluates to true:

```
if (<test>)  
    <code executed if <test> is true>;
```

- You can also specify additional code using the else statement in combination with an if statement.

```
if (<test>)  
    <code executed if <test> is true>;  
else  
    <code executed if <test> is false>;
```

BRANCHING

- Because the result of the if statement cannot be assigned to a variable, you have to assign a value to the variable in a separate step:

```
string resultString;  
if (myInteger < 10)  
    resultString = "Less than 10";  
else  
    resultString = "Greater than or equal to 10";
```

BRANCHING

Checking More Conditions Using if Statements:

```
if (var1 == 1)
{
    // Do something.
}
else
{
    if (var1 == 2)
    {
        // Do something else.
    }
    else
    {
        if (var1 == 3 || var1 == 4)
        {
            // Do something else.
        }
        else
        {
            // Do something else.
        }
    }
}
```

BRANCHING

The switch Statement:

This test is limited to discrete values, rather than clauses such as “greater than X,” so its use is slightly different; however, it can be a powerful technique.

- The basic structure of a switch statement is as follows:

```
switch (<testVar>)  
{  
    case <comparisonVal1>:  
        <code to execute if <testVar> == <comparisonVal1> >  
        break;  
    case <comparisonVal2>:  
        <code to execute if <testVar> == <comparisonVal2> >  
        break;  
    ...  
    default:  
        <code to execute if <testVar> != comparisonVals>  
        break;  
}
```

```
static void Main(string[] args)
{
    const string myName = "benjamin";
    const string niceName = "andrea";
    const string sillyName = "ploppy";
    string name;
    WriteLine("What is your name?");
    name = ReadLine();
    switch (name.ToLower())
    {
        case myName:
            WriteLine("You have the same name as me!");
            break;
        case niceName:
            WriteLine("My, what a nice name you have!");
            break;
        case sillyName:
            WriteLine("That's a very silly name.");
            break;
    }
    WriteLine($"Hello {name}!");
    ReadKey();
}
```

LOOPING

- *Looping* refers to the repeated execution of statements.
- do Loops:
The structure of a do loop is as follows, where <Test> evaluates to a Boolean value:

```
do  
{    <code to be looped>  
} while (<Test>;
```

LOOPING

- For example, you could use the following to write the numbers . . .

```
int i = 1;  
do  
{    WriteLine("{0}", i++);  
} while (i <= 10);
```


LOOPING

- while Loops:

while loops are very similar to do loops, but they have one important difference: The Boolean test in a while loop takes place at the start of the loop cycle, not at the end.

Here's how while loops are specified:

```
while (<Test>)  
{  
    <code to be looped>  
}
```

They can be used in almost the same way as do loops:

```
int i = 1;  
while (i <= 10)  
{  
    WriteLine($"{i++}");  
}
```

LOOPING

- for Loops:

This type of loop executes a set number of times and maintains its own counter. To define a for loop you need the following information:

- A starting value to initialize the counter variable
- A condition for continuing the loop, involving the counter variable
- An operation to perform on the counter variable at the end of each loop cycle

- This information must be placed into the structure of a for loop as follows:

```
for (<initialization>; <condition>; <operation>)  
{  
    <code to loop>  
}
```

Interrupting Loops

- Sometimes you want finer-grained control over the processing of looping code. C# provides commands to help you here:
 - `break`—Causes the loop to end immediately
 - `continue`—Causes the current loop cycle to end immediately (execution continues with the next loop cycle)
 - `return`—Jumps out of the loop and its containing function

Interrupting Loops

- **The `break`** command simply exits the loop, and execution continues at the first line of code after the loop, as shown in the following example:

```
int i = 1;
while (i <= 10)
{
    if (i == 6)
        break;
    WriteLine($"{i++}");
}
```

- This code writes out the numbers from 1 to 5 because the `break` command causes the loop to exit when `i` reaches 6.

Interrupting Loops

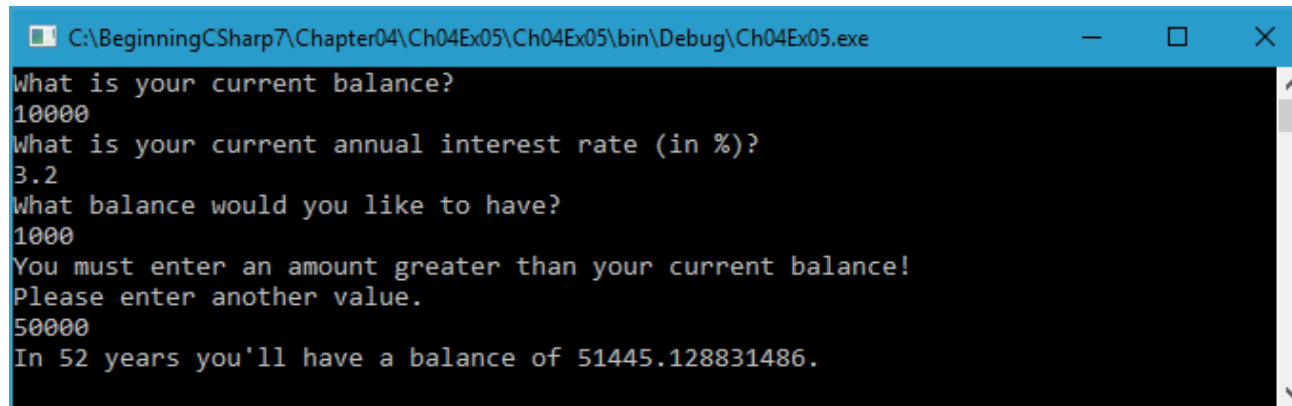
- **continue** only stops the current cycle, not the whole loop, as shown here:

```
int i;  
for (i = 1; i <= 10; i++)  
{    if ((i % 2) == 0)  
        continue;  
    WriteLine(i);  
}
```

- In the preceding example, whenever the remainder of *i* divided by 2 is zero, the `continue` statement stops the execution of the current cycle, so only the numbers 1, 3, 5, 7, and 9 are displayed.

homework

- you use a loop to calculate how many years it will take to get a specified amount of money in the account, based on a starting amount and a fixed interest rate.(in winform)



```
C:\BeginningCSharp7\Chapter04\Ch04Ex05\Ch04Ex05\bin\Debug\Ch04Ex05.exe
What is your current balance?
10000
What is your current annual interest rate (in %)?
3.2
What balance would you like to have?
1000
You must enter an amount greater than your current balance!
Please enter another value.
50000
In 52 years you'll have a balance of 51445.128831486.
```