Microsoft Small Basic

An introduction to Programming

Chapter 1

An Introduction

Small Basic and Programming

Computer Programming is defined as the process of creating computer software using programming languages. Just like we speak and understand English or Spanish or French, computers can understand programs written in certain languages. These are called programming languages. In the beginning there were just a few programming languages and they were really easy to learn and comprehend. But as computers and software became more and more sophisticated, programming languages evolved fast, gathering more complex concepts along the way. As a result most modern programming languages and their concepts are pretty challenging to grasp by a beginner. This fact has started discouraging people from learning or attempting computer programming.

Small Basic is a programming language that is designed to make programming extremely easy, approachable and fun for beginners. Small Basic's intention is to bring down the barrier and serve as a stepping stone to the amazing world of computer programming.

The Small Basic Environment

Let us start with a quick introduction to the Small Basic Environment. When you first launch SmallBasic, you will see a window that looks like the following figure.

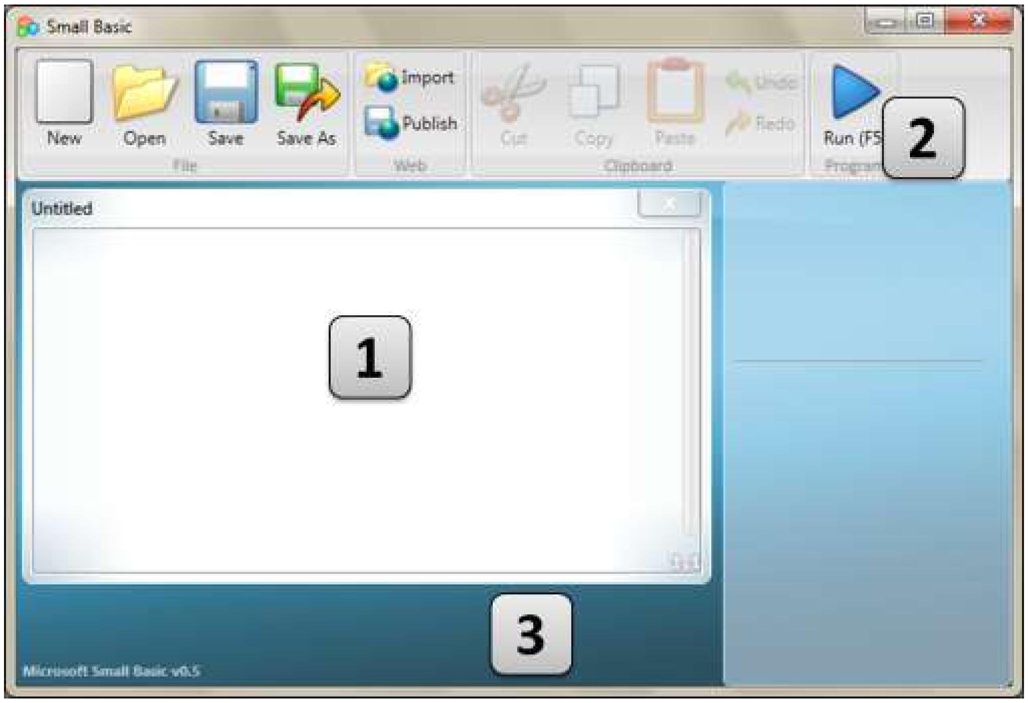


Figure 1 - The Small Basic Environment

This is the Small Basic Environment, where we'll write and run our Small Basic programs. This environment has several distinct elements which are identified by numbers.

The **Editor**, identified by [1] is where we will write our Small Basic programs. When you open a sample program or a previously saved program, it will show up on this editor. You can then modify it and save if for later use.

You can also open and work with more than one program at one time. Each program you are working with will be displayed in a separate editor. The editor that contains the program you are currently working with is called the *active editor*.

The **Toolbar**, identified by [2] is used to issue commands either to the *active editor* or the environment. We'll learn about the various commands in the toolbar as we go.

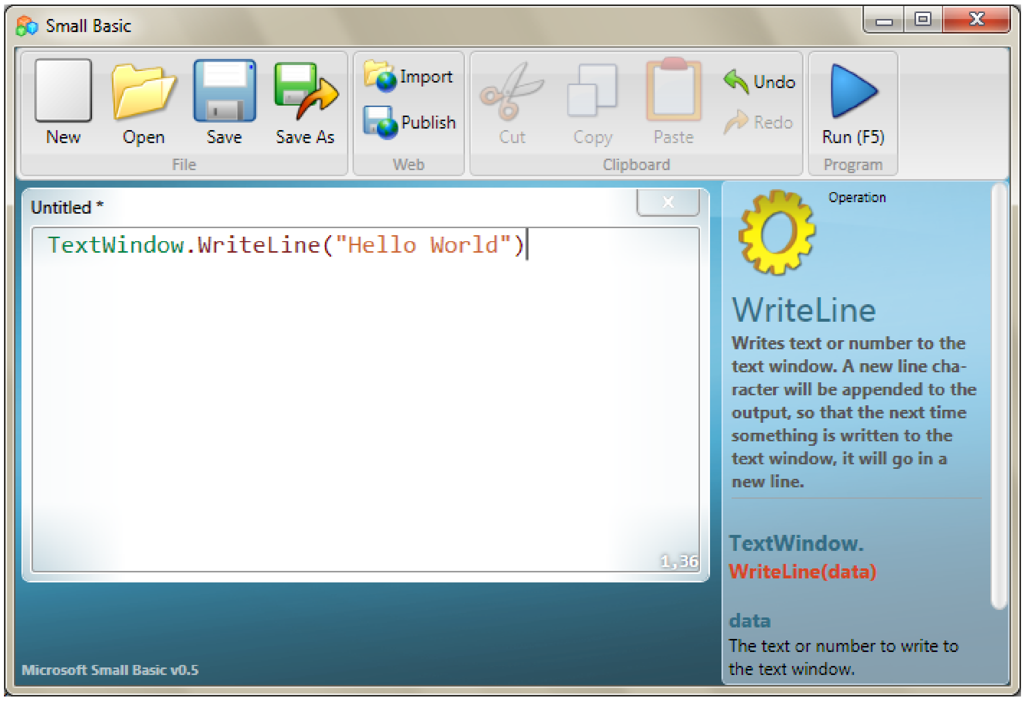
The **Surface**, identified by [3] is the place where all the editor windows go.

Our First Program

Now that you are familiar with the Small Basic Environment, we will go ahead and start programming in it. Like we just noted above, the editor is the place where we write our programs. So let's go ahead and type the following line in the editor.

```
TextWindow.WriteLine("Hello World")
```

This is our first Small Basic program. And if you have typed it correctly, you should see something similar to the figure below.



```
TextWindow.WriteLine("Hello World")
```

WriteLine

Writes text or number to the text window. A new line character will be appended to the output, so that the next time something is written to the text window, it will go in a new line.

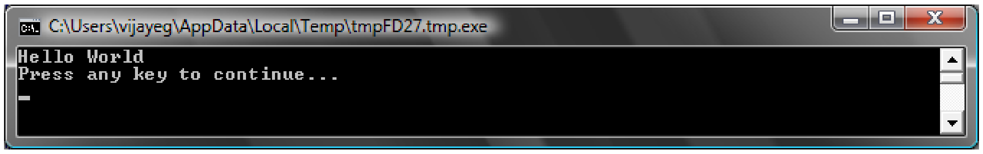
TextWindow.WriteLine(data)

data

The text or number to write to the text window.

Figure 2 - First Program

Now that we have typed our new program, let's go ahead and run it to see what happens. We can run our program either by clicking on the *Run* button on the toolbar or by using the shortcut key, F5 on the keyboard. If everything goes well, our program should run with the result as shown below.

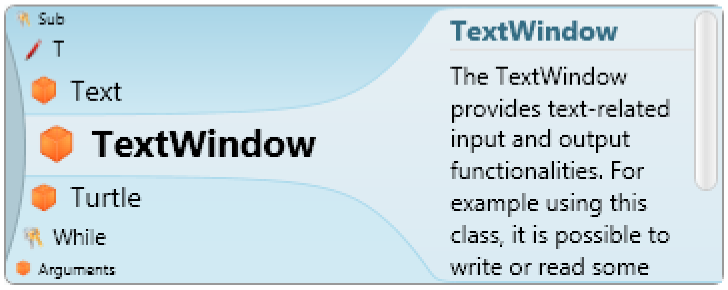


```
Hello World
Press any key to continue...
```

Figure 3 - First Program Output

Congratulations! You have just written and run the first Small Basic program. A very small and simple program, but nevertheless a big step towards becoming a real computer programmer! Now, there's just one more detail to cover before we go on to create bigger programs. We have to understand what just happened – what exactly did we tell the computer and how did the computer know what to do? In the next chapter, we'll analyze the program we just wrote, so we can gain that understanding.

As you typed your first program, you might have noticed that a popup appeared with a list of items (Figure 4). *This is called "intellisense" and it helps you type your program faster. You can traverse that list by pressing the Up/Down arrow keys, and when you find something you want, you can hit the Enter key to insert the selected item in your program.*



TextWindow

The TextWindow provides text-related input and output functionalities. For example using this class, it is possible to write or read some

Figure 4 - Intellisense

Saving our program

If you want to close Small Basic and come back later to work on the program you just typed, you can save the program. It is in fact a good practice to save programs from time to time, so that you don't lose information in the event of an accidental shutdown or a power failure. You can save the current program by either clicking on the "save" icon on the toolbar or by using the shortcut "Ctrl+S" (press the S key while holding down the Ctrl key).

Chapter 2

Understanding Our First Program

What really is a computer program?

A program is a set of instructions for the computer. These instructions tell the computer precisely what to do, and the computer always follows these instructions. Just like people, computers can only follow instructions if specified in a language they can understand. These are called programming languages. There are very many languages that the computer can understand and **Small Basic** is one.

Imagine a conversation happening between you and your friend. You and your friends would use words, organized as sentences to convey information back and forth. Similarly, programming languages contain collections of words that can be organized into sentences that convey information to the computer. And programs are basically sets of sentences (sometimes just a few and sometimes many thousands) that together make sense to both the programmer and the computer alike.

There are many languages that the computer can understand. Java, C++, Python, VB, etc. are all powerful modern computer languages that are used to develop simple to complex software programs.

Small Basic Programs

A typical Small Basic program consists of a bunch of *statements*. Every line of the program represents a statement and every statement is an instruction for the computer. When we ask the computer to execute a Small Basic program, it takes the program and reads the first statement. It understands what we're trying to say and then executes our instruction. Once it's done executing our first statement, it comes back to the program and reads and executes the second line. It continues to do so until it reaches the end of the program. That is when our program finishes.

Back to Our First Program

Here is the first program we wrote:

```
TextWindow.WriteLine("Hello World")
```

This is a very simple program that consists of one *statement*. That statement tells the computer to write a line of text which is **Hello World**, into the Text Window.

It literally translates in the computer's mind to:

```
Write Hello World
```

You might have already noticed that the statement can in turn be split into smaller segments much like sentences can be split into words. In the first statement we have 3 distinct segments:

a) TextWindow
b) WriteLine
c) "Hello World"

The dot, parentheses and the quotes are all punctuations that have to be placed at appropriate positions in the statement, for the computer to understand our intent.

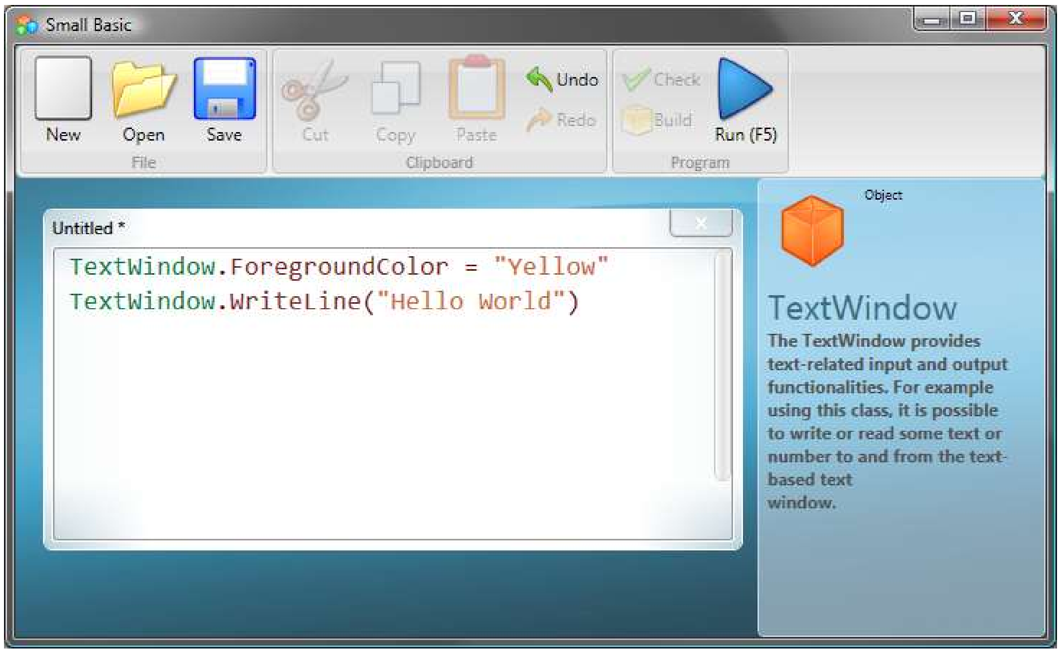
You might remember the black window that appeared when we ran our first program. That black window is called the TextWindow or sometimes referred to as the Console. That is where the result of this program goes. **TextWindow**, in our program, is called an *object*. There are a number of such objects available for us to use in our programs. We can perform several different *operations* on these objects. We've already used the*WriteLine* operation in our program. You might also have noticed that the WriteLine operation is followed by **Hello World** inside quotes. This text is passed as input to the WriteLine operation, which it then prints out to the user. This is called an *input* to the operation. Some operations take one or more inputs while others don't take any.

Punctuations such as quotes, spaces and parenthesis are very important in a computer program. Based on their position and count, they can change the meaning of what is being expressed.

Our Second Program

Now that you have understood our first program, let's go ahead and make it fancier by adding some colors.

```
TextWindow.ForegroundColor = "Yellow"
TextWindow.WriteLine("Hello World")
```



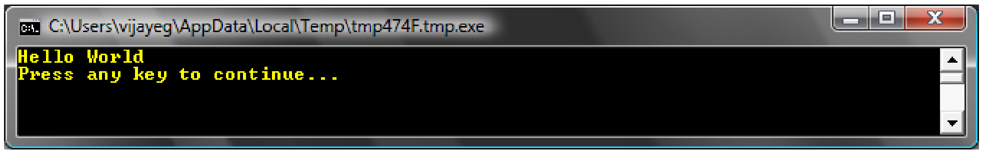
```
TextWindow.ForegroundColor = "Yellow"
TextWindow.WriteLine("Hello World")
```

TextWindow

The TextWindow provides text-related input and output functionalities. For example using this class, it is possible to write or read some text or number to and from the text-based text window.

Figure 5 - Adding Colors

When you run the above program, you'll notice that it prints out the same "Hello World" phrase inside TextWindow, but this time it prints it out in yellow instead of the gray that it did earlier.



```
Hello World
Press any key to continue...
```

Figure 6 - Hello World in Yellow

Notice the new statement we added to our original program. It uses a new word, *ForegroundColor* which we equated to a value of *"Yellow."* This means we've *assigned* "Yellow" to *ForegroundColor.* Now, the difference between ForegroundColor and the operation WriteLine is that ForegroundColor did not take any inputs nor did it need any parenthesis. Instead it was followed by an *equals to* symbol and a word. We define ForegroundColor as a *Property* of TextWindow. Here is a list of values that are valid for the ForegroundColor property. Try replacing "Yellow" with one of these and see the results – don't forget the quotes, they are required punctuations.

```
Black
Blue
Cyan
Gray
Green
Magenta
Red
White
```

```
Yellow
DarkBlue
DarkCyan
DarkGray
DarkGreen
DarkMagenta
DarkRed
DarkYellow
```

Chapter 3

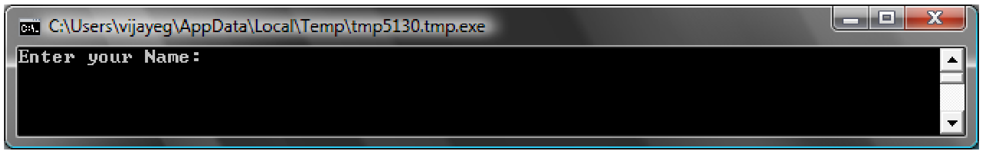
Introducing Variables

Using Variables in our program

Wouldn't it be nice if our program can actually say "Hello" with the users name instead of saying the generic "Hello World?" In order to do that we must first ask the user for his/her name and then store it somewhere and then print out "Hello" with the user's name. Let's see how we can do that:

```
TextWindow.Write("Enter your Name: ")
name = TextWindow.Read()
TextWindow.WriteLine("Hello " + name)
```

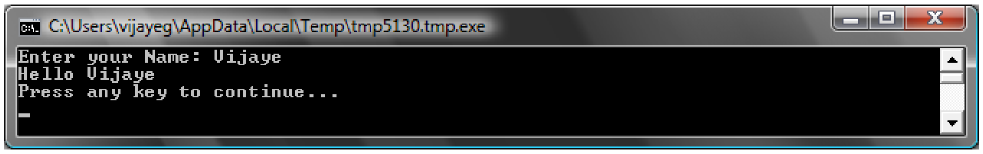
When you type and execute this program, you'll see an output like the following:



```
C:\Users\vijayeg\AppData\Local\Temp\tmp5130.tmp.exe
Enter your Name:
```

Figure 7 - Ask the user's name

And when you type in your name and hit ENTER, you'll see the following output:



```
C:\Users\vijayeg\AppData\Local\Temp\tmp5130.tmp.exe
Enter your Name: Vijaye
Hello Vijaye
Press any key to continue...
_
```

Figure 8 - A Warm Hello

Now, if you run the program again, you'll be asked the same question again. You can type in a different name and the computer will say Hello with that name.

Analysis of the program

In the program you just ran, the line that might have caught your attention is this:

```
name = TextWindow.Read()
```

Read() looks just like *WriteLine()*, but with no inputs. It is an operation and basically it tells the computer to wait for the user to type in something and hit the ENTER key. Once the user hits the ENTER key, it takes what the user has typed and returns it to the program. The interesting point is that whatever the user had typed is now stored in a *variable* called **name**. A *variable* is defined as a place where you can store values temporarily and use them later. In the line above, **name** was used to store the name of the user.

The next line is also interesting:

```
TextWindow.WriteLine("Hello " + name)
```

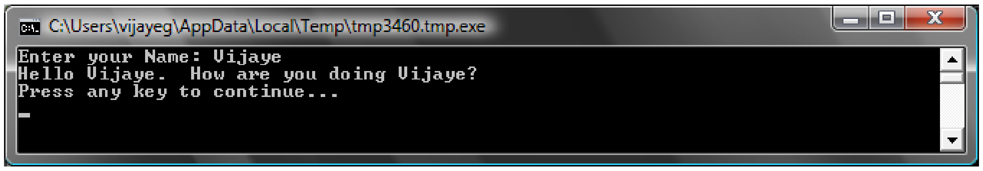
This is the place where we use the value stored in our variable, **name**. We take the value in **name** and append it to "Hello" and write it to the TextWindow.

Write*, just like* WriteLine *is another operation on ConsoleWindow. Write allows you to write something to the ConsoleWindow but allows succeeding text to be on the same line as the current text.*

Once a variable is set, you can reuse it any number of times. For example, you can do the following:

```
TextWindow.Write("Enter your Name: ")
name = TextWindow.Read()
TextWindow.Write("Hello " + name + ".  ")
TextWindow.WriteLine("How are you doing " + name + "?")
```

And you'll see the following output:



```
C:\Users\vijayeg\AppData\Local\Temp\tmp3460.tmp.exe
Enter your Name: Vijaye
Hello Vijaye.  How are you doing Vijaye?
Press any key to continue...
_
```

Figure 9 - Reusing a Variable

Rules for naming Variables

Variables have names associated with them and that's how you identify them. There are certain simple rules and some really good guidelines for naming these variables. They are:

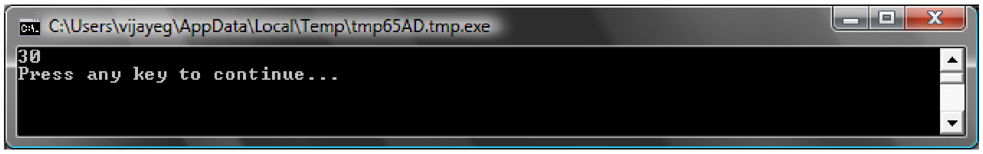
1. The name should start with a letter and should not collide with any of the keywords like **if**, **for**, **then**, etc.
2. A name can contain any combination of letters, digits and underscores.
3. It is useful to name variables meaningfully – since variables can be as long as you want, use variable names to describe their intent.

Playing with Numbers

We've just seen how you can use variables to store the name of the user. In the next few programs, we'll see how we can store and manipulate numbers in variables. Let's start with\ a really simple program:

```
number1 = 10
number2 = 20
number3 = number1 + number2
TextWindow.WriteLine(number3)
```

When you run this program you'll get the following as output:



```
C:\Users\vijayeg\AppData\Local\Temp\tmp65AD.tmp.exe
30
Press any key to continue...
```

Figure 10 - Adding Two Numbers

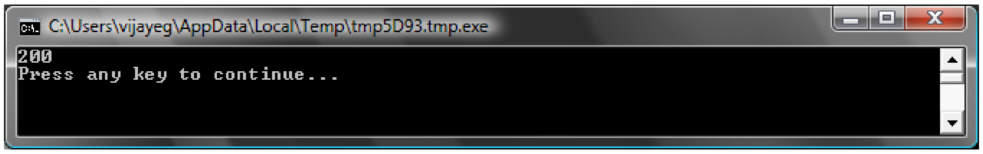
Notice that the numbers don't have quotes around them. For numbers, quotes are not necessary. You need quotes only when you're using text.

In the first line of the program, you're assigning the variable **number1** with a value of 10. And in the second line, you're assigning the variable **number2** with a value of 20. In the third line, you're adding **number1** and **number2** and then assigning the result of that to **number3**. So, in this case, **number3** will have a value of 30. And that is what we printed out to the TextWindow.

Now, let's modify that program slightly and see the results:

```
number1 = 10
number2 = 20
number3 = number1 * number2
TextWindow.WriteLine(number3)
```

The program above will multiply **number1** with **number2** and store the result in **number3**. And you can see in the result of that program below:



C:\Users\vijayeg\AppData\Local\Temp\tmp5D93.tmp.exe

```
200
Press any key to continue...
```

Figure 11 - Multiplying Two Numbers

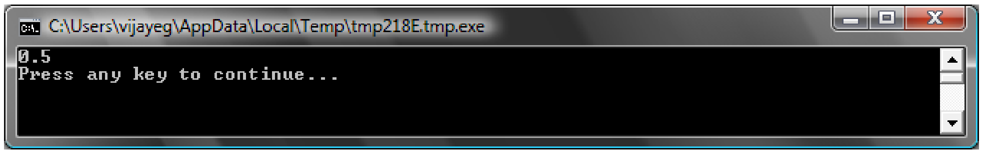
Similarly, you can subtract or divide numbers. Here is the subtraction:

```
number3 = number1 - number2
```

And the symbol for division is '/'. The progam will look like:

```
number3 = number1 / number2
```

And the result of this division would be:



C:\Users\vijayeg\AppData\Local\Temp\tmp218E.tmp.exe

```
0.5
Press any key to continue...
```

Figure 12 - Dividing Two Numbers

A Simple Temperature Converter

For the next program we'll use the formula $℃ = \frac{5(℉-32)}{9}$ to convert Fahrenheit temperatures to Celsius temperatures.

First, we'll get the temperature in Fahrenheit from the user and store it in a variable. There's a special operation that lets us read numbers from the user and that is **TextWindow.ReadNumber**.

```
TextWindow.Write("Enter temperature in Fahrenheit: ")
fahr = TextWindow.ReadNumber()
```

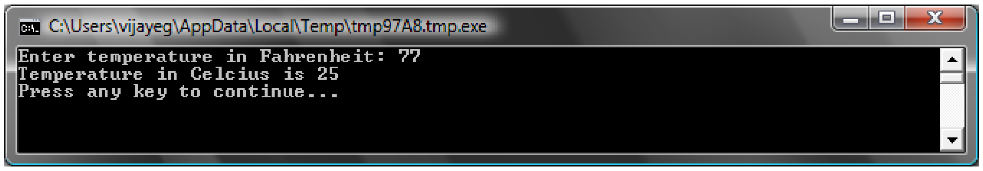
Once we have the Fahrenheit temperature stored in a variable, we can convert it to Celsius like this:

```
celsius = 5 * (fahr - 32) / 9
```

The parentheses tell the computer to calculate the **fahr – 32** part first and then process the rest. Now all we have to do is print the result out to the user. Putting it all together, we get this program:

```
TextWindow.Write("Enter temperature in Fahrenheit: ")
fahr = TextWindow.ReadNumber()
celsius = 5 * (fahr - 32) / 9
TextWindow.WriteLine("Temperature in Celsius is " + celsius)
```

And the result of this program would be:



```
Enter temperature in Fahrenheit: 77
Temperature in Celcius is 25
Press any key to continue...
```

Figure 13 - Temperature Conversion

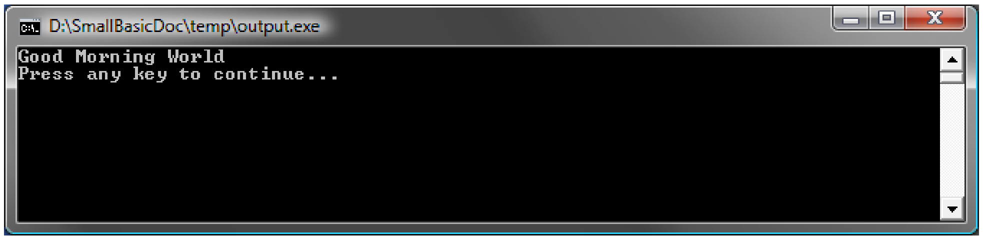
Chapter 4

Conditions and Branching

Going back to our first program, wouldn't it be cool that instead of saying the general *Hello World*, we could say *Good Morning World,* or *Good Evening World* depending on the time of the day? For our next program, we'll make the computer say *Good Morning World* if the time is earlier than 12PM; and *Good Evening* if the time is later than 12PM.

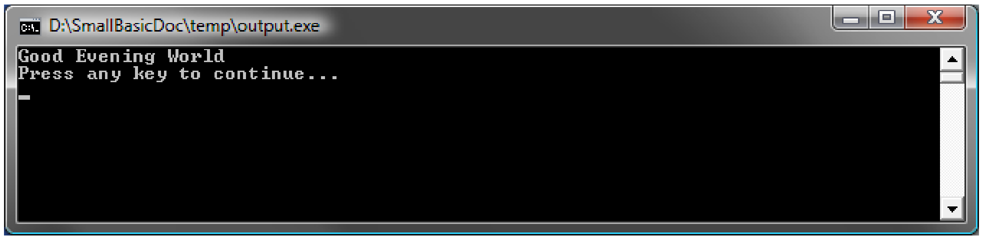
```
If (Clock.Hour < 12) Then
  TextWindow.WriteLine("Good Morning World")
EndIf
If (Clock.Hour >= 12) Then
  TextWindow.WriteLine("Good Evening World")
EndIf
```

Depending on when you run the program you'll see either of the following outputs:



```
Good Morning World
Press any key to continue...
```

Figure 14 - Good Morning World



```
Good Evening World
Press any key to continue...
```

Figure 15 - Good Evening World

Let's analyze the first three lines of the program. You'd have already figured out that this line tells the computer that if the Clock.Hour is lesser than 12, print out "Good Morning World." The words **If**, **Then** and **EndIf** are special words that are understood by the computer when the program is run. The word **If** is always followed by a condition, which in this case is (**Clock.Hour < 12)**. Remember that the parentheses are necessary for the computer to understand your intentions. The condition is followed by **then** and the actual operation to execute. And after the operation comes **EndIf**. This tells the computer that the conditional execution is over.

In Small Basic, you can use the Clock object to access the current date and time. It also provides you a bunch of properties that allow you to get the current Day, Month, Year, Hour, Minutes, Seconds separately.

Between the **then** and the **EndIf**, there could be more than one operation and the computer will execute them all if the condition is valid. For example, you could write something like this:

```
If (Clock.Hour < 12) Then
  TextWindow.Write("Good Morning. ")
  TextWindow.WriteLine("How was breakfast?")
EndIf
```

Else

In the program at the start of this chapter, you might have noticed that the second condition is kind of redundant. The **Clock.Hour** value could either be less than 12 or not. We didn't really have to do the second check. At times like this, we can shorten the two **if..then..endif** statements to be just one by using a new word, **else**.

If we were to rewrite that program using **else**, this is how it will look:

```
If (Clock.Hour < 12) Then
  TextWindow.WriteLine("Good Morning World")
Else
  TextWindow.WriteLine("Good Evening World")
EndIf
```

And this program will do exactly the same as the other one, which brings us to a very important lesson in computer programming:

> *In programming, there usually are many ways of doing the same thing. Sometimes one way makes more sense than the other way. The choice is left to the programmer. As you write more programs and get more experienced, you'll start to notice these different techniques and their advantages and disadvantages.*

Indentation

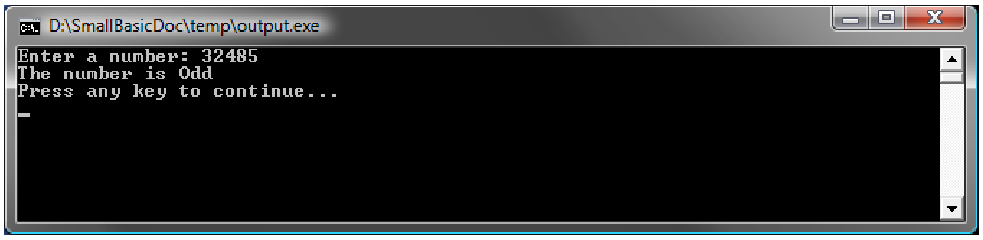
In all the examples you can see how the statements between *If, Else* and *EndIf* are indented. This indentation is not necessary. The computer will understand the program just fine without them. However, they help us see and understand the structure of the program easier. Hence, it's usually considered as a good practice to indent the statements between such blocks.

Even or Odd

Now that we have the **If..Then..Else..EndIf** statement in our bag of tricks, let's write out a program that, given a number, will say if it's even or odd.

```
TextWindow.Write("Enter a number: ")
num = TextWindow.ReadNumber()
remainder = Math.Remainder(num, 2)
If (remainder = 0) Then
  TextWindow.WriteLine("The number is Even")
Else
  TextWindow.WriteLine("The number is Odd")
EndIf
```

And when you run this program, you'll see an output like:



```
Enter a number: 32485
The number is Odd
Press any key to continue...
_
```

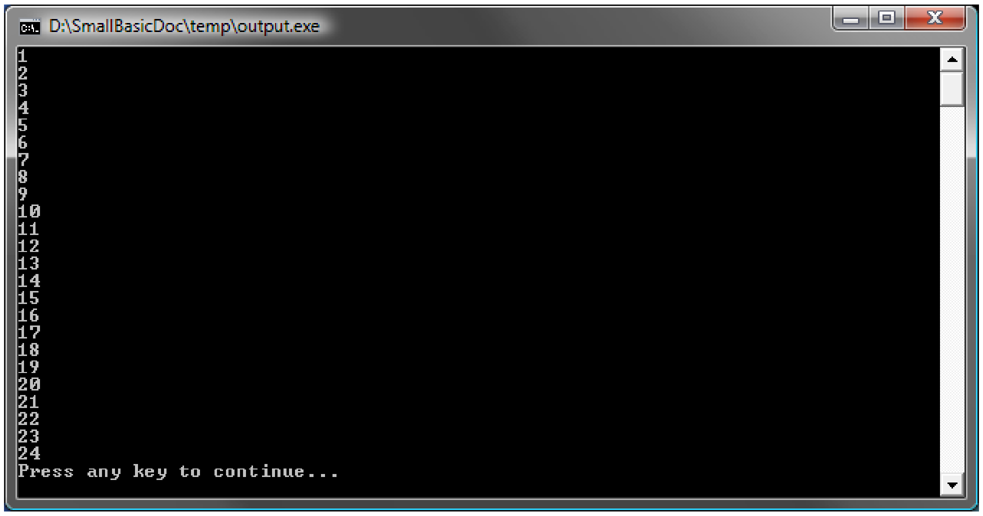
Figure 16 - Even or Odd

In this program, we've introduced another new useful operation, **Math.Remainder**. And yes, as you already might have figured out, **Math.Remainder** will divide the first number by the second number and then give back the remainder.

Branching

Remember, in the second chapter you learned that the computer processes a program one statement at a time, in order from the top to bottom. However, there's a special statement that can make the computer jump to another statement out of order. Let's take a look at the next program.

```
i = 1
start:
TextWindow.WriteLine(i)
i = i + 1
If (i < 25) Then
  Goto start
EndIf
```



```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
Press any key to continue...
```

Figure 17 - Using Goto

In the program above, we assigned a value of 1 to the variable **i**. And then we added a new statement which ends in a colon (:)

```
start:
```

This is called a *label*. Labels are like bookmarks that the computer understands. You can name the bookmark anything and you can add as many labels as you want in your program, as long as they are all uniquely named.

Another interesting statement here is:

```
i = i + 1
```

This just tells the computer to add 1 to the variable **i** and assign it back to **i**. So if the value of **i** was 1 before this statement, it will be 2 after this statement is run.

And finally,

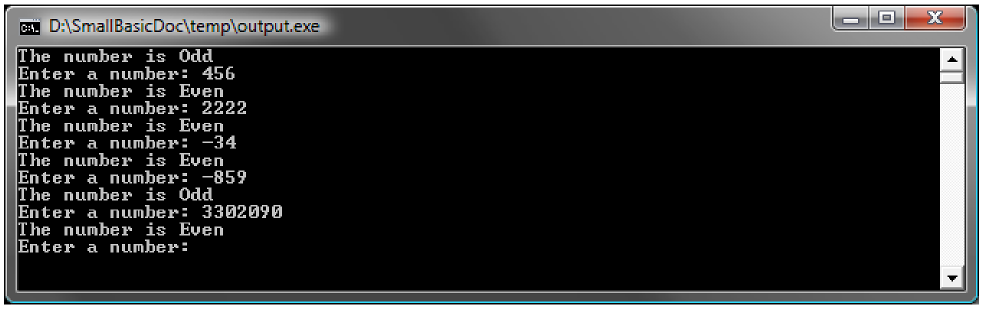
```
If (i < 25) Then
  Goto start
EndIf
```

This is the part that tells the computer that if the value of **i** is less than 25, start executing statements from the bookmark **start**.

Endless execution

Using the **Goto** statement you can make the computer repeat something any number of times. For example, you can take the Even or Odd program and modify it like below, and the program will run for ever. You can stop the program by clicking on the Close (X) button on the top right corner of the window.

```
begin:
TextWindow.Write("Enter a number: ")
num = TextWindow.ReadNumber()
remainder = Math.Remainder(num, 2)
If (remainder = 0) Then
  TextWindow.WriteLine("The number is Even")
Else
  TextWindow.WriteLine("The number is Odd")
EndIf
Goto begin
```



```
The number is Odd
Enter a number: 456
The number is Even
Enter a number: 2222
The number is Even
Enter a number: -34
The number is Even
Enter a number: -859
The number is Odd
Enter a number: 3302090
The number is Even
Enter a number:
```

Figure 18 - Even or Odd running endlessly

Chapter 5

Loops

For Loop

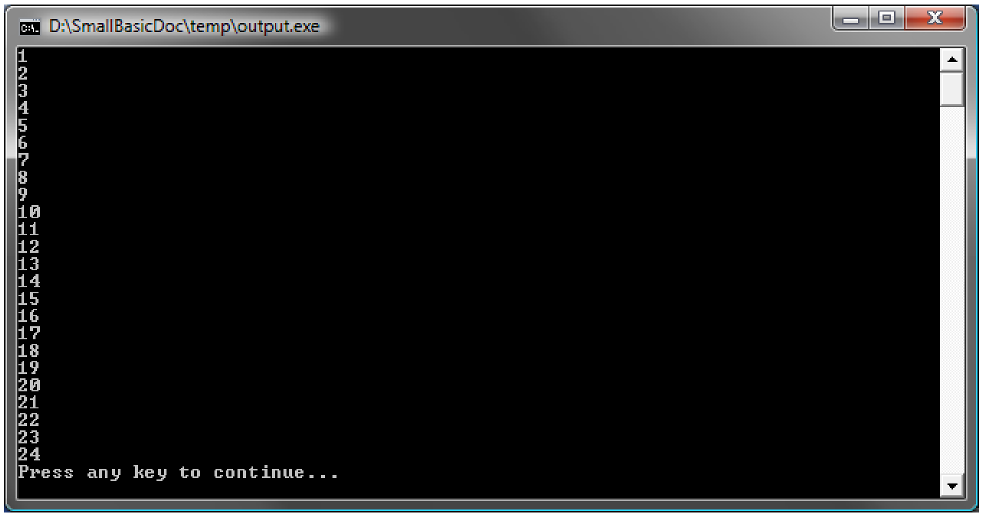
Let's take a program we wrote in the previous chapter.

```
i = 1
start:
TextWindow.WriteLine(i)
i = i + 1
If (i < 25) Then
  Goto start
EndIf
```

This program prints out numbers from 1 to 24 in order. This process of incrementing a variable is very common in programming that programming languages usually provide an easier method of doing this. The above program is equivalent to the program below:

```
For i = 1 To 24
  TextWindow.WriteLine(i)
EndFor
```

And the output is:



```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
Press any key to continue...
```

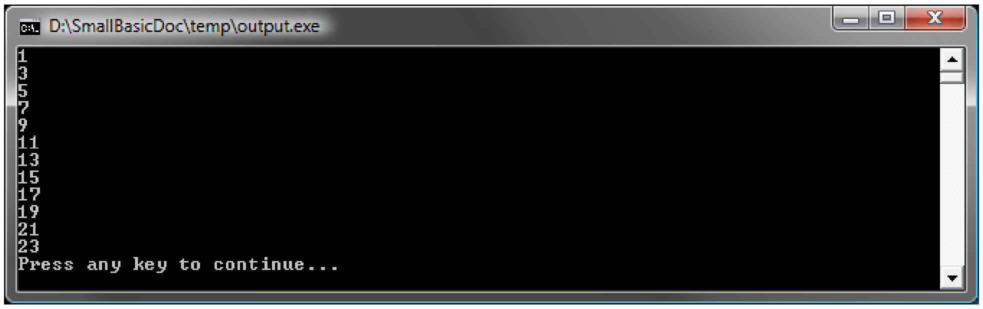
Figure 19 - Using the For Loop

Notice that we've reduced the 8 line program to a 4 line program, and it still does exactly the same as the 8 line program! Remember earlier we said that there are usually several ways of doing the same thing? This is a great example.

For..EndFor is, in programming terms, called a *loop*. It allows you to take a variable, give it an initial and an end value and let the computer increment the variable for you. Every time the computer increments the variable, it runs the statements between **For** and **EndFor**.

But if you wanted the variable to be incremented by 2 instead of 1 – like say, you wanted to print out all the odd numbers between 1 and 24, you can use the loop to do that too.

```
For i = 1 To 24 Step 2
  TextWindow.WriteLine(i)
EndFor
```

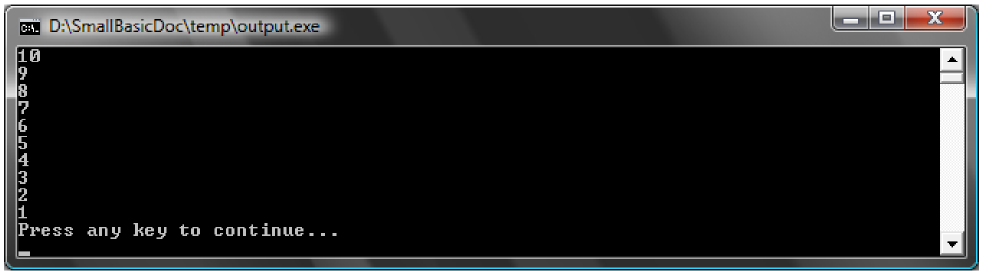


```
1
3
5
7
9
11
13
15
17
19
21
23
Press any key to continue...
```

Figure 20 - Just the Odd Numbers

The **Step 2** part of the **For** statement tells the computer to increment the value of **i** by 2 instead of the usual 1. By using **Step** you can specify any increment that you want. You can even specify a negative value for the step and make the computer count backwards, like in the example below:

```
For i = 10 To 1 Step -1
  TextWindow.WriteLine(i)
EndFor
```



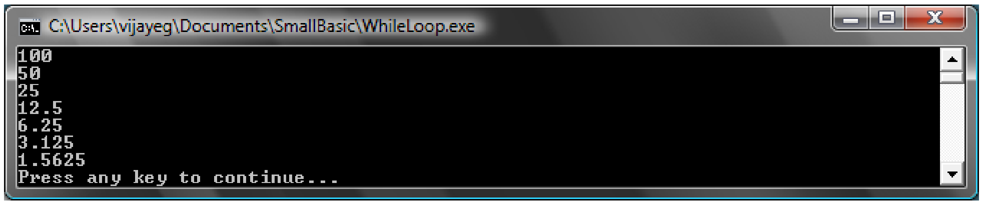
```
10
9
8
7
6
5
4
3
2
1
Press any key to continue...
```

Figure 21 - Counting Backwards

While Loop

The While loop is yet another looping method, that is useful especially when the loop count is not known ahead of time. Whereas a For loop runs for a pre-defined number of times, the While loop runs until a given condition is true. In the example below, we're halving a number until the result is greater than 1.

```
number = 100
While (number > 1)
  TextWindow.WriteLine(number)
  number = number / 2
EndWhile
```



```
100
50
25
12.5
6.25
3.125
1.5625
Press any key to continue...
```

Figure 22 - Halving Loop

In the program above, we assign the value 100 to *number* and run the while loop as long as number is greater than 1. Inside the loop, we print out the number and then we divide it by two, effectively halving it. And as expected, the output of the program is numbers that are progressively getting halved one after another.

It'll be really hard to write this program using a For loop, because we don't know how many times the loop will run. With a while loop it's easy to check for a condition and ask the computer to either continue the loop or quit.

It'll be interesting to note that every while loop can be unwrapped into an If..Then statement. For instance, the program above can be rewritten as follows, without affecting the end result.

```
number = 100
startLabel:
TextWindow.WriteLine(number)
number = number / 2

If (number > 1) Then
  Goto startLabel
EndIf
```

In fact, the computer internally rewrites every While loop into statements that use If..Then along with one or more Goto statements.

Chapter 6

Beginning Graphics

So far in all our examples, we've used the TextWindow to explain the fundamentals of the Small Basic language. However, Small Basic comes with a powerful set of Graphics capabilities that we'll start exploring in this chapter.

Introducing GraphicsWindow

Just like we had TextWindow that allowed us to work with Text and Numbers, Small Basic also provides a **GraphicsWindow** that we can use to draw things. Let's begin by displaying the GraphicsWindow.

```
GraphicsWindow.Show()
```

When you run this program, you'll notice that instead of the usual black text window, you get a white Window like the one shown below. There's nothing much to do on this window yet. But this will be the base window on which we'll work on in this chapter. You can close this window by clicking on the 'X' button on the top right corner.

Figure 23 - An empty Graphics Window

Setting up the Graphics Window

The graphics window allows you to customize its appearance to your desire. You can change the title, the background and its size. Let's go ahead and modify it a bit, just to get familiar with the window.

```
GraphicsWindow.BackgroundColor = "SteelBlue"
GraphicsWindow.Title = "My Graphics Window"
GraphicsWindow.Width = 320
GraphicsWindow.Height = 200
GraphicsWindow.Show()
```

Here's how the customized graphics window looks. You can change the background color to one of the many values listed in Appendix B. Play with these properties to see how you can modify the window's appearance.

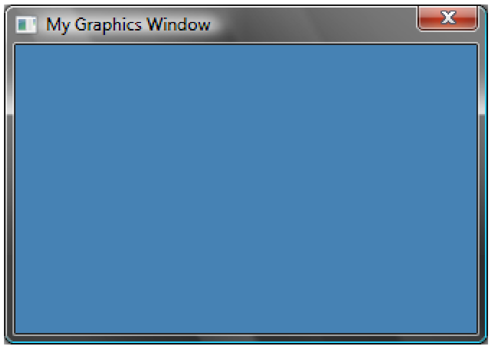


Figure 24 - A Custom Graphics Window

Drawing Lines

Once we have the GraphicsWindow up, we can draw shapes, text and even pictures on it. Let's start by drawing some simple shapes. Here's a program that draws a couple lines on the Graphics Window.

```
GraphicsWindow.Width = 200
GraphicsWindow.Height = 200
GraphicsWindow.DrawLine(10, 10, 100, 100)
GraphicsWindow.DrawLine(10, 100, 100, 10)
```

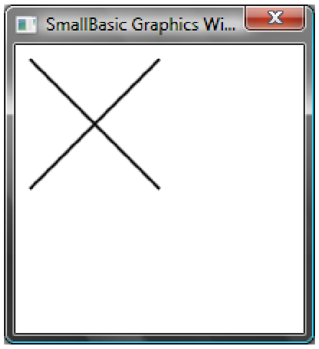


Figure 25 – CrissCross

The first two lines of the program setup the window and the next two lines draw the crisscross lines. The first two numbers that follow *DrawLine* specify the starting x and y co-ordinates and the other two specify the ending x and y co-ordinates. The interesting thing with computer graphics is

Instead of using names for colors you can use the web color notation (#RRGGBB). For example, #FF0000 denotes Red, #FFFF00 for Yellow, and so on. We'll learn more about colors in [TODO Colors chapter] that the co-ordinates (0, 0) start at the top left corner of the window. In effect, in the co-ordinate space the window is considered to be on the 2nd quadrant.

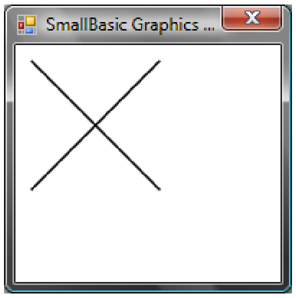


Figure 26 - The co-ordinate map

If we go back to the line program, it's interesting to note that Small Basic allows you to modify the properties of the line, such as the color and its thickness. First, let's modify the color of the lines as shown in the program below.

```
GraphicsWindow.Width = 200
GraphicsWindow.Height = 200
GraphicsWindow.PenColor = "Green"
GraphicsWindow.DrawLine(10, 10, 100, 100)
GraphicsWindow.PenColor = "Gold"
GraphicsWindow.DrawLine(10, 100, 100, 10)
```

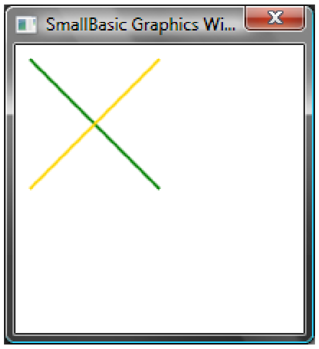


Figure 27 - Changing Line Color

Now, let's modify the size too. In the program below, we change the line width to be 10, instead of the default which is 1.

```
GraphicsWindow.Width = 200
GraphicsWindow.Height = 200
GraphicsWindow.PenWidth = 10
GraphicsWindow.PenColor = "Green"
GraphicsWindow.DrawLine(10, 10, 100, 100)
GraphicsWindow.PenColor = "Gold"
GraphicsWindow.DrawLine(10, 100, 100, 10)
```

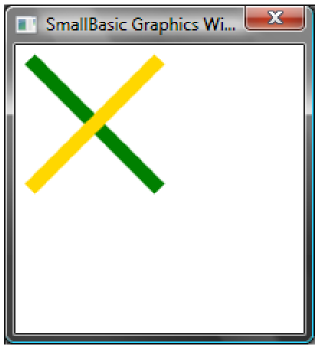


Figure 28 - Thick Colorful Lines

PenWidth and *PenColor* modify the pen with which these lines are drawn. They not only affect lines but also any shape that is drawn after the properties are updated.

By using the looping statements we learned in the previous chapters, we can easily write a program that draws multiple lines with increasing pen thickness.

```
GraphicsWindow.BackgroundColor = "Black"
GraphicsWindow.Width = 200
GraphicsWindow.Height = 160
GraphicsWindow.PenColor = "Blue"

For i = 1 To 10
  GraphicsWindow.PenWidth = i
  GraphicsWindow.DrawLine(20, i * 15, 180, i * 15)
endfor
```

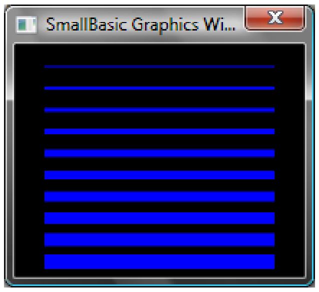


Figure 29 - Multiple Pen Widths

The interesting part of this program is the loop, where we increase the *PenWidth* every time the loop is run and then draw a new line under the old one.

Drawing and Filling Shapes

When it comes to drawing shapes, there are usually two types of operations for every shape. They are *Draw* operations and *Fill* operations. Draw operations draw the outline of the shape using a pen, and Fill operations paint the shape using a brush. For example in the program below, there are two rectangles, one that is drawn using the Red pen and one that's filled using the Green Brush.

```
GraphicsWindow.Width = 400
GraphicsWindow.Height = 300

GraphicsWindow.PenColor = "Red"
GraphicsWindow.DrawRectangle(20, 20, 300, 60)

GraphicsWindow.BrushColor = "Green"
GraphicsWindow.FillRectangle(60, 100, 300, 60)
```

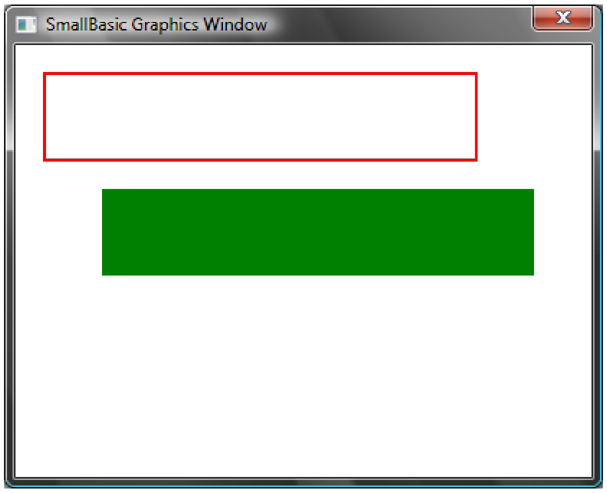


Figure 30 Drawing and Filling

To draw or fill a rectangle, you need four numbers. The first two numbers represent the X and Y co-ordinates for the top left corner of the rectangle. The third number specifies the width of the rectangle while the fourth specifies its height. In fact, the same applies for drawing and filling ellipses, as shown in the program below.

```
GraphicsWindow.Width = 400
GraphicsWindow.Height = 300

GraphicsWindow.PenColor = "Red"
GraphicsWindow.DrawEllipse(20, 20, 300, 60)

GraphicsWindow.BrushColor = "Green"
GraphicsWindow.FillEllipse(60, 100, 300, 60)
```

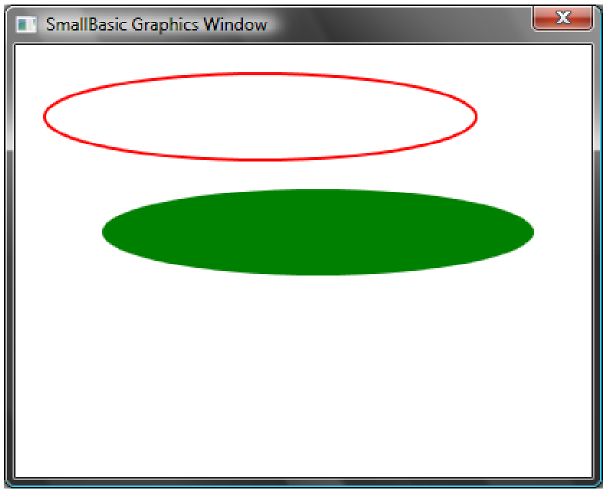


Figure 31 - Drawing and Filling Ellipses

Ellipses are just a general case of circles. If you want to draw circles, you would have to specify the same width and height.

```
GraphicsWindow.Width = 400
GraphicsWindow.Height = 300

GraphicsWindow.PenColor = "Red"
GraphicsWindow.DrawEllipse(20, 20, 100, 100)

GraphicsWindow.BrushColor = "Green"
GraphicsWindow.FillEllipse(100, 100, 100, 100)
```

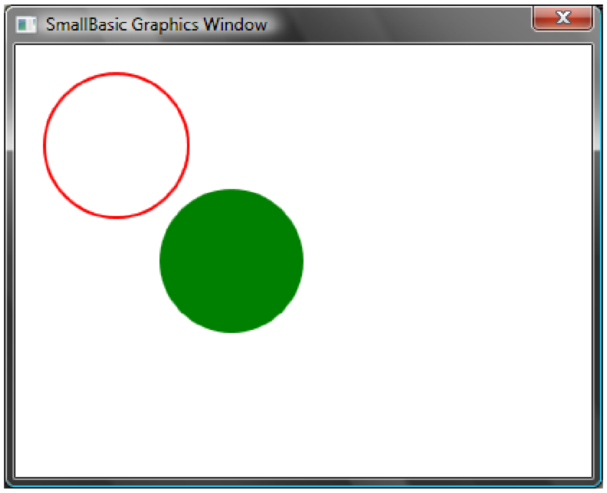


Figure 32 – Circles

Chapter 7

Fun with Shapes

We're going to have some fun in this chapter with whatever we've learned so far. This chapter contains samples that show some interesting ways of combining all that you've learned so far to create some cool looking programs.

Rectangalore

Here we draw multiple rectangles in a loop, with increasing size.

```
GraphicsWindow.BackgroundColor = "Black"
GraphicsWindow.PenColor = "LightBlue"
GraphicsWindow.Width = 200
GraphicsWindow.Height = 200

For i = 1 To 100 Step 5
  GraphicsWindow.DrawRectangle(100 - i, 100 - i, i * 2, i * 2)
EndFor
```

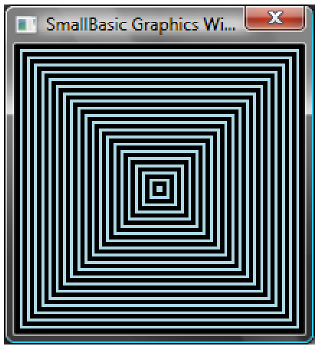


Figure 33 - Rectangalore

Circtacular

A variant of the previous program, draws circles instead of squares.

```
GraphicsWindow.BackgroundColor = "Black"
GraphicsWindow.PenColor = "LightGreen"
GraphicsWindow.Width = 200
GraphicsWindow.Height = 200

For i = 1 To 100 Step 5
  GraphicsWindow.DrawEllipse(100 - i, 100 - i, i * 2, i * 2)
EndFor
```

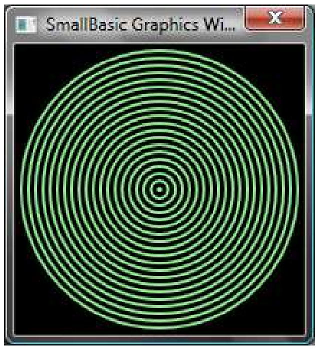


Figure 34 – Circtacular

Randomize

This program uses the operation GraphicsWindow.*GetRandomColor* to set random colors for the brush and then uses Math.*GetRandomNumber* to set the x and y co-ordinates for the circles. These two operations can be combined in interesting ways to create interesting programs that give different results each time they are run.

```
GraphicsWindow.BackgroundColor = "Black"
For i = 1 To 1000
  GraphicsWindow.BrushColor = GraphicsWindow.GetRandomColor()
  x = Math.GetRandomNumber(640)
  y = Math.GetRandomNumber(480)
  GraphicsWindow.FillEllipse(x, y, 10, 10)
EndFor
```

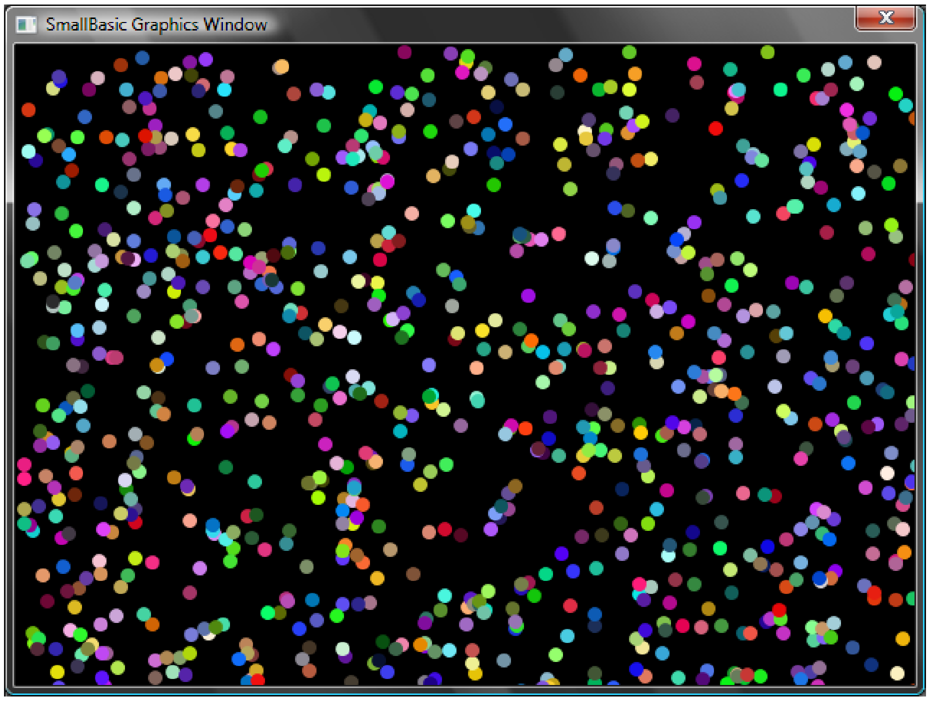


Figure 35 – Randomize

Fractals

The following program draws a simple triangle fractal using random numbers. A fractal is a geometric shape that can be subdivided into parts, each of which resembles the parent shape accurately. In this case, the program draws hundreds of triangles each of which resembles its parent triangle. And since the program runs for a few seconds, you can actually see the triangles forming slowly from mere dots. The logic itself is somewhat hard to describe and I'll leave it as an exercise for you to explore.

```
GraphicsWindow.BackgroundColor = "Black"
x = 100
y = 100

For i = 1 To 100000
  r = Math.GetRandomNumber(3)
  ux = 150
  uy = 30
  If (r = 1) then
    ux = 30
    uy = 1000
  EndIf

  If (r = 2) Then
    ux = 1000
    uy = 1000
  EndIf

  x = (x + ux) / 2
  y = (y + uy) / 2

  GraphicsWindow.SetPixel(x, y, "LightGreen")
EndFor
```

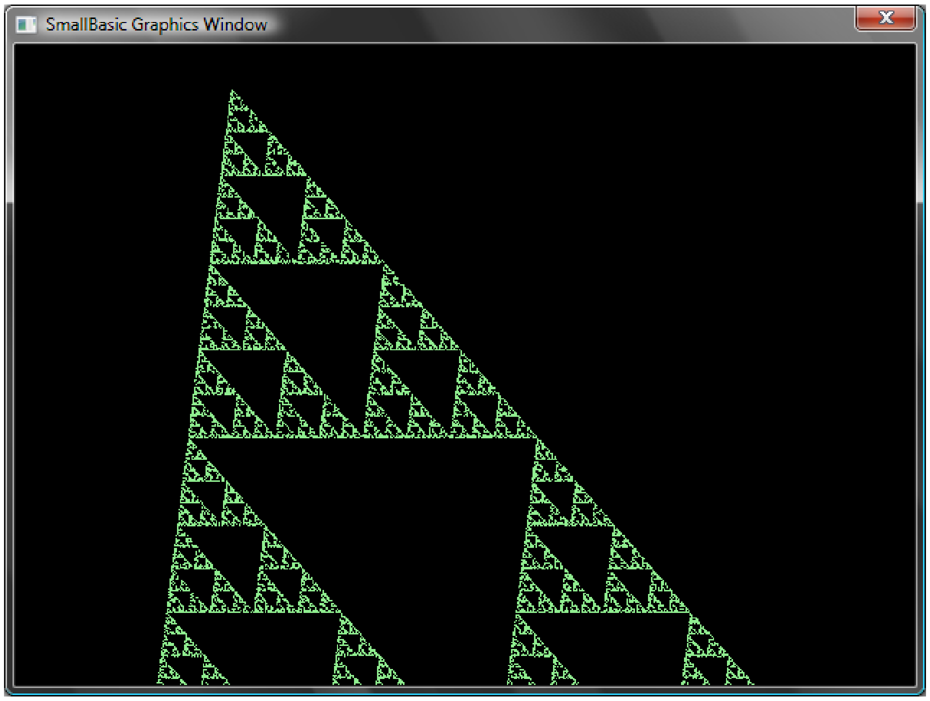


Figure 36 - Triangle Fractal

If you want to really see the dots slowly forming the fractal, you can introduce a delay in the loop by using the **Program.***Delay* operation. This operation takes in a number that specifies in milliseconds, how long to delay. Here's the modified program, with the modified line in bold.

```
GraphicsWindow.BackgroundColor = "Black"
x = 100
y = 100

For i = 1 To 100000
  r = Math.GetRandomNumber(3)
  ux = 150
  uy = 30
  If (r = 1) then
    ux = 30
    uy = 1000
  EndIf

  If (r = 2) Then
    ux = 1000
    uy = 1000
  EndIf

  x = (x + ux) / 2
  y = (y + uy) / 2

  GraphicsWindow.SetPixel(x, y, "LightGreen")
  Program.Delay(2)
EndFor
```

Increasing the delay will make the program slower. Experiment with the numbers to see what's best for your taste.

Another modification you can make to this program is to replace the following line:

```
GraphicsWindow.SetPixel(x, y, "LightGreen")
```

with

```
color = GraphicsWindow.GetRandomColor()
GraphicsWindow.SetPixel(x, y, color)
```

This change will make the program draw the pixels of the triangle using random colors.

Chapter 8

Turtle Graphics

Logo

In the 1970s, there was a very simple but powerful programming language, called Logo that was used by a few researchers. This was until someone added what is called “Turtle Graphics” to the language and made available a “Turtle” that was visible on the screen and responded to commands like *Move Forward, Turn Right, Turn Left,* etc. Using the Turtle, people were able to draw interesting shapes on the screen. This made the language immediately accessible and appealing to people of all ages, and was largely responsible for its wild popularity in the 1980s.

Small Basic comes with a **Turtle** object with many commands that can be called from within Small Basic programs. In this chapter, we’ll use the Turtle to draw graphics on the screen.

The Turtle

To begin with, we need to make the Turtle visible on the screen. This can be achieved by a simple one line program.

```
Turtle.Show()
```

When you run this program you’ll notice a white window, just like the one we saw in the previous chapter, except this one has a Turtle in the center. It is this Turtle that is going to follow our instructions and draw whatever we ask it to.

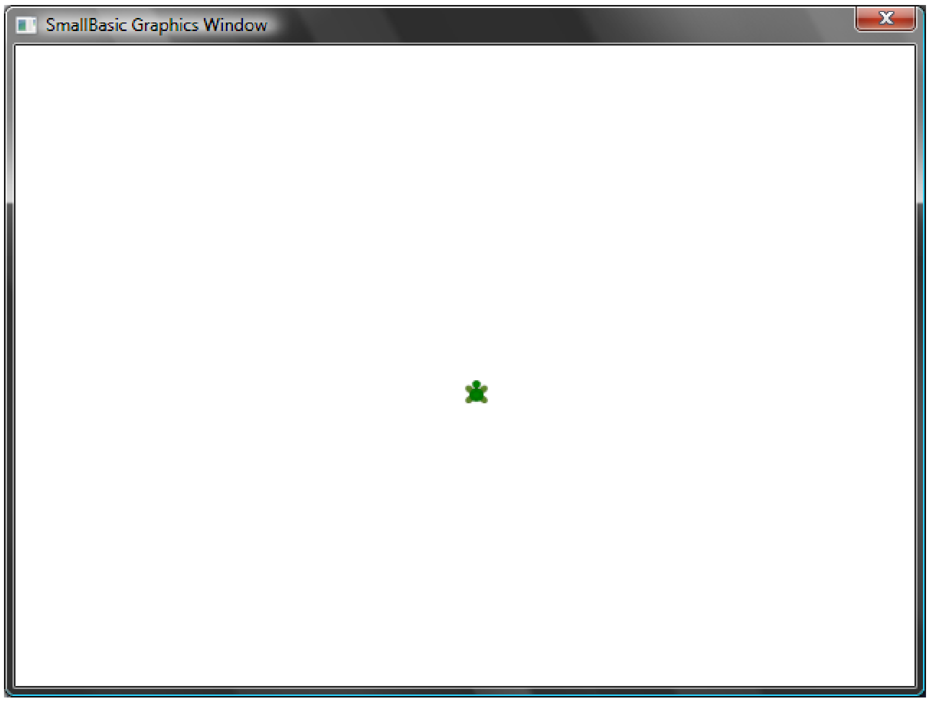


Figure 37 - Turtle is visible

Moving and Drawing

One of the instructions that the Turtle understands is **Move**. This operation takes a number as input. This number tells the Turtle how far to move. Say, in the example below, we'll ask the Turtle to move 100 pixels.

```
Turtle.Move(100)
```

When you run this program, you can actually see the turtle move slowly a 100 pixels upwards. As it moves, you'll also notice it drawing a line behind it. When the Turtle has finished moving, the result will look something like the figure below.

When using operations on the Turtle, it is not necessary to call Show(). The Turtle will be automatically made visible whenever any Turtle operation is performed.

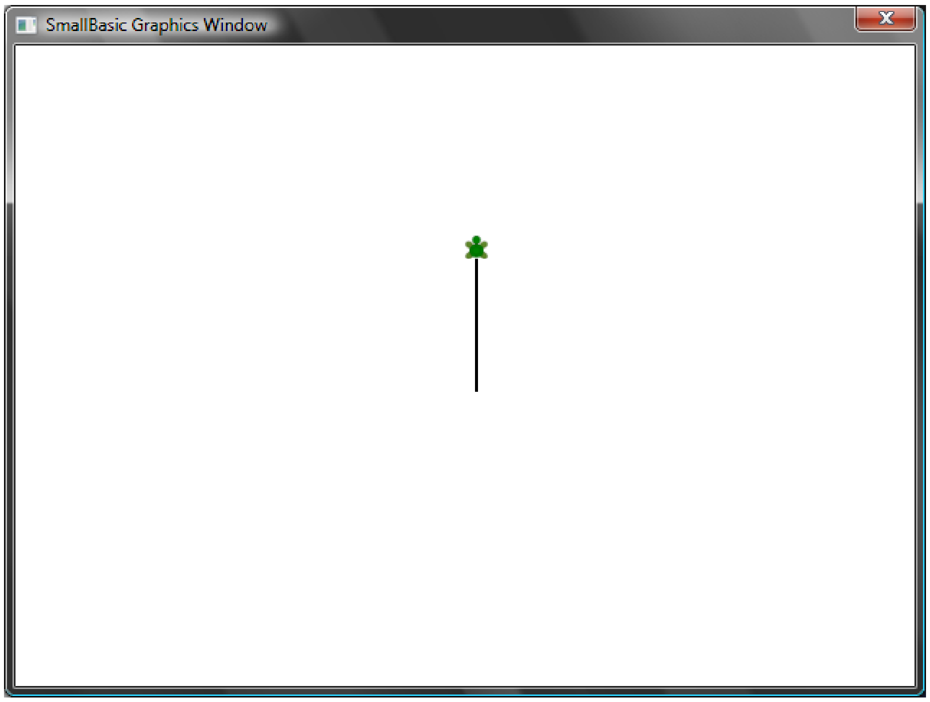


Figure 38 - Move a hundred pixels

Drawing a Square

A square has four sides, two vertical and two horizontal. In order to draw a square we need to be able to make the Turtle draw a line, turn right and draw another line and continue this until all four sides are finished. If we translated this to a program, here's how it would look.

```
Turtle.Move(100)
Turtle.TurnRight()
Turtle.Move(100)
Turtle.TurnRight()
Turtle.Move(100)
Turtle.TurnRight()
Turtle.Move(100)
Turtle.TurnRight()
```

When you run this program, you can see the Turtle drawing a square, one line at a time, and the result looks like the figure below.

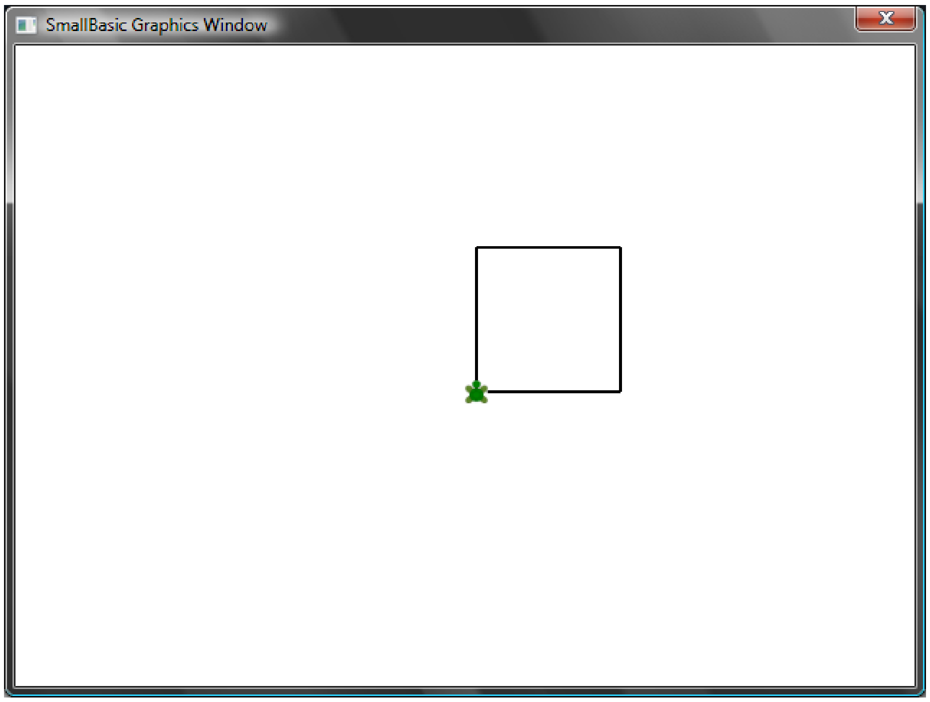


Figure 39 - Turtle drawing a square

It will be interesting to note that we're issuing the same two instructions over and over – four times precisely. And we've already learnt that such repetitive commands can be executed using loops. So, if we take the program above and modify it to use the **For..EndFor** loop, we'll end up with a much simpler program.

```
 For i = 1 To 4
  Turtle.Move(100)
  Turtle.TurnRight()
EndFor
```

Changing Colors

The Turtle draws on the exact same GraphicsWindow that we saw in the previous chapter. This means that all the operations that we learned in the previous chapter are still valid here. For instance, the following program will draw the square with each side in a different color.

```
For i = 1 To 4
  GraphicsWindow.PenColor = GraphicsWindow.GetRandomColor()
  Turtle.Move(100)
  Turtle.TurnRight()
EndFor
```

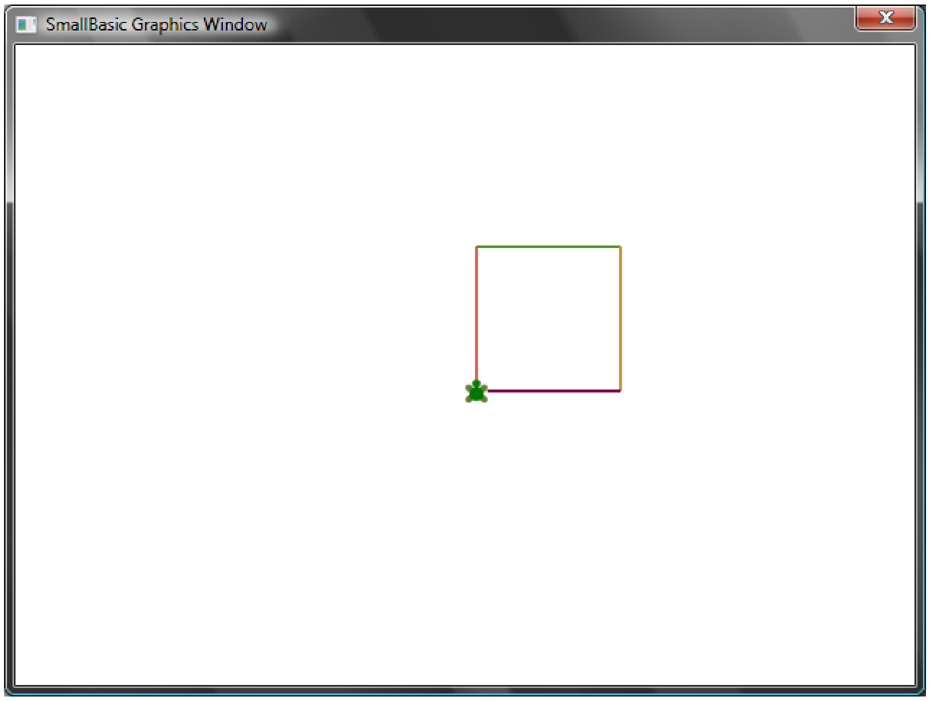


Figure 40 - Changing Colors

Drawing more complex shapes

The Turtle, in addition to the **TurnRight** and **TurnLeft** operations, has a **Turn** operation. This operation takes one input which specifies the angle of rotation. Using this operation, it is possible to draw any sided polygon. The following program draws a hexagon (a six-sided polygon).

```
For i = 1 To 6
  Turtle.Move(100)
  Turtle.Turn(60)
EndFor
```

Try this program out to see if it really draws a hexagon. Observe that since the angle between the sides is 60 degrees, we use **Turn(60)**. For such a polygon, whose sides are all equal, the angle between the sides can be easily obtained by dividing 360 by the number of sides. Armed with this information and using variables, we can write a pretty generic program that can draw any sided polygon.

```
sides = 12

length = 400 / sides
angle = 360 / sides

For i = 1 To sides
```

```
  Turtle.Move(length)
  Turtle.Turn(angle)
EndFor
```

Using this program, you can draw any polygon by just modifying the **sides** variable. Putting 4 here would give us the Square we started with. Putting a sufficiently large value, say 50 would make the result indistinguishable from a circle.

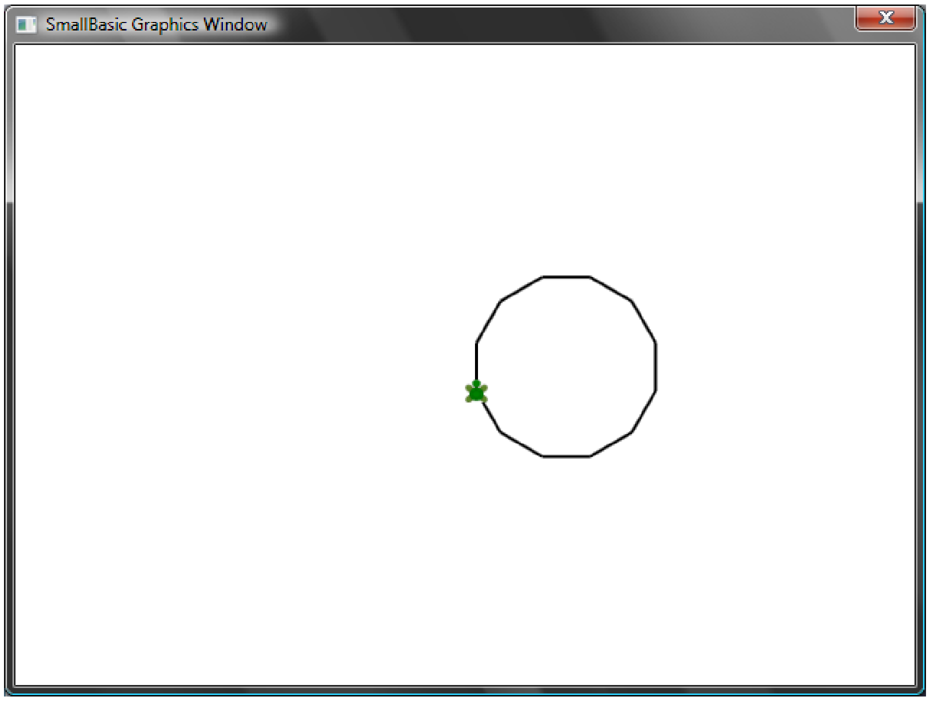


Figure 41 - Drawing a 12 sided polygon

Using the technique we just learned, we can make the Turtle draw multiple circles each time with a little shift resulting in an interesting output.

```
sides = 50
length = 400 / sides
angle = 360 / sides

Turtle.Speed = 9

For j = 1 To 20
  For i = 1 To sides
    Turtle.Move(length)
    Turtle.Turn(angle)
  EndFor
  Turtle.Turn(18)
```

```
EndFor
```

The program above has two **For..EndFor** loops, one within the other. The inner loop (*i = 1 to sides*) is similar to the polygon program and is responsible for drawing a circle. The outer loop (*j = 1 to 20*) is responsible for turning the Turtle by a small bit for every circle that is drawn. This tells the Turtle to draw 20 circles. When put together, this program results in a very interesting pattern, like the one shown below.

In the program above, we have made the Turtle go faster by setting the Speed to 9. You can set this property to any value between 1 and 10 to make the Turtle go as fast as you want.

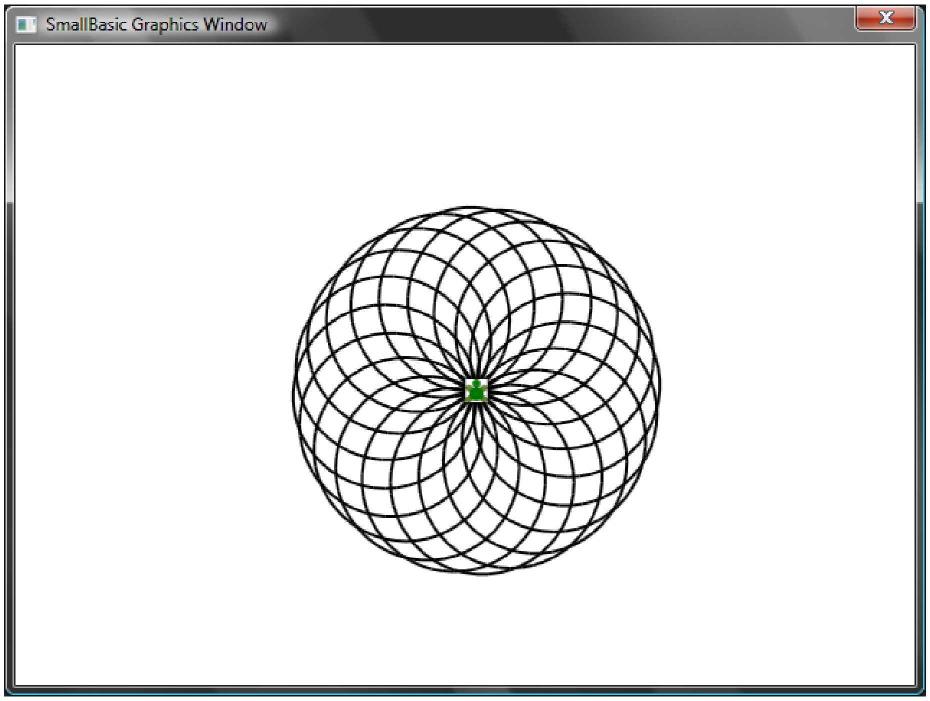


Figure 42 - Going in circles

Moving Around

You can make the turtle not draw by calling the **PenUp** operation. This allows you to move the turtle to anywhere on the screen without drawing a line. Calling **PenDown** will make the turtle draw again. This can be used to get some interesting effects, like say, dotted lines. Here's a program that uses this to draw a dotted line polygon.

```
sides = 6

length = 400 / sides
angle = 360 / sides
```

```
For i = 1 To sides
  For j = 1 To 6
    Turtle.Move(length / 12)
    Turtle.PenUp()
    Turtle.Move(length / 12)
    Turtle.PenDown()
  EndFor
  Turtle.Turn(angle)
EndFor
```

Again, this program has two loops. The inner loop draws a single dotted line, while the outer loop specifies how many lines to draw. In our example, we used 6 for the **sides** variable and hence we got a dotted line hexagon, as below.

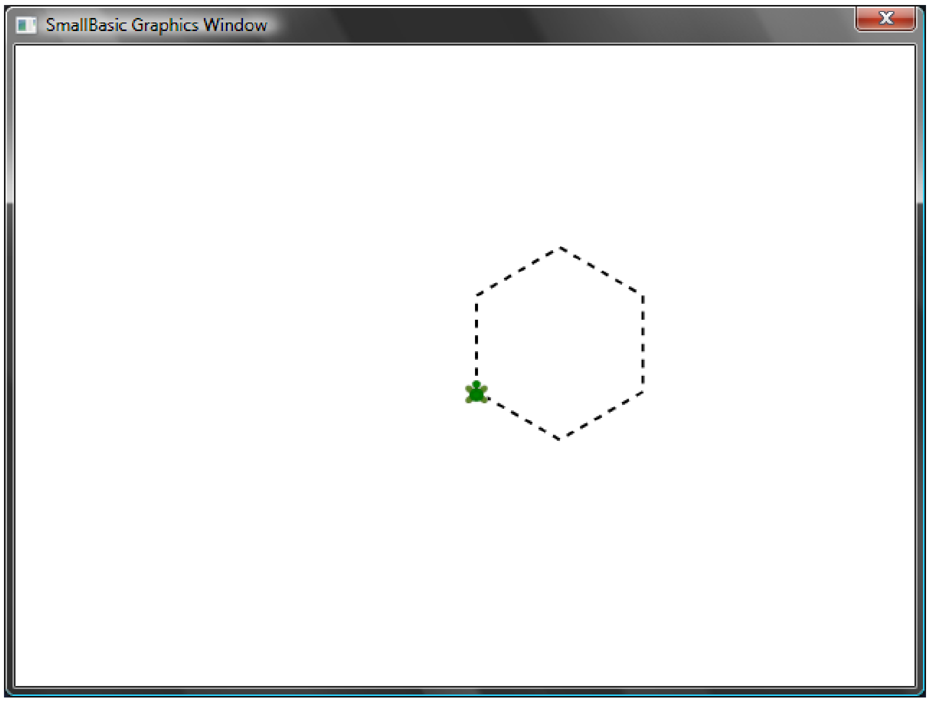


Figure 43 - Using PenUp and PenDown

Chapter 9

Subroutines

Very often while writing programs we'll run into cases where we'll have to execute the same set of steps, over and over again. In those cases, it probably wouldn't make sense to rewrite the same statements multiple times. That's when *Subroutines* come in handy.

A subroutine is a portion of code within a larger program that usually does something very specific, and that can be called from anywhere in the program. Subroutines are identified by a name that follows the **Sub** keyword and are terminated by the **EndSub** keyword. For example, the following snippet represents a subroutine whose name is *PrintTime*, and it does the job of printing the current time to the TextWindow.

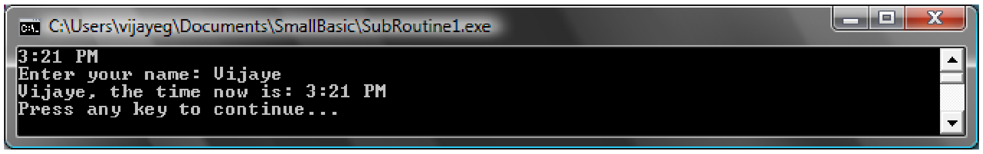
```
Sub PrintTime
  TextWindow.WriteLine(Clock.Time)
EndSub
```

Below is a program that includes the subroutine and calls it from various places.

```
PrintTime()
TextWindow.Write("Enter your name: ")
name = TextWindow.Read()
TextWindow.Write(name + ", the time now is: ")
PrintTime()

Sub PrintTime
  TextWindow.WriteLine(Clock.Time)
```

```
EndSub
```



C:\Users\vijayeg\Documents\SmallBasic\SubRoutine1.exe

```
3:21 PM
Enter your name: Vijaye
Vijaye, the time now is: 3:21 PM
Press any key to continue...
```

Figure 44 - Calling a simple Subroutine

You execute a subroutine by calling *SubroutineName()*. As usual, the punctuators "()" are necessary to tell the computer that you want to execute a subroutine.

Advantages of using Subroutines

As we just saw above, subroutines help reduce the amount of code you have to type in. Once you have the *PrintTime* subroutine written, you can call it from anywhere in your program and it'll print the current time.

Remember, you can only call a SmallBasic subroutine from within the same program. You cannot call a subroutine from within another program.

In addition, subroutines can help decompose complex problems into simpler pieces. Say you had a complex equation to solve, you can write several subroutines that solved smaller pieces of the complex equation. Then you can put the results together to get the solution to the original complex equation.

Subroutines can also aid in improving the readability of a program. In other words, if you have well named subroutines for commonly run portions of your program, your program becomes easy to read and comprehend. This is very important if you want to understand someone else's program or if you want your program to be understood by others. Sometimes, it is helpful even when you want to read your own program, say a week after you wrote it.

Using variables

You can access and use any variable that you have in a program from within a subroutine. As an example, the following program accepts two numbers and prints out the larger of the two. Notice that the variable *max* is used both inside and outside of the subroutine.

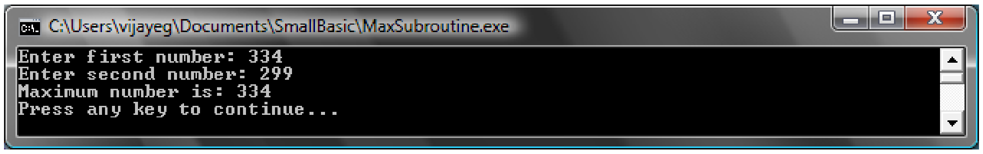
```
TextWindow.Write("Enter first number: ")
num1 = TextWindow.ReadNumber()
TextWindow.Write("Enter second number: ")
num2 = TextWindow.ReadNumber()

FindMax()
```

```
TextWindow.WriteLine("Maximum number is: " + max)

Sub FindMax
  If (num1 > num2) Then
    max = num1
  Else
    max = num2
  EndIf
EndSub
```

And the output of this program looks like this.



C:\Users\vijayeg\Documents\SmallBasic\MaxSubroutine.exe

```
Enter first number: 334
Enter second number: 299
Maximum number is: 334
Press any key to continue...
```

Figure 45 - Max of two numbers using Subroutine

Let's look at another example that will illustrate the usage of Subroutines. This time we'll use a graphics program that computes various points which it will store in variables *x* and *y*. Then it calls a subroutine **DrawCircleUsingCenter** which is responsible for drawing a circle using *x* and *y* as the center.

```
GraphicsWindow.BackgroundColor = "Black"
GraphicsWindow.PenColor = "LightBlue"
GraphicsWindow.Width = 480
For i = 0 To 6.4 Step 0.17
  x = Math.Sin(i) * 100 + 200
  y = Math.Cos(i) * 100 + 200

  DrawCircleUsingCenter()
EndFor

Sub DrawCircleUsingCenter
  startX = x - 40
  startY = y - 40

  GraphicsWindow.DrawEllipse(startX, startY, 120, 120)
EndSub
```

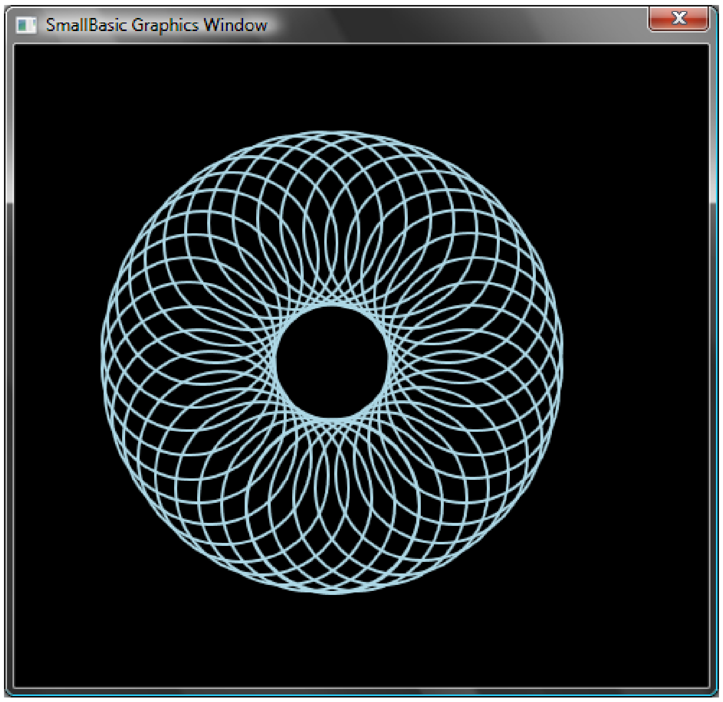


Figure 46 - Graphics Example for Subroutines

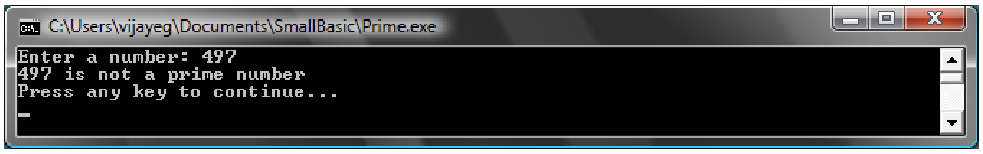
Calling Subroutines inside Loops

Sometimes subroutines get called from inside a loop, during which time they execute the same set of statements but with different values in one or more of the variables. For instance, say if you have a subroutine named *PrimeCheck* and this subroutine determines if a number is prime or not. You can write a program that lets the user to enter a value and you can then say if it is prime or not, using this subroutine. The program below illustrates that.

```
TextWindow.Write("Enter a number: ")
i = TextWindow.ReadNumber()
isPrime = "True"
PrimeCheck()
If (isPrime = "True") Then
  TextWindow.WriteLine(i + " is a prime number")
Else
  TextWindow.WriteLine(i + " is not a prime number")
EndIf

Sub PrimeCheck
  For j = 2 To Math.SquareRoot(i)
    If (Math.Remainder(i, j) = 0) Then
      isPrime = "False"
      Goto EndLoop
    EndIf
  Endfor
EndLoop:
EndSub
```

The PrimeCheck subroutine takes the value of *i* and tries to divide it by smaller numbers. If a number divides *i* and leaves no remainder, then *i* is not a prime number. At that point the subroutine sets the value of *isPrime* to “False” and exits. If the number was indivisible by smaller numbers then the value of *isPrime* remains as “True.”



```
Enter a number: 497
497 is not a prime number
Press any key to continue...
```

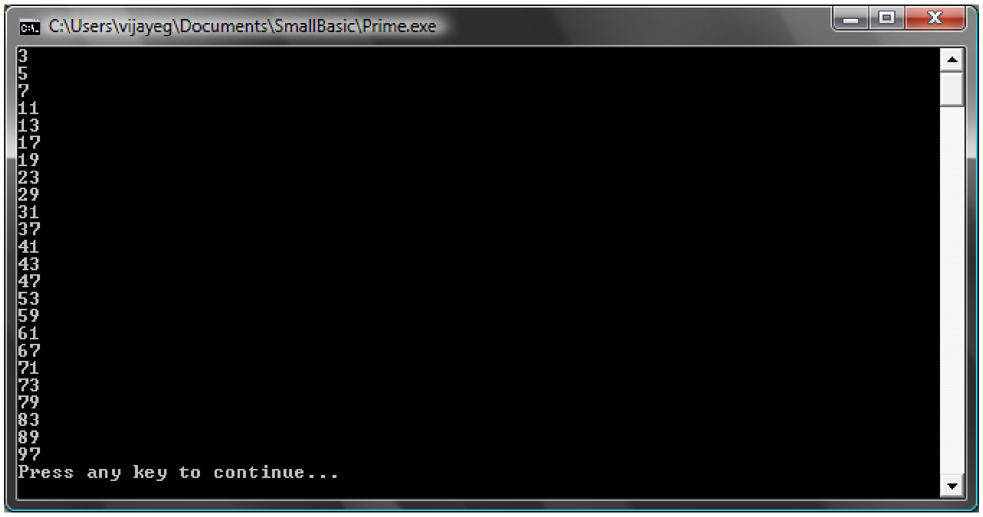
Figure 47 - Prime Check

Now that you have a subroutine that can do the Prime test for us, you might want to use this to list out all the prime numbers below, say, 100. It is really easy to modify the above program and make the call to *PrimeCheck* from inside a loop. This gives the subroutine a different value to compute each time the loop is run. Let’s see how this is done with the example below.

```
For i = 3 To 100
  isPrime = "True"
  PrimeCheck()
  If (isPrime = "True") Then
    TextWindow.WriteLine(i)
  EndIf
EndFor

Sub PrimeCheck
  For j = 2 To Math.SquareRoot(i)
    If (Math.Remainder(i, j) = 0) Then
      isPrime = "False"
      Goto EndLoop
    EndIf
  Endfor
EndLoop:
EndSub
```

In the program above, the value of *i* is updated every time the loop is run. Inside the loop, a call to the subroutine *PrimeCheck* is made. The subroutine *PrimeCheck* then takes the value of *i* and computes whether or not *i* is a prime number. This result is stored in the variable *isPrime* which is then accessed by the loop outside of the subroutine. The value of *i* is then printed if it turns out to be a prime number. And since the loop starts from 3 and goes up to 100, we get a list of all the prime numbers that are between 3 and 100. Below is the result of the program.



```
3
5
7
11
13
17
19
23
29
31
37
41
43
47
53
59
61
67
71
73
79
83
89
97
Press any key to continue...
```

Figure 48 - Prime Numbers

Chapter 10

Arrays

By now you must be well versed with how to use variables – after all you have come this far and you're still having fun, right?

Let's for a moment, revisit the first program we wrote with variables:

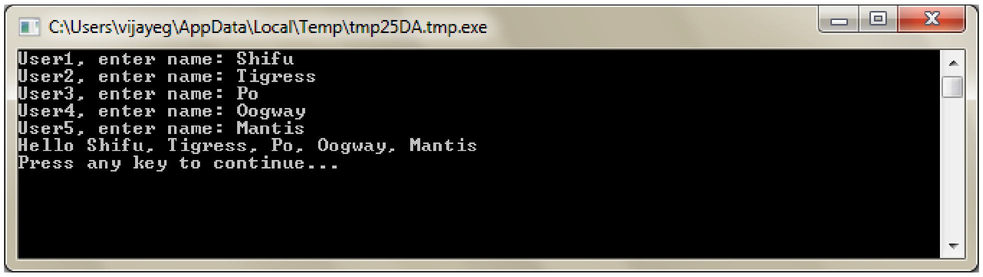
```
TextWindow.Write("Enter your Name: ")
name = TextWindow.Read()
TextWindow.WriteLine("Hello " + name)
```

In this program, we received and stored the name of the user in a variable called **name**. Then later we said "Hello" to the user. Now, let's say there is more than one user – say, there are 5 users. How would we store all their names? One way of doing this is:

```
TextWindow.Write("User1, enter name: ")
name1 = TextWindow.Read()
TextWindow.Write("User2, enter name: ")
name2 = TextWindow.Read()
TextWindow.Write("User3, enter name: ")
name3 = TextWindow.Read()
TextWindow.Write("User4, enter name: ")
name4 = TextWindow.Read()
TextWindow.Write("User5, enter name: ")
name5 = TextWindow.Read()
```

```
TextWindow.Write("Hello ")
TextWindow.Write(name1 + ", ")
TextWindow.Write(name2 + ", ")
TextWindow.Write(name3 + ", ")
TextWindow.Write(name4 + ", ")
TextWindow.WriteLine(name5)
```

When you run this you'll get the following result:



```
C:\Users\vijayeg\AppData\Local\Temp\tmp25DA.tmp.exe
User1, enter name: Shifu
User2, enter name: Tigress
User3, enter name: Po
User4, enter name: Oogway
User5, enter name: Mantis
Hello Shifu, Tigress, Po, Oogway, Mantis
Press any key to continue...
```

Figure 49 - Not using arrays

Clearly there must be a better way to write such a simple program, right? Especially since the computer is really good at doing repetitive tasks, why should we bother with writing the same code over and over for every new user? The trick here is to store and retrieve more than one user's name using the same variable. If we can do that then we can use a **For** loop we learned in earlier chapters. This is where arrays come to our help.

What is an array?

An array is a special kind of variable which can hold more than one value at a time. Basically, what it means is that instead of having to create **name1, name2, name3, name4** and **name5** in order to store five user names, we could just use **name** to store all five users' name. The way we store multiple values is by use of this thing called "index." For example, **name[1], name[2], name[3], name[4]** and **name[5]** can all store a value each. The numbers 1, 2, 3, 4 and 5 are called *indices* for the array.

Even though the **name[1]**, **name[2]**, **name[3]**, **name[4]** and **name[5]** all look like they are different variables, they're in reality all just one variable. And what's the advantage of this, you may ask. The best part of storing values in an array is that you can specify the index using another variable – which allows us to easily access arrays inside loops.

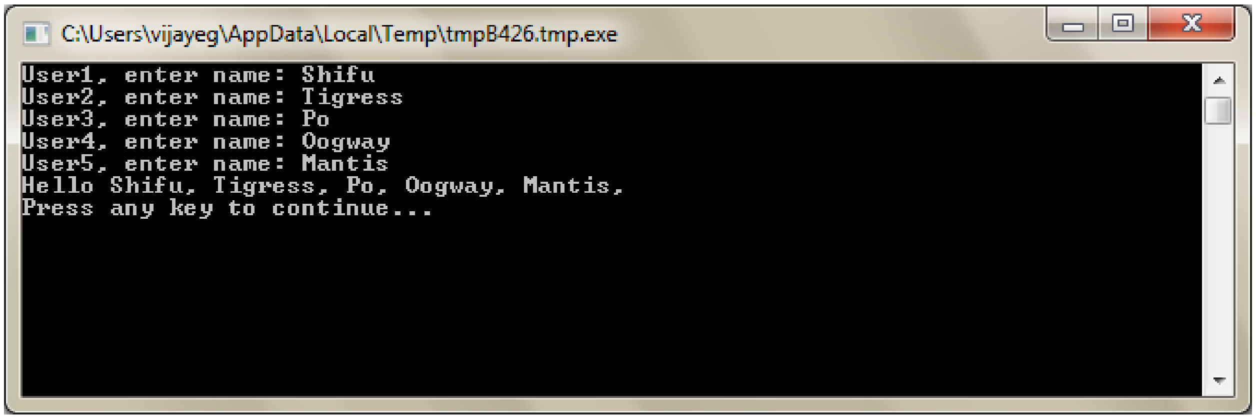
Now, let's look at how we can put our new knowledge to use by rewriting our previous program with arrays.

```
For i = 1 To 5
  TextWindow.Write("User" + i + ", enter name: ")
  name[i] = TextWindow.Read()
```

```
EndFor

TextWindow.Write("Hello ")
For i = 1 To 5
  TextWindow.Write(name[i] + ", ")
EndFor
TextWindow.WriteLine("")
```

Much easier to read, isn't it? Notice the two bolded lines. The first one stores a value in the array and the second one reads it from the array. The value you store in **name[1]** will not be affected by what you store in **name[2]**. Hence for most purposes you can treat **name[1]** and **name[2]** as two different variables with the same identity.



```
User1, enter name: Shifu
User2, enter name: Tigress
User3, enter name: Po
User4, enter name: Oogway
User5, enter name: Mantis
Hello Shifu, Tigress, Po, Oogway, Mantis,
Press any key to continue...
```

Figure 50 - Using arrays

The above program gives almost the exact same result as the one without arrays, except for the comma at the end of *Mantis*. We can fix that by rewriting the printing loop as:

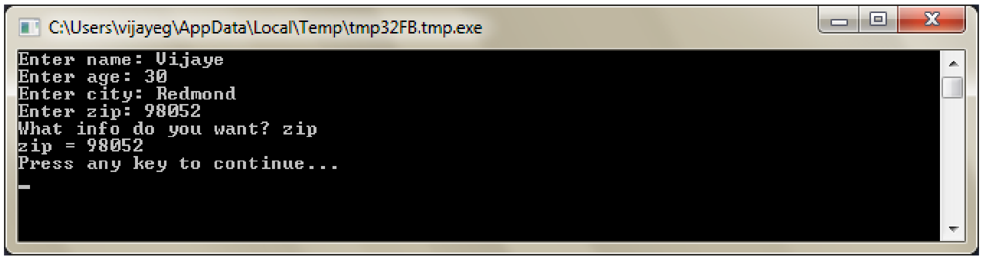
```
TextWindow.Write("Hello ")
For i = 1 To 5
  TextWindow.Write(name[i])
  If i < 5 Then
    TextWindow.Write(", ")
  EndIf
EndFor
TextWindow.WriteLine("")
```

Indexing an array

In our previous program you saw that we used numbers as indices to store and retrieve values from the array. It turns out that the indices are not restricted to just numbers and in practice it's very useful to use textual indices too. For example, in the following program, we ask and store various pieces of information about a user and then print out the info that the user asks for.

```
TextWindow.Write("Enter name: ")
user["name"] = TextWindow.Read()
TextWindow.Write("Enter age: ")
user["age"] = TextWindow.Read()
TextWindow.Write("Enter city: ")
user["city"] = TextWindow.Read()
TextWindow.Write("Enter zip: ")
user["zip"] = TextWindow.Read()

TextWindow.Write("What info do you want? ")
index = TextWindow.Read()
TextWindow.WriteLine(index + " = " + user[index])
```



C:\Users\vijayeg\AppData\Local\Temp\tmp32FB.tmp.exe

```
Enter name: Vijaye
Enter age: 30
Enter city: Redmond
Enter zip: 98052
What info do you want? zip
zip = 98052
Press any key to continue...
```

Figure 51 - Using non-numeric indices

More than one dimension

Let's say you want to store the name and phone number of all your friends and then be able to lookup on their phone numbers whenever you need – kinda like a phonebook. How would we go about writing such a program?

Array indices are not case sensitive. Just like regular variables, array indices match don't have to match the precise capitalization.

In this case, there are two sets of indices (also known as the array's dimension) involved. Assume we identify each friend by their nick name. This becomes our first index in the array. Once we use the first index to get our friend variable, the second of indices, **name** and **phone number** would help us get to the actual name and phone number of that friend.

The way we store this data would be like this:

```
friends["Rob"]["Name"] = "Robert"
friends["Rob"]["Phone"] = "555-6789"

friends["VJ"]["Name"] = "Vijaye"
friends["VJ"]["Phone"] = "555-4567"

friends["Ash"]["Name"] = "Ashley"
friends["Ash"]["Phone"] = "555-2345"
```

Since we have two indices on the same array, **friends**, this array is called a two dimensional array.

Once we have set this program up, we can then take as input the nickname of a friend and then print out the information we have stored about them. Here's the full program that does that:

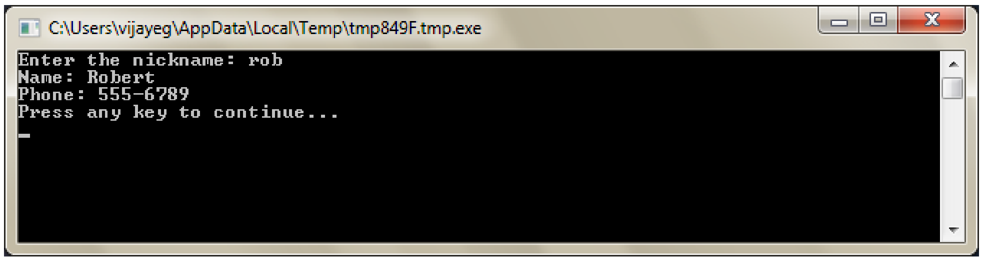
```
friends["Rob"]["Name"] = "Robert"
friends["Rob"]["Phone"] = "555-6789"

friends["VJ"]["Name"] = "Vijaye"
friends["VJ"]["Phone"] = "555-4567"

friends["Ash"]["Name"] = "Ashley"
friends["Ash"]["Phone"] = "555-2345"

TextWindow.Write("Enter the nickname: ")
nickname = TextWindow.Read()

TextWindow.WriteLine("Name: " + friends[nickname]["Name"])
TextWindow.WriteLine("Phone: " + friends[nickname]["Phone"])
```



```
Enter the nickname: rob
Name: Robert
Phone: 555-6789
Press any key to continue...
```

Figure 52 - A simple phone book

Using Arrays to represent grids

A very common use of multi-dimensional arrays is to represent grids/tables. Grids have rows and columns, which can fit nicely into a two dimensional array. A simple program that lays out boxes in a grid is given below:

```
rows = 8
columns = 8
size = 40

For r = 1 To rows
  For c = 1 To columns
    GraphicsWindow.BrushColor = GraphicsWindow.GetRandomColor()
    boxes[r][c] = Shapes.AddRectangle(size, size)
    Shapes.Move(boxes[r][c], c * size, r * size)
  EndFor
EndFor
```

This program adds rectangles and positions them to form an 8x8 grid. In addition to laying these boxes, it also stores these boxes in an array. Doing so makes it easy for us to keep track of these boxes and use them again when we need them.

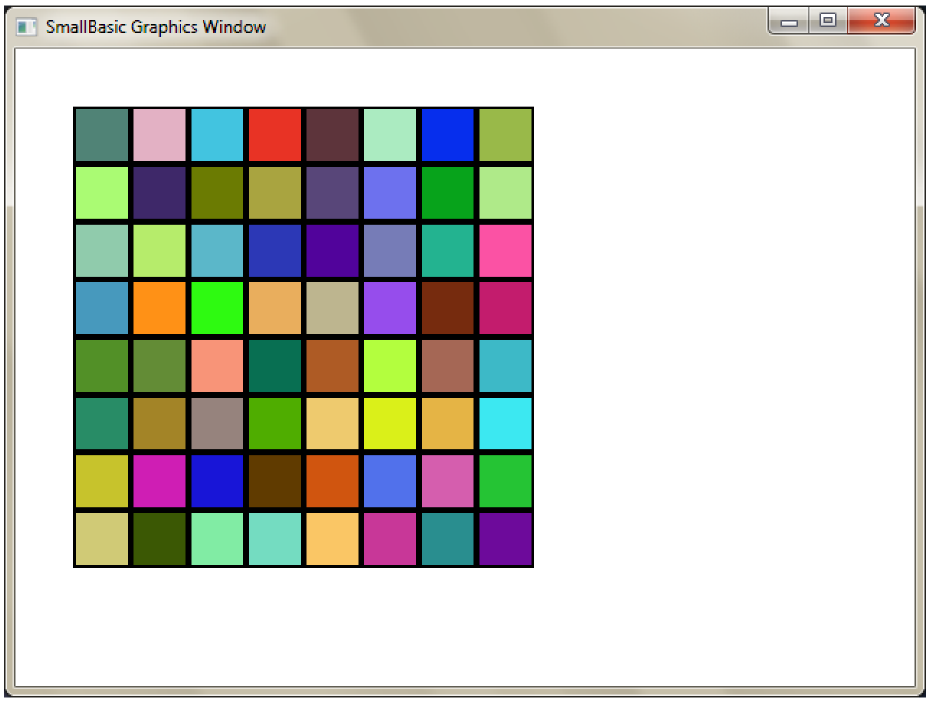


Figure 53 - Laying out boxes in a grid

For example, adding the following code to the end of the previous program would make these boxes animate to the top left corner.

```
For r = 1 To rows
  For c = 1 To columns
    Shapes.Animate(boxes[r][c], 0, 0, 1000)
    Program.Delay(300)
  EndFor
EndFor
```

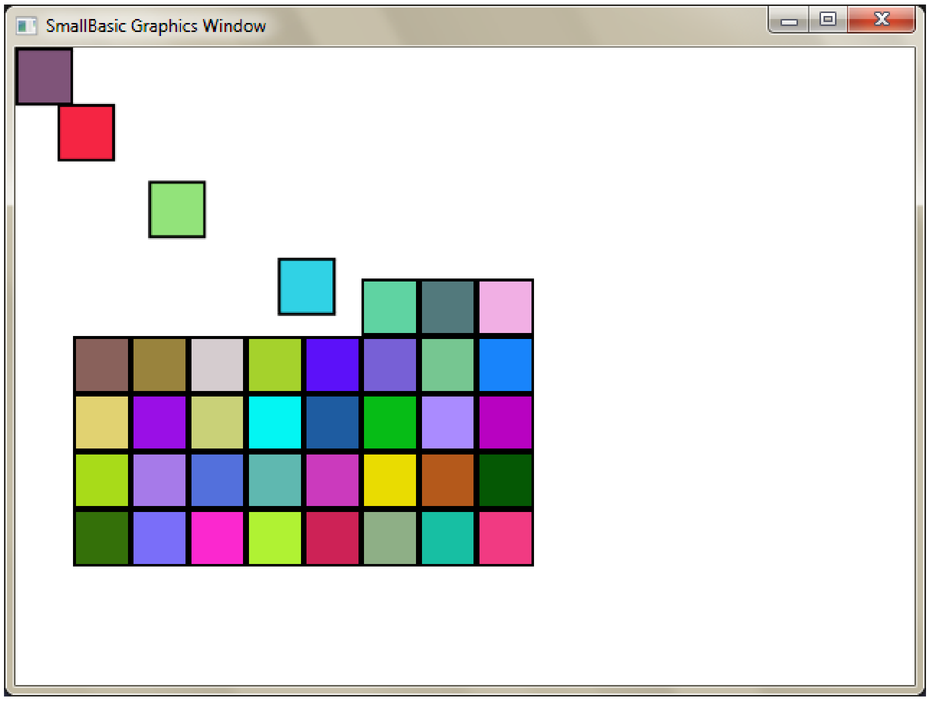


Figure 54 - Keeping track of boxes in the grid

Chapter 11

Events and Interactivity

In the first two chapters, we introduced objects that have *Properties* and *Operations*. In addition to properties and operations, some objects have what are called ***Events***. Events are like signals that are raised, for example, in response to user actions, like moving the mouse or clicking it. In some sense events are the opposite of operations. In the case of operation, you as a programmer call it to make the computer do something; whereas in the case of events, the computer lets you know when something interesting has happened.

How are events useful?

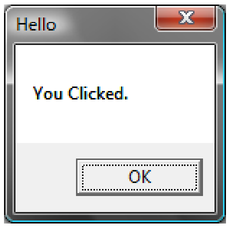
Events are central to introducing interactivity in a program. If you want to allow a user to interact with your program, events are what you'll use. Say, you're writing a Tic-Tac-Toe game. You'll want to allow the user to choose his/her play, right? That's where events come in - you receive user input from within your program using events. If this seems hard to grasp, don't worry, we'll take a look at a very simple example that will help you understand what events are and how they can be used.

Below is a very simple program that has just one statement and one subroutine. The subroutine uses the *ShowMessage* operation on the GraphicsWindow object to display a message box to the user.

```
GraphicsWindow.MouseDown = OnMouseDown

Sub OnMouseDown
  GraphicsWindow.ShowMessage("You Clicked.", "Hello")
EndSub
```

The interesting part to note in the program above is the line where we assign the subroutine name to the **MouseDown** event of GraphicsWindow object. You'll notice that MouseDown looks very much like a property – except that instead of assigning some value, we're assigning the subroutine *OnMouseDown* to it. That's what is special about events – when the event happens, the subroutine is called automatically. In this case, the subroutine *OnMouseDown* is called every time the user clicks using the mouse, on the GraphicsWindow. Go ahead, run the program and try it out. Anytime you click on the GraphicsWindow with your mouse, you'll see a message box just like the one shown in the picture below.



You Clicked.

Figure 55 - Response to an event

This kind of event handling is very powerful and allows for very creative and interesting programs. Programs written in this fashion are often called event-driven programs.

You can modify the *OnMouseDown* subroutine to do other things than popup a message box. For instance, like in the program below, you can draw big blue dots where the user clicks the mouse.

```
GraphicsWindow.BrushColor = "Blue"
GraphicsWindow.MouseDown = OnMouseDown

Sub OnMouseDown
  x = GraphicsWindow.MouseX - 10
  y = GraphicsWindow.MouseY - 10
  GraphicsWindow.FillEllipse(x, y, 20, 20)
EndSub
```

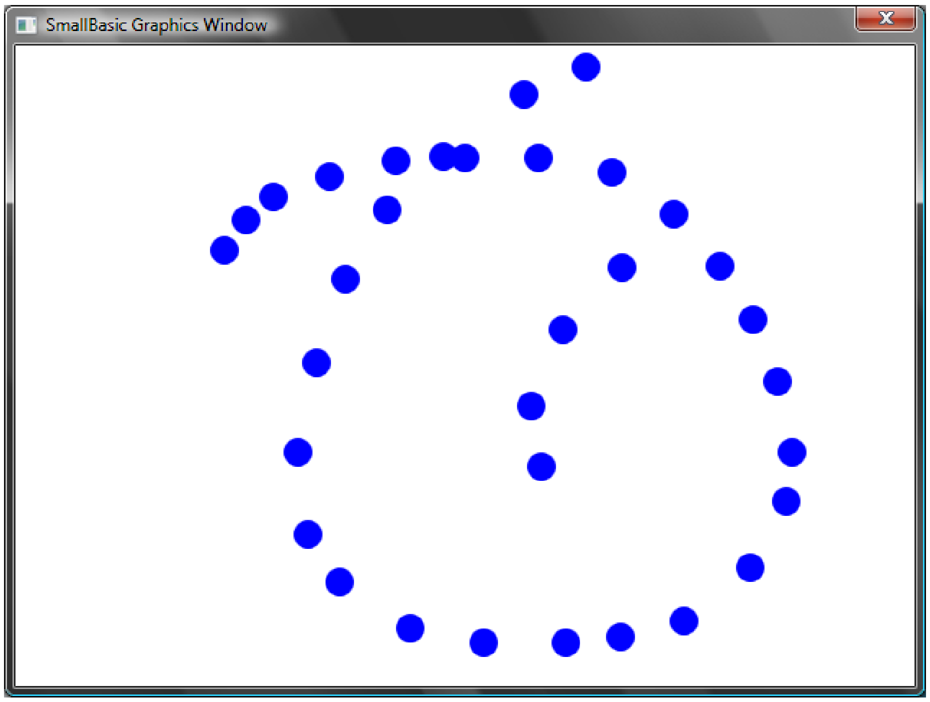


Figure 56 - Handling Mouse Down Event

Notice that in the program above, we used *MouseX* and *MouseY* to get the mouse co-ordinates. We then use this to draw a circle using the mouse co-ordinates as the center of the circle.

Handling multiple events

There are really no limits to how many events you want to handle. You can even have one subroutine handle multiple events. However, you can handle an event only once. If you try to assign two subroutines to the same event, the second one wins.

To illustrate this, let's take the previous example and add a subroutine that handles key presses. Also, let's make this new subroutine change the color of the brush, so that when you click your mouse, you'll get a different colored dot.

```
GraphicsWindow.BrushColor = "Blue"
GraphicsWindow.MouseDown = OnMouseDown
GraphicsWindow.KeyDown = OnKeyDown

Sub OnKeyDown
  GraphicsWindow.BrushColor = GraphicsWindow.GetRandomColor()
EndSub

Sub OnMouseDown
  x = GraphicsWindow.MouseX - 10
  y = GraphicsWindow.MouseY - 10
  GraphicsWindow.FillEllipse(x, y, 20, 20)
EndSub
```

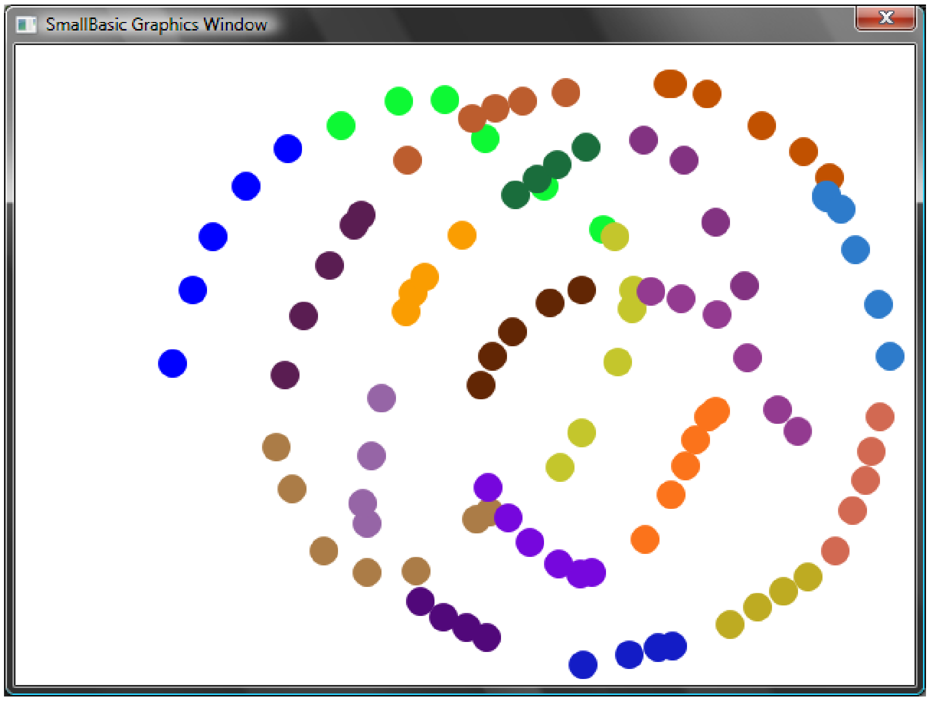


Figure 57 - Handling multiple events

If you ran this program and clicked on the window, you'll get a blue dot. Now, if you press any key once and click again, you'll get a different colored dot. What's happening when you press a key is that the subroutine *OnKeyDown* gets executed which changes the brush color to a random color. After that when you click the mouse, a circle is drawn using the newly set color – giving the random color dots.

A paint program

Armed with events and subroutines, we can now write a program that lets users draw on the window. It's surprisingly easy to write such a program, provided we break down the problem into smaller bits. As a first step, let's write a program that will allow users to move the mouse anywhere on the graphics window, leaving a trail wherever they move the mouse.

```
GraphicsWindow.MouseMove = OnMouseMove

Sub OnMouseMove
  x = GraphicsWindow.MouseX
  y = GraphicsWindow.MouseY
  GraphicsWindow.DrawLine(prevX, prevY, x, y)
  prevX = x
  prevY = y
EndSub
```

However, when you run this program, the first line always starts from the top left edge of the window (0, 0). We can fix this problem by handling the *MouseDown* event and capture the *prevX* and *prevY* values when that event comes.

Also, we really only need the trail when the user has the mouse button down. Other times, we shouldn't draw the line. In order to get this behavior, we'll use the *IsLeftButtonDown* property on the **Mouse** object. This property tells whether the Left button is being held down or not. If this value is true, then we'll draw the line, if not we'll skip the line.

```
GraphicsWindow.MouseMove = OnMouseMove
GraphicsWindow.MouseDown = OnMouseDown

Sub OnMouseDown
  prevX = GraphicsWindow.MouseX
  prevY = GraphicsWindow.MouseY
EndSub

Sub OnMouseMove
  x = GraphicsWindow.MouseX
  y = GraphicsWindow.MouseY
  If (Mouse.IsLeftButtonDown) Then
    GraphicsWindow.DrawLine(prevX, prevY, x, y)
  EndIf
  prevX = x
  prevY = y
EndSub
```

Appendix A

Fun Samples

Turtle Fractal

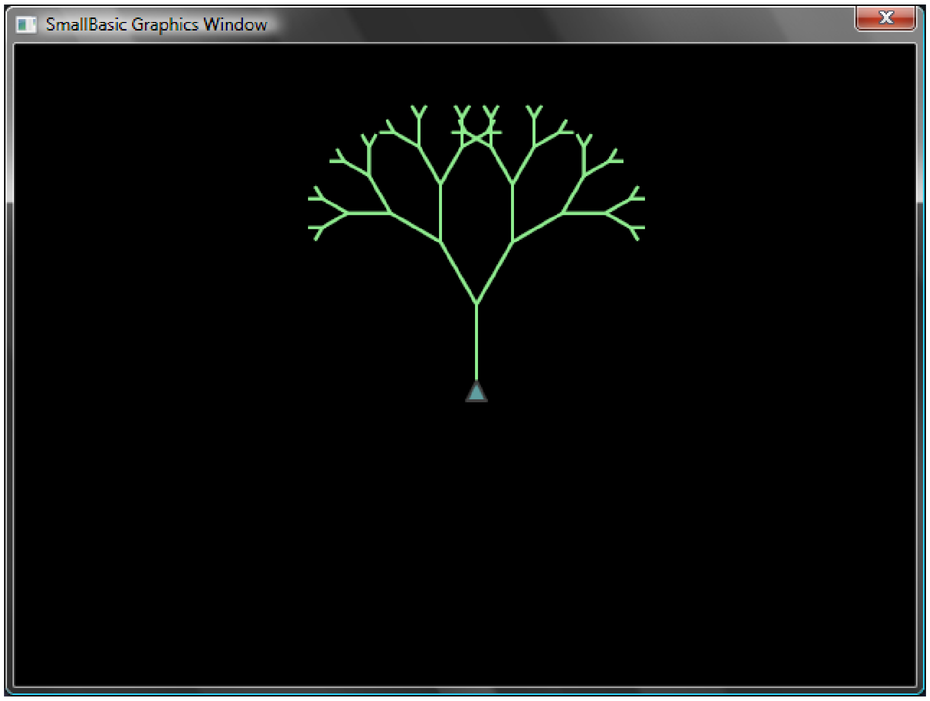


Figure 58 - Turtle drawing a tree fractal

```
angle = 30
delta = 10
distance = 60
Turtle.Speed = 9
```

```
GraphicsWindow.BackgroundColor = "Black"
GraphicsWindow.PenColor = "LightGreen"
DrawTree()

Sub DrawTree
  If (distance > 0) Then
    Turtle.Move(distance)
    Turtle.Turn(angle)

    Stack.PushValue("distance", distance)
    distance = distance - delta
    DrawTree()
    Turtle.Turn(-angle * 2)
    DrawTree()
    Turtle.Turn(angle)
    distance = Stack.PopValue("distance")

    Turtle.Move(-distance)
  EndIf
EndSub
```

Photos from Flickr

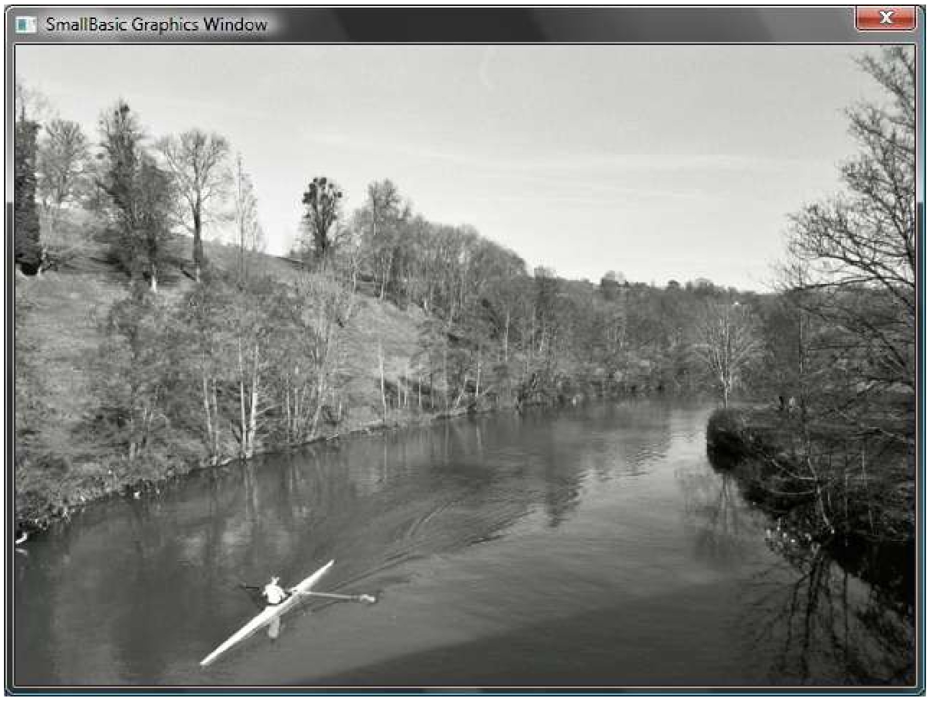


Figure 59 - Retrieving pictures from Flickr

```
GraphicsWindow.BackgroundColor = "Black"
GraphicsWindow.MouseDown = OnMouseDown

Sub OnMouseDown
    pic = Flickr.GetRandomPicture("mountains, river")
    GraphicsWindow.DrawResizedImage(pic, 0, 0, 640, 480)
EndSub
```

Dynamic Desktop Wallpaper

```
For i = 1 To 10
  pic = Flickr.GetRandomPicture("mountains")
  Desktop.SetWallPaper(pic)
  Program.Delay(10000)
EndFor
```

Paddle Game

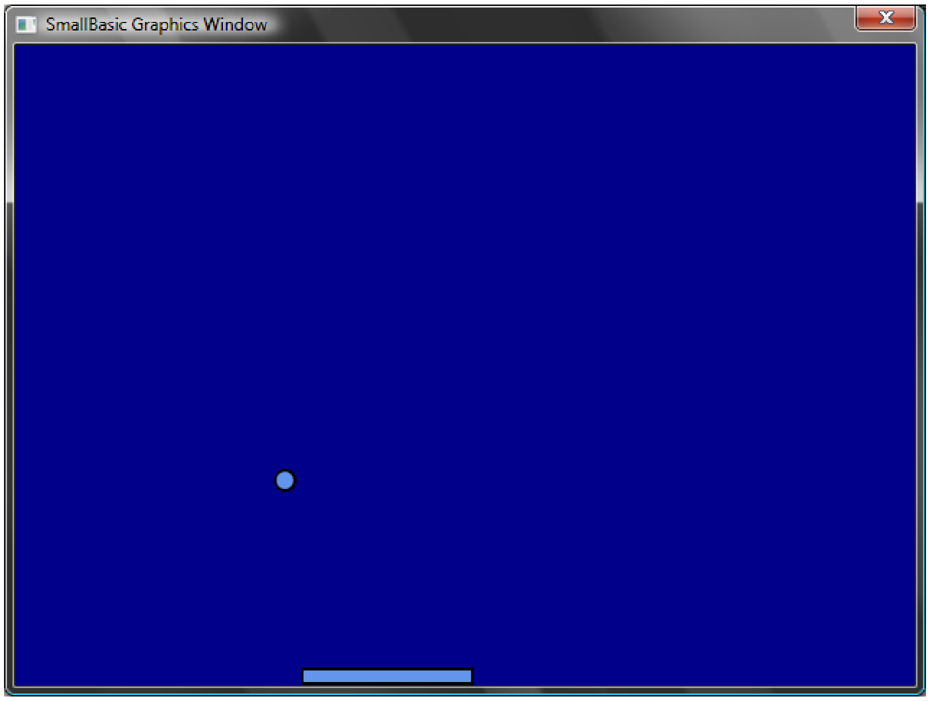


Figure 60 - Paddle Game

```
GraphicsWindow.BackgroundColor = "DarkBlue"
paddle = Shapes.AddRectangle(120, 12)
ball = Shapes.AddEllipse(16, 16)
```

```
GraphicsWindow.MouseMove = OnMouseMove

x = 0
y = 0
deltaX = 1
deltaY = 1

RunLoop:
  x = x + deltaX
  y = y + deltaY

  gw = GraphicsWindow.Width
  gh = GraphicsWindow.Height
  If (x >= gw - 16 or x <= 0) Then
    deltaX = -deltaX
  EndIf
  If (y <= 0) Then
    deltaY = -deltaY
  EndIf

  padX = Shapes.GetLeft (paddle)
  If (y = gh - 28 and x >= padX and x <= padX + 120) Then
    deltaY = -deltaY
  EndIf

  Shapes.Move(ball, x, y)
  Program.Delay(5)

  If (y < gh) Then
    Goto RunLoop
  EndIf

GraphicsWindow.ShowMessage("You Lose", "Paddle")

Sub OnMouseMove
  paddleX = GraphicsWindow.MouseX
  Shapes.Move(paddle, paddleX - 60, GraphicsWindow.Height - 12)
EndSub
```

Colors

Here's a list of named colors supported by Small Basic, categorized by their base hue.

Red Colors

IndianRed	#CD5C5C
LightCoral	#F08080
Salmon	#FA8072
DarkSalmon	#E9967A
LightSalmon	#FFA07A
Crimson	#DC143C
Red	#FF0000
FireBrick	#B22222
DarkRed	#8B0000

Pink Colors

Pink	#FFC0CB
LightPink	#FFB6C1
HotPink	#FF69B4
DeepPink	#FF1493
MediumVioletRed	#C71585
PaleVioletRed	#DB7093

Orange Colors

LightSalmon	#FFA07A
Coral	#FF7F50
Tomato	#FF6347
OrangeRed	#FF4500
DarkOrange	#FF8C00
Orange	#FFA500

Yellow Colors

Gold	#FFD700
Yellow	#FFFF00
LightYellow	#FFFFE0
LemonChiffon	#FFFACD
LightGoldenrodYellow	#FAFAD2
PapayaWhip	#FFEFD5
Moccasin	#FFE4B5
PeachPuff	#FFDAB9

PaleGoldenrod	#EEE8AA
Khaki	#F0E68C
DarkKhaki	#BDB76B

Purple Colors

Lavender	#E6E6FA
Thistle	#D8BFD8
Plum	#DDA0DD
Violet	#EE82EE
Orchid	#DA70D6
Fuchsia	#FF00FF
Magenta	#FF00FF
MediumOrchid	#BA55D3
MediumPurple	#9370DB
BlueViolet	#8A2BE2
DarkViolet	#9400D3
DarkOrchid	#9932CC
DarkMagenta	#8B008B
Purple	#800080
Indigo	#4B0082
SlateBlue	#6A5ACD
DarkSlateBlue	#483D8B
MediumSlateBlue	#7B68EE

Green Colors

GreenYellow	#ADFF2F
Chartreuse	#7FFF00
LawnGreen	#7CFC00
Lime	#00FF00
LimeGreen	#32CD32
PaleGreen	#98FB98
LightGreen	#90EE90

MediumSpringGreen	#00FA9A
SpringGreen	#00FF7F
MediumSeaGreen	#3CB371
SeaGreen	#2E8B57
ForestGreen	#228B22
Green	#008000
DarkGreen	#006400
YellowGreen	#9ACD32
OliveDrab	#6B8E23
Olive	#808000
DarkOliveGreen	#556B2F
MediumAquamarine	#66CDAA
DarkSeaGreen	#8FBC8F
LightSeaGreen	#20B2AA
DarkCyan	#008B8B
Teal	#008080

Blue Colors

Aqua	#00FFFF
Cyan	#00FFFF
LightCyan	#E0FFFF
PaleTurquoise	#AFEEEE
Aquamarine	#7FFFD4
Turquoise	#40E0D0
MediumTurquoise	#48D1CC
DarkTurquoise	#00CED1
CadetBlue	#5F9EA0
SteelBlue	#4682B4
LightSteelBlue	#B0C4DE
PowderBlue	#B0E0E6
LightBlue	#ADD8E6
SkyBlue	#87CEEB

LightSkyBlue	#87CEFA
DeepSkyBlue	#00BFFF
DodgerBlue	#1E90FF
CornflowerBlue	#6495ED
MediumSlateBlue	#7B68EE
RoyalBlue	#4169E1
Blue	#0000FF
MediumBlue	#0000CD
DarkBlue	#00008B
Navy	#000080
MidnightBlue	#191970

Brown Colors

Cornsilk	#FFF8DC
BlanchedAlmond	#FFEBCD
Bisque	#FFE4C4
NavajoWhite	#FFDEAD
Wheat	#F5DEB3
BurlyWood	#DEB887
Tan	#D2B48C
RosyBrown	#BC8F8F
SandyBrown	#F4A460
Goldenrod	#DAA520
DarkGoldenrod	#B8860B
Peru	#CD853F
Chocolate	#D2691E
SaddleBrown	#8B4513
Sienna	#A0522D
Brown	#A52A2A
Maroon	#800000

White Colors

White	#FFFFFF
Snow	#FFFAFA
Honeydew	#F0FFF0
MintCream	#F5FFFA
Azure	#F0FFFF
AliceBlue	#F0F8FF
GhostWhite	#F8F8FF
WhiteSmoke	#F5F5F5
Seashell	#FFF5EE
Beige	#F5F5DC
OldLace	#FDF5E6
FloralWhite	#FFFAF0
Ivory	#FFFFF0
AntiqueWhite	#FAEBD7
Linen	#FAF0E6
LavenderBlush	#FFF0F5
MistyRose	#FFE4E1

Gray Colors

Gainsboro	#DCDCDC
LightGray	#D3D3D3
Silver	#C0C0C0
DarkGray	#A9A9A9
Gray	#808080
DimGray	#696969
LightSlateGray	#778899
SlateGray	#708090
DarkSlateGray	#2F4F4F
Black	#000000