CHAPTER 1

BASIC Programming Concepts

Programming is all about getting the computer to do what you want it to do. The key is knowing how to tell the computer in a way it will understand. That's where programming languages come in. There are many different programming languages that are designed to make the communication easier in different situations.

In this chapter you will learn about the BASIC programming language, how it 's different in REALbasic and the fundamentals of programming.

Contents

- Executing Instructions Repeatedly with Loops
- Decision Making

BASIC versus REALbasic

The language BASIC was created in the 1960's for the purpose of teaching people programming. Most of what made other languages difficult to master, was removed from BASIC to make learning it easier. In fact, BASIC is an acronym which stands for Beginners All-Purpose Symbolic Instruction Code.

BASIC for a long time was considered less powerful than other languages but this was mostly due to the way the it was implemented rather than the language itself. Spoken languages wouldn't be considered to be very powerful if you could only speak one word every 10 minutes for example. Computers actually only understand two things, 1 and 0. That's it. that's all the know. The rest of what a computer does all breaks down to that fundamental concept. These 1's and 0's that computers understand are referred to as Machine Language. Most versions of BASIC have used an interpreter program to execute the code. This means that each time a program ran, the BASIC interpreter had to turn the BASIC code into Machine Language. Other languages had compilers which are special programs that translate the programing language into Machine Language all at once which makes them execute faster because the constant interpretation is removed.

REALbasic has a compiler built-in to it. That means your code is always running as fast as possible. BASIC is a traditional programming language that starts with the first line programming code and continues until the last line. REALbasic is a modern, object-

oriented version of BASIC. If you are new to programming that might not mean much now but it will. REALbasic takes the simplicity of the BASIC language and adds the power of modern programming through it's object-oriented implementation and compiler. Also, most programming languages require you to know quite a bit about how to communicate with the computer's operating system. REALbasic abstracts you from all of that making it easier for you to learn and easier to run your application on computers running operating systems that are different from the one you created your application on.

Storing Values in Properties and Variables

When you need to store information so you can access it again even after you have shut off your computer, you tell you computer to store the information in a document. When a computer needs to store information temporarily, it's stored in the computer's memory. The computer's memory is like a series of organized boxes. Each box has a location in memory with an address that is used to locate it. These locations are given names to make them easier to work with. Depending on how these memory locations are used, they are called Variables and Properties.

What are Properties?

The values that make up the description of an object like a window are called Properties. The title of a window is a property. The width of the window is a property. When a window is opened, these properties are copied into memory. You can access them using their names. You can get values from them and you can store new values in them. For example, if you wanted the title of

a window to change when the user clicks a button, you would set the title property of the window to the new value. Each property can hold a certain type of data. Some properties store text (like a window title) while others store numbers (like the window's width property). Later in this chapter, you will learn how to assign values to properties and how to get the values that are stored in properties.

Variables

Sometimes you will need to store a value that isn't related directly to an object like a window or a button. In this case you use a variable. A variable is just like a property but it isn't directly related to any particular object. Later in this chapter, you will learn how to create variables, assign values to them and get values from them.

Data Types

To make programming code execute faster and to provide powerful commands that save you time when programming, computers have to be able to make certain assumptions about the information you give them. For example, when you give a computer a piece of information, the computer needs to know if it's a number, a string of characters, a date, etc. If you didn't tell the computer what kind of data you are giving it, it wouldn't know whether you meant 1 plus 1 to be 2 or 11. In this example, telling the computer that you are giving it numbers will result in 2. Telling it you are giving it simply a string of typed characters will result in 11. There are many data types that REALbasic understand but there are five data types that are by far the most common and they are String, Integer, Single, Double and Boolean.

String

A string is just series (or string) of characters. Basically any kind of information can be stored as a string. "Jannice", "3/17/98", "45.90" are all examples of strings. You might be thinking "Hey, those last two don't look like strings" but they are. When you place quotes around information in your code, you are telling REALbasic to look at the data as just a string of characters and nothing more. The maximum length of a string is based only on available memory.

You can concatenate two strings together by adding them together with the addition symbol (+). For example "Big" + "Dog" would result in the string "BigDog". That is really the extent of the mathematics you can perform on strings. However, REALbasic has many built-in functions that make processing strings easy.

Integer

An Integer is a whole number between approximately -2 billion and +2 billion. In other programming languages, REALbasic's Integer type is called a Long Integer or just a Long. Because integers are numbers, you can perform mathematical calculations on them. Unlike strings, integers do not have quotes around them in your code. An Integer value uses 4 bytes of memory.

Single

A Single is a number that can contain a decimal value. There is no practical limit as their is with Integer. In other languages, REALbasic's Single may be referred to as a single precision real number. Because Singles are numbers, you can perform mathematical calculations on them. Single numbers use 4 bytes of memory.

Double

A Double is a number that can contain decimal value. Unlike Integers, Doubles have no limit to the range of numbers they can hold. In other languages, REALbasic's Double may be referred to as a double precision real number. Because Doubles are numbers, you can perform mathematical calculations on them. Doubles use 8 bytes of memory. The PowerPC microprocessor converts Singles to Doubles before performing calculations on them so you are probably better off using a Double instead of a Single.

Boolean

Boolean means true or false. Boolean values are false by default but can be set to true using REALbasic's True function and back to false using the False function. Some of the properties of objects in REALbasic are boolean values. For example, most of the controls have an Enabled property that is boolean.

Other Data Types

There are many other data types. You will learn about these in the next chapter.

Changing a Value From One Data Type to Another

There may be times when you need to change a value from one data type to another. This is usually because you want to use the value with something that is designed to work with a different data type. For example, you might want to include a number in the title of a window. The title of a window is a string, not a number. Consequently, if you try to assign a number to the title of a window, REALbasic will display an error message when you run your application. The error will tell you that the two data types are not compatible (they are different). Since the window

title is a string, you will need to change the number into a string before you can assign it to the window title.

Fortunately, REALbasic has a built-in function called Str (which stands for String) that can change a number into a string. See "Str Function" on page 101 of the Language Reference for more information. There is also a built-in function called Val (which is short for Value) that changes strings into numbers. See "Val Function" on page 113 of the Language Reference for more information.

Assigning Values to Properties

The basic syntax for assigning a value is:

```
objectName.propertyName=value
```

For example, if you have a pushbutton called pushbutton1, and you want to set it's caption property to "OK", you would use the following code:

```
pushbutton1.caption="OK"
```

You can read this as *change pushbutton1's caption property to "OK".* This syntax is used when you want a control in a window to change a property of a control in the same window. If you want a control to change a property of a control in another open window, you must include the target window's name (not title) in the syntax. For example, say you have two open windows whose names are window1 and window2 respectively. You want a push-button on window1 to set the value of pushbutton1's caption on window2 to "OK". The syntax would look like this:

```
window2.pushbutton1.caption="OK"
```

If you didn't specify the window, REALbasic would implicitly assume you meant the control called pushbutton1 in the window that contains the object executing the code. If you specify a window that is not open, REALbasic will open the window then make the change. If you have more than one copy of the window open that contains the control you are trying to change, this syntax won't work because you won't be able to tell REALbasic which copy of the window you are referring to. You will learn how to deal with this issue in the next chapter.

If a control is going to change a property of it's own window, the window name is not required. The window name is implicit. For example, if you wanted a pushbutton when clicked to change it's window's title property to "Hello World", you would use this syntax:

```
title="Hello World"
```

Getting Values From Properties

You can get a value from a property in almost the same way you store values in properties. The only different is that the target of the value (where you want the value of the property stored) goes on the left side of the equals sign and the object and property names go on the right. For example, if you had a variable named X and you wanted to assign pushbutton1's caption to it, the syntax would be:

```
x=pushbutton1.caption
```

And just as in setting properties, you can get the property of a control in another window by including the window's name. For example, if you want to assign the variable x to window2's pushbutton1 caption property, you would use this syntax:

```
x=window2.pushbutton1.caption
```

And just like setting properties, if you include only the property name, REALbasic assumes you are referring to a property of the window that contains the control that is executing the code. For example, if you have a pushbutton called pushbutton1 and you want it to assign the window title to the variable x when it is clicked, you would use this syntax:

```
x=title
```

Getting and Setting Values in Variables

When you need to store a value that is not associated with an object (the way a property is associated with a control or window), you use a variable. A variable is nothing more than a location in memory to store a value. Variables have names just like properties do. The name you give a variable should describe the purpose of the variable. Say you wanted to calculate the number of days old a person is based on the year they were born. You might have a variable called "Days" to keep track of that information. Variable names can be any length but must begin with a letter and can contain only alphanumeric characters (A-Z, a-z, 0-9). Variable names are case-insensitive so REALbasic sees x and X as the same variable.

You can put values in variables and get values from variables in the same way you do with properties. To get a value from a variable, it must be on the right side of the assignment operator (=). Say for example, you wanted to set the caption of a pushbutton to the value in a variable called "buttonTitle". The example below accomplishes that:

```
Pushbutton1.Caption=buttonTitle
```

Conversely, if you wanted to store the value of a property (like the pushbutton's caption in the last example) in a variable, you would simply reverse the syntax:

```
buttontitle=pushButton1.Caption
```

Like properties, variables have data types. Before you can use a variable, it's data type must be made known using the **Dim** statement. Dim is short for *Dimension* which means to make space for the variable. In the example below, the variable i is dimensioned (or dimed) as an integer:

```
Dim i as integer
```

If you have several variables of the same type, you can declare them all with one Dim statement:

```
Dim i,j,k as integer
```

You already know about the data types Integer, String, Boolean, Single and Double. But variables can also be declared as specific object types. For example, REALbasic has an object type called a *FolderItem*. A FolderItem can represent any item that can exist in a folder on the desktop (file, application or another folder). To store a FolderItem object, you must first declare a variable of type FolderItem as in this example:

```
Dim f as FolderItem
```

In this case, f is now an object with properties. One of the properties of a FolderItem is it's name which is the name of the file, application or folder that the FolderItem represents. The variable f's name property could then be assigned to say, variable n like this:

```
n=f.name
```

The Dim statement creates the variable but when does the variable get erased from memory? You will find out the answer to that question in the next chapter.

Just like properties, you can only assign values to variables that are compatible with the variable's data type. The last line of the following example generates an error because the types don't match:

```
Dim x as integer
Dim y as string
x=1
y="Hello"
z=x+y
```

In the example above x is a number and y is a string. An error is generated because you can't add different data types together.

Mathematical Operators

Performing mathematical calculations is a very common task in programming. REALbasic supports all of the common mathematical operations.

Operation Performed	Operator	Example
Addition	+	2 + 3 = 5
Subtraction	-	3 - 2 = 1
Multiplication	*	3 * 2 = 6
Floating Point Division	/	6 / 4 = 1.5
Integer Division	\	6 \ 4 = 1
Modulo	Mod	6 Mod 3 = 0
		6 Mod 4 = 2

There are also many built-in mathematical functions. See the Language Reference for more information.

REALbasic supports standard mathematical precedence. This means that equations surrounded by parens are handled first. REALbasic will begin with the set of parens that is embedded inside the most other sets of parens. Next any addition or subtraction from left to right is performed. Finally any multiplication or division is performed. In the example below, the three equations return different results because of the placement of parens:

Equation	Result
2+3*(5*3)	75
2+(3*(5*3))	47
2+(3*5)*3	51

Reserved Words

The following words should not be used as variable names because they are used as part of the REALbasic language itself:

Executing Instructions with Methods

A method is simply one or more instructions that are performed for the purpose of accomplishing a specific task. REALbasic has many built-in methods. For example, the **Quit** method will cause your application to exit back to the Finder. Some object types (classes) have built-in methods. For example, the ListBox class has a method called AddRow for adding rows to a ListBox (as the name implies). You can also create your own custom methods. Just like variables, methods are given names to describe them

and the same rules apply: the name can be any length, but must start with a letter and can contain only alphanumeric values (a-z, A-Z, 0-9).

Below is an example of a simple method that calculates how many days old a person is in 1998 who was born in 1960:

```
Dim yearBorn, thisYear, daysOld as Integer
yearBorn=1960
thisYear=1998
daysOld=(thisYear-yearBorn)*365
```

Methods can of course be far more complex and longer than this example. There are three different places you can put your code. You will learn about these in the next chapter.

Documenting (Commenting) Your Code

Documenting your code is important because while it might make sense at the time you write it, it may not make sense days or weeks later. Also, if someone else has to understand your methods, documentation will make their job a whole lot easier. Comments can be added to your code as separate lines or to the right of any code on an existing line. Comments are ignored by REALbasic when it runs your application and have no impact on performance. In order for REALbasic to ignore your comments, you must start the comment with a hyphen (‘), two forward slashes (//) or the word REM (short for reminder). The example below shows how the previous example could be commented:

```
//Create the necessary variables
Dim yearBorn, thisYear, daysOld as Integer
yearBorn=1960  //set the year they were born
```

```
thisYear=1998  //store the current year
//Now calculate the number of days old
daysOld=(thisYear-yearBorn)*365
```

Comments in your code will automatically appear in red.

Passing Values to Methods

Some of REALbasic's built-in methods require one or more pieces of information to perform their function. These pieces of information are called *parameters*. Parameters are passed to a method placing them to the right of the method name in your code. In the example below, the AddRow method of a listBox called ListBox1 is being called. AddRow requires one parameter which is the text that should be displayed in the new row:

```
ListBox1.AddRow "January"
```

If a method requires more than one parameter, commas are used to separate them. The ListBox class has a method called InsertRow which is used to insert new rows into a ListBox at any position. The InsertRow method requires two values: the row number where the new row should appear and the text value that should be displayed in the new row. Because more than one parameter is required, the parameters are separated by commas:

```
ListBox1.InsertRow 3, "January"
```

Parameters can also be variables. If a variable is passed as a parameter, it is the current value of the variable that is passed. In the example below, a variable is assigned a value then passed as a parameter:

```
Month="January"
```

```
ListBox1.InsertRow 3, Month
```

In the next chapter, you will learn how to define parameters for your own custom methods.

Returning Values from Methods

Some methods return values. This means that a value is passed back from the method to the line of code that called the method. For example, REALbasic's built-in method, Ticks, returns the number of ticks (60th's of a second) that have passed since you turned on your computer. You can assign the value returned by a method the same way you assign a value. In the example below, the value returned by Ticks is assigned to the variable x:

```
x=Ticks
```

Some methods require parameters and return a value. For example, the Chr method returns the character whose ASCII code is passed to it. When you pass parameters to a method that returns a value, the parameters must be enclosed in parens. In the example below, the Chr method is passed 13 (the ASCII code for a carriage return) and returns a carriage return to the variable x:

```
x=Chr(13)
```

The parens are required because the value returned might be passed as a parameter to yet another method. Without the parens, it would be difficult to distinguish which parameters where being passed to the which method. In the example below, the numeric value returned by the Len method (which returns the number of characters in the string passed to it) is then passed to the Str method (which converts a numeric value to a string). The string returned by the Str method is then passed as a parameter to the InsertRow method of a Listbox:

```
ListBox1.InsertRow 3, Str(Len("Hello"))
```

Methods that return a value are also referred to as *functions*. In the REALbasic Language Reference, the names of methods that return a value are followed by the word *function*. In the next chapter, you will learn how to return values from your own custom functions.

Comparison Operators

There are many times when you need to compare two values to determine whether or not a particular condition exists. When making a comparison, what you are really doing is making a statement that will either be true or false. For example, the statement "My dog is a cat" evaluates to false. However, the statement "My dog weighs more than my cat" may evaluate to true. The table below shows examples of the comparison operators that are available:

Description	Symbol	Numeric Example	Evaluates To
Equality	=	5=5	True
Inequality	<>	5<>5	False
Greater Than	>	6>5	True
Less Than	<	6<5	False
Greater Than or Equal To	>=	6>=5	True
Less Than or Equal To	<=	6<=5	True

String and boolean values can also be used for comparisons. String comparisons are case insensitive and alphabetical. This means that "Jannice" and "jannice" are equal. But "Jannice" is less than "Jason" because "Jannice" falls alphabetically before

"Jason". If you need to make case sensitive or lexicographic comparisons, See "StrComp Function" on page 101 of the Language Reference.

Testing Multiple Comparisons

You can test more than one comparison at a time using the **And** and **Or** operators.

And Operator

Use this operator when you need to know if all comparisons evaluate to true. In the example below, if the variable x is 5 then the expression evaluates to false:

```
x>1 And x<5
```

Or Operator

Use this operator when you need to know if any of the comparisons evaluate to true. In the example below, if the variable x is 5 then the expression evaluates to true:

```
x>1 Or x<5
```

Executing Instructions Repeatedly with Loops

There may be times when one or more lines of code as a group will need to be executed more than once. If you know how many times the code should execute, you could simply repeat the code that many times. For example, if you wanted a pushbutton to

beep three times when clicked, you could simply put the Beep method in your code three times like this:

```
Beep
Beep
Beep
```

But say you need it to beep fifty times or perhaps until a certain condition is met? Simply repeating the code over and over in these cases will either be just tedious or not possible. How do you solve this problem? The answer is a *loop*.

Loops are design to allow one or more lines of code to execute over and over again.

While...Wend

A While loop is designed to execute one or more lines of code between the While and the Wend (While End) statements. The code between these statements will be executed repeatedly provided that the condition passed to the While statement continues to evaluate to true. Consider the following example:

```
Dim n As Integer
While n<10
   n=n+1
   Beep
Wend
```

The variable "n" will be zero by default when it is created by the Dim statement. Because zero is less than ten, execution will move inside the While...Wend loop. The variable n is incremented by

one. The Beep method plays the alert sound. REALbasic checks to see if the condition is still true and if it is, then the code inside the loop executes again. This continues until the condition is no longer true. If the variable n was not less than ten in the first place, execution would continue at the line of code after the Wend statement.

Do...Loop

Do loops are similar to While loops but a bit more flexible. Do loops continue to execute all lines of code between the Do and Loop statements *until* a particular condition is true. While loops on the other hand execute as long as the condition remains true. Do loops provide more flexibility than While loops because they allow you to test the condition at the beginning or end of the loop. The example below shows two loops; one testing the condition at the beginning and the other testing it at the end:

```
Do Until n=10
  n=n+1
  Beep
Loop


Do
  n=n+1
  Beep
Loop Until n=10
```

The difference between these two loops is that in the first loop, the loop will not execute if the variable n is already equal to ten. The second loop will execute at least one time regardless of the

value of n because the condition is not tested until the end of the loop.

It is possible to create a Do loop that does not test for any condition. Consider this loop:

```
Do
   n=n+1
   Beep
Loop
```

Because there is no test, this loop will run endlessly. You can call the Exit method to force a loop to exit without testing for a condition. However, this is poor design because you have to read through the code to figure out what will cause the loop to end.

Endless Loops

Make sure that the code inside your While and Do loops eventually causes the condition to be satisfied. Otherwise, you will end up with an endless loop that runs forever. Should you do this accidently, you can attempt to switch back to the Design environment by clicking on one of the Design environment's windows. Then you can choose Debug → Kill (⌘-K) to stop the loop. If this doesn't work, you will need to force REALbasic to quit by pressing ⌘-Control-Escape.

For...Next

While and Do loops are great when the number of times the loop should execute cannot be determined because it's based on a condition. A For loop is for those times when you can determine the number of times to execute the loop. For example, say you

want to add the numbers one through ten to a ListBox. Since you know exactly how many times the code should execute, a For loop is the right choice. For loops also differ from While and Do loops because For loops have a loop counter variable, a starting value for that variable and an ending value. The basic construction of a For loop is:

```
Dim counter As Integer
For counter=startingValue to endingValue
   [you code goes here]
Next
```

Notice the Dim statement is declaring counter as an integer. This is because the counter variable in a For loop must be an integer. The first time through the loop, counter variable will be set to startingValue. When the loop reaches the Next statement, the counter variable will be incremented by one. When the Next statement is reached and the counter variable is equal to endingValue, the counter will be incremented and the loop will end.

Let's take a look at the example mentioned earlier. You want to add the numbers one through ten to a ListBox. The following example accomplishes that:

```
Dim i As Integer
For i=1 to 10
  ListBox1.AddRow Str(i)
Next
```

The counter variable (i in this case) is passed to the Str function to be converted to a string so that it can be passed to the AddRow method of ListBox1.

Note: The letter "i" is commonly used for a loop counter because it's short for *iteration*.

While For loops, by default, increment the counter by one, it can be incremented (or decremented) by other values using the *Step* statement. In this example, the Step statement is added to increment the counter variable by 5 instead of 1:

```
Dim i As Integer
For i=5 to 100 Step 5
   ListBox1.AddRow Str(i)
Next
```

In this example, the For loop starts the counter at 100 and decrements by 5:

```
Dim i As Integer
For i=100 to 1 Step -5
   ListBox1.AddRow Str(i)
Next
```

A For loop (as well as any other kind of loop) can have another loop inside it. In the case of a For loop, the only thing you will have to watch out for is making sure that the counter variables are different so that the loops won't confuse each other. The example below uses a For loop embedded inside another For loop to go through all the cells of a multi-column ListBox counting the number of items the word "Hello" appears:

```
Dim row, column, count As Integer
For row=0 to listBox1.ListCount-1
  For column=0 to listBox1.ColumnCount-1
```

```
    if listbox1.cell(row,column)="hello" then
    count=count+1
    End if
  Next
Next
MsgBox Str(count)
```

For loops are generally more efficient than Do and While loops because the condition that will cause the loop to exit is only tested once at the beginning of the loop rather than each time through the loop.

Making Decisions with Branching

The methods you write execute one line at a time from top to bottom, left to right. There will be times when you want your application to execute some of it's code based on certain conditions. When your application's logic needs to make decisions it's called *branching*. This allows you to control what code gets executed and when. REALbasic provides two branching statements: If...Then and Select...Case.

If...Then...End If

This statement is used when your code needs to test a single boolean (true or false) condition and then execute code based on that condition. If the condition you are testing is true, then the lines of code you place between the If...Then line and the End If line are executed.

```
If condition Then

 [Your code goes here]
End If
```

Say you want to test the integer variable month and if it's value is 1, execute some code:

```
If month=1 Then

 [Your code goes here]

End If
```

month=1 is a boolean expression; it's either true or false. The variable month is either 1 or it's not 1.

Say you have a pushbutton that should perform an additional task if a particular checkbox is checked. The value property of a checkbox is boolean so you can test it in an If statement easily:

```
If checkbox1.value Then
End If
```

If...Then...Else...End If

There may be times when you need to perform one action if the boolean condition is true and another if it's false. In these cases, you can use the optional Else clause of an If statement. The Else clause allows you to divide up the code to be executed into two sections: the code that should be executed if the condition is true and the code that should be executed if it's false. In this example, one message is displayed if the condition is true while another is displayed if it's false:

```
If month=1 Then
  MsgBox "It’s January."
Else
  MsgBox "It’s not January."
End If
```

If...Then...ElseIf...End If

There may be times when you need to perform an additional test should the initial condition be false. In this case, use the optional ElseIf statement. In the example below, if the variable month is not 1 then the ElseIf statement is called to perform an additional test:

```
If month=1 Then
  MsgBox "It’s January."
ElseIf month<4 Then
  MsgBox "It’s still Winter."
End If
```

You could of course, use an additional If...Then...EndIf statement inside the Else portion of the first If statement to perform another test but this would add another EndIf and needlessly complicate your code. You can use as many ElseIf statements as you need. In this example, another ElseIf has been added to perform an additional test:

```
If month=1 Then
  MsgBox "It’s January."
ElseIf month<4 Then
  MsgBox "It’s still Winter."
```

```
ElseIf month<6 Then
  MsgBox "It must be Spring."
End If
```

Should the initial condition be false, REALbasic will continue to test the ElseIf conditions until it finds one that is true. It will then execute the code associated with that ElseIf statement and then continue with the line of code that follows the End If statement.

Select...Case

When you need to test a property or variable for one of many possible values and then take action based on that value, use a Select...Case statement. Consider the following example that tests a variable (dayNumber) then displays a message to the user to tell them which day of the week it is:

```
If dayNumber=2 Then
  MsgBox "It’s Monday."
ElseIf dayNumber=3 Then
  MsgBox "It’s Tuesday."
ElseIf dayNumber=4 Then
  MsgBox "It’s Wednesday."
ElseIf dayNumber=5 Then
  MsgBox "It’s Thursday."
ElseIf dayNumber=6 Then
  MsgBox "It’s Friday."
Else
  MsgBox "It’s the weekend."
```

```
End If
```

No two of these conditions can be true at the same time. While this method of writing the code will work it's not that easy to read. In this example, the same code is presented in a Select...Case statement making it far easier to read:

```
Select Case dayNumber
Case 2
  MsgBox "It's Monday."
Case 3
  MsgBox "It's Tuesday."
Case 4
  MsgBox "It's Wednesday."
Case 5
  MsgBox "It's Thursday."
Case 6
  MsgBox "It's Friday."
Else
  MsgBox "It's the weekend."
End Select
```

The Select...Case statement compares the variable or property passed in the first line to each case value. Once a match is found, the code between that case and the next is executed. Select...Case statements can contain an Else statement to handle all other values not explicitly handled by a case.

The Select...Case statement supports string and integer comparisons only. If you need to compare boolean, single or double values, or if you need to use a comparison operator other than the equality operator (=), use an If statement.