A Quick Journey Through Small Basic

By Stephen DeVoy

Forward

The author of this booklet is currently working on a much larger text of which this booklet is a single chapter. The work in progress is an introductory programming course that uses a sequence of short forays into increasingly sophisticated programming languages as a means to accelerate the learning process for students with an innate talent to program. Within this larger book, this chapter is the first of several that quickly dive into and out of a specific language, not with the intent of making students experts in the language, but in an effort to learn all that is worth learning from the language before moving onto another language that has other goodies to offer. The hope is that by the time the student has reached the end of the book, she will have not only learned how to program, but will understand that there are many different ways of accomplishing any task, that there are many different languages to choose from, and the advantages and disadvantages of different language types. We don't teach a carpenter to use only a hammer for the first six months of his apprenticeship, do we? No, we teach him about many tools and he gradually becomes better and better in using all of them.

> *"I suppose it is tempting, if the only tool you have is a hammer, to treat everything as if it were a nail." - Abraham Maslow*

When we were children, didn't we all want to take the training wheels off of our bicycles as quickly as possible? Many programming languages begin their lives as an attempt to make teaching computer programming easier. Later, they become major programming languages, not because they are good, but because a lot of students know them well and know others not so well. Why not challenge students to move on and take off the training wheels as quickly as possible?

Many programming languages force programmers to use the wrong strategies to solve certain problem sets. Would it not be better to teach a student to choose the correct language for solving a problem instead of teaching her how to poorly solve a problem in a language inadequate to the task?

Despite being part of a large work, this chapter stands on its own and may be useful to teachers wishing to learn Small Basic programming for the purpose of teaching their own students how to program. It may also be useful for students who are annoyed by learning only by example. I know that I have always preferred to read about something and then dive in on my own. Likewise, I've become bored to tears with books that ask the learner to follow tedious steps for the sole purpose of mimicking the learning process. For that reason, this booklet focuses more on the language than on examples. Later chapters of the book in which this booklet is embedded will address deeper topics along the way, such as algorithms, data structures, and discrete mathematics.

About the Author

Stephen DeVoy holds a bachelor degree in computer science, a bachelor degree in philosophy, and a CELTA certificate for teaching English to speakers of foreign languages. He has worked as a software engineer and software developer for most of the past 38 years. He has tutored students in computer science as a side profession for more than 40 years. Students seeking tutoring may contact Stephen DeVoy by email at steve@stephendevoy.com

A Quick Journey Through Small Basic

This book utilizes Small Basic, not as an end, but as a springboard. The intent behind Small Basic has been to teach children and others about programming. It is designed for use in education only. Small Basic is not designed to be a general purpose programming language, and few are expected to make a career out of programming in Small Basic. The people that would benefit most from becoming experts in Small Basic are teachers and employees of Microsoft responsible for Small Basic. It is a very good tool for teaching fundamental programming concepts to students without any prior programming experience. I believe that this chapter of my book will be of great use to those teaching pre-university level students, and for this reason, I am releasing this chapter on Small Basic to the public. Many schools are under-funded and I happily invite them to use this chapter of my book freely and without compensation. My only condition is that this chapter neither be modified nor sold.

The purpose of this book, of which this is only a single chapter, is to teach adults to program and to use Small Basic in the same way that a swimmer uses her toes to test the water of a river or pond. It is good to test your desire, will, and ability to succeed at something, before you commit all of your resources and dive in. As Deepak Malhotra, a Professor in the Negotiations, Organizations and Markets Unit at the Harvard Business School has said, “Quit early. Quit often. Be the best quitter you know.”

As my book employs Smart Basic as a tool for self evaluation - a litmus test for a reader to determine if a career in programming best fits their talents - and as a way of gently introducing fundamental programming concepts before transitioning to progressively more difficult languages in the chapters that follow; this chapter progresses quickly, covers the language, shows examples, encourages the reader to write some code and overcome some challenges, and then moves on - abandoning Smart Basic in favor of more useful languages. I use Smart Basic to prime the student for more and not for any other reason.

The Buddha's parable of the raft poses a situation where a monk, in need of crossing a river to get to his destination, is forced to build a raft. Building the raft requires great effort. Upon crossing the river he is left with the dilemma of choosing to take the raft with him, thus not abandoning the work he invested in creating it, or discarding the raft, and traveling unburdened to his destination. The parable is a metaphor for methodology and enlightenment. Once a methodology or practice enlightens you, why hold on to it? As someone enlightened, abandon the methodology that enlightened you and move forward and seek the next level of enlightenment using the methodology best for achieving that goal. In this book, Small Basic is the first raft. We use it until there is little more use for it and then move on.

Things That Exist in Small Basic
And What You Can Do With Them

Literals and Variables

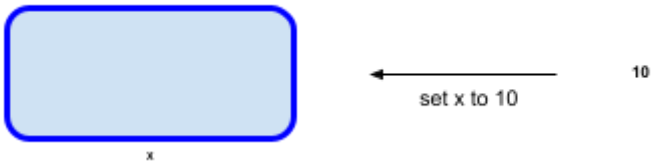
Every language has nouns and computer programming languages are no exception. Small Basic nouns include literals and variables. A literal is something that is literally what it appears to be. For example, the number 10 is a literal. It is a sequence of digits representing the number 10. It is important to distinguish between the sequence of digits representing the number 10 and the number 10 itself. The number 10 is a concept that exists independently of any programming language, or book, or chalkboard. The literal 10 represents that number in the Arabic numeral system. In Small Basic, the literal 10 denotes a value described by the digits 1 and 0 written sequentially with no spaces between them and no other digits immediately before or after them. The literal 10 may be added to something, subtracted from something, multiplied by something, or divided by something. The something at may combined with the literal 10 is always something that has a numerical value. The literal 10 may also be compared to other nouns, provided those nouns have a numeric value.

Another kind of noun, in Small Basic, is the variable. A Small Basic variable is the name of a place where a value may be stored. Similar to names of people, places, and things, the name of a variable is a sequence of characters, always beginning with a letter from the alphabet. Aside from the first letter of the name of a variable, the other characters making up its name can be other alphabetic characters or digits. We refer to such sequences of characters as alphanumerics. While a variable has a name, the relationship between a variable's name, and the value stored within the place it names, it is not exactly the same as the relationship between a person's name and a person. People have names. Names do not have people. In the case of a Small Basic variable, the name has a location and the location has a value. This distinction will be readdressed later in this chapter. The more concise term for a variable's name is "identifier". From this point forward, we will refer to names for a variable as identifiers.

Consider the drawing below:

On the left, we have a variable. The variable's identifier is **x**. The blue box represents the location that **x** identifies. On the right, we have a literal. In this case, the literal **10**.

Often, we need to store a value into a variable so that it can be acted upon by the program. While literals never change in value, the value stored in the location identified by a variable can change. In fact, that is why Small Basic variables exist. They exist to hold values. Let us suppose that we wish to place the value referenced by the literal **10** and store it into the location identified by **x**. We can do this by setting the value of **x** to the value of **10**.



Our desire, denoted in the figure above, is to take a copy of the value of the literal **10** and place it into the location identified by variable **x**. This kind of transfer of a value into the location identified by a variable is called an assignment. We are assigning the value denoted by **10** to the location denoted by **x**. The common term to use, among programmers, is the verb "to set". We simplify the complexity of explaining this kind of operation to the simple statement: set **x** to **10**. In Small Basic, a variable is set using the equals sign "=". Below is a single line of Small Basic code setting **x** to **10**.

```
x = 10
```

If I were to read this line of code outloud in English, I would say, "set x to 10." Alternatively, programmers may also say, "x gets 10". "set x to 10" and "x gets 10" have the same

meaning and both describe the assignment of the value denoted by **10** to the location identified by **x**.

It is important to understand that the equals sign, in this context, does not mean that **x** equals **10**. The mathematical notion of "x = 10" means that **x** is **10** throughout an equation, not just in some places within the equation. When I solve a mathematical equation for **x**, it is solved. **x** doesn't change, it is discovered. The small basic assignment "**x = 10**" is not an equation. It is an action to be performed. "**=**" is an operator and it performs an action. Operators are one kind of verb found in procedural programming languages.

After we set **x** to **10**, the state of the world becomes what is shown in the diagram below.

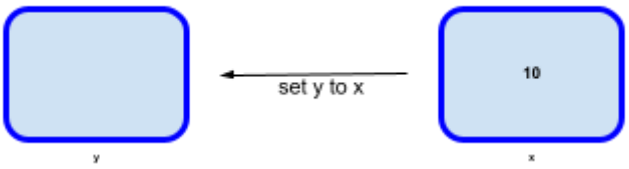
Notice, the literal **10** has not moved. It continues to exist. The contents of the location identified by **x** has changed. It now contains a copy of the value of the literal **10**. **10** hasn't moved. **x** still identifies the same location (the blue box). All that has changed is what is stored in that location. The stored value is now **10**.

I used very precise language to describe variables, literals, and locations named by variables in order to show that things are more complicated than statements like "x gets 10" imply. I do not want to mislead you into believing that things are actually that simple because later, when we explore other programming languages, we will find that the distinctions I have made here are important, despite the fact that they are not so important in Small Basic. In order to make reading easier, I now assume you understand the true underlying nature of the statement "x gets 10" and I will begin using phrases such as "x gets 10" without explaining all of the details behind the assignment. While discussing Small Basic, the assignment will no longer be described as "a copy of the value denoted by the literal 10 is placed into the location referred to by the identifier x." We will just say "x gets 10" or "set x to 10". Likewise, instead of saying, "the value stored in the location identified

by x is 10", we will say, "the value of X is 10". When we say, "the value of X is 10", we don't not mean that x will always have a value of 10, we mean that x, at this moment, has the value of 10 because if we set x to another value later, the value of x will be this new value.

Assignments are not only between a variable and a literal. Once a variable has a value it may become the source of an assignment to another variable or even itself.

In our example above, the value of **x** is **10**. If we were to assign a value to another variable, and that value were to come from **x**, then **x** could replace **10** in the assignment. The figure below represents our intent to set **y** to **x**.



After performing this assignment, **y** contains a copy of the value of **x**.

Now both **x** and **y** have a copy of the value of **10**. When speaking of this assignment, a programmer may say, "y gets x" or "set y to x". A programmer may also say, **y** is **10** or **y** contains **10**. In the context of Small Basic, these English statements are shorthand for the scenario described above.

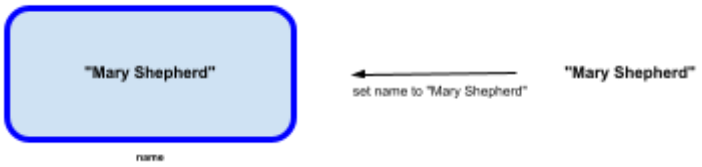
The Small Basic statement that accomplishes this state of affairs is:

y = x

Once again, this does not mean that **x** and **y** are the same thing or that they refer to the same locations in memory. It simply means that after the statement is executed, **y** will contain the same value that **x** contained at the moment of execution.

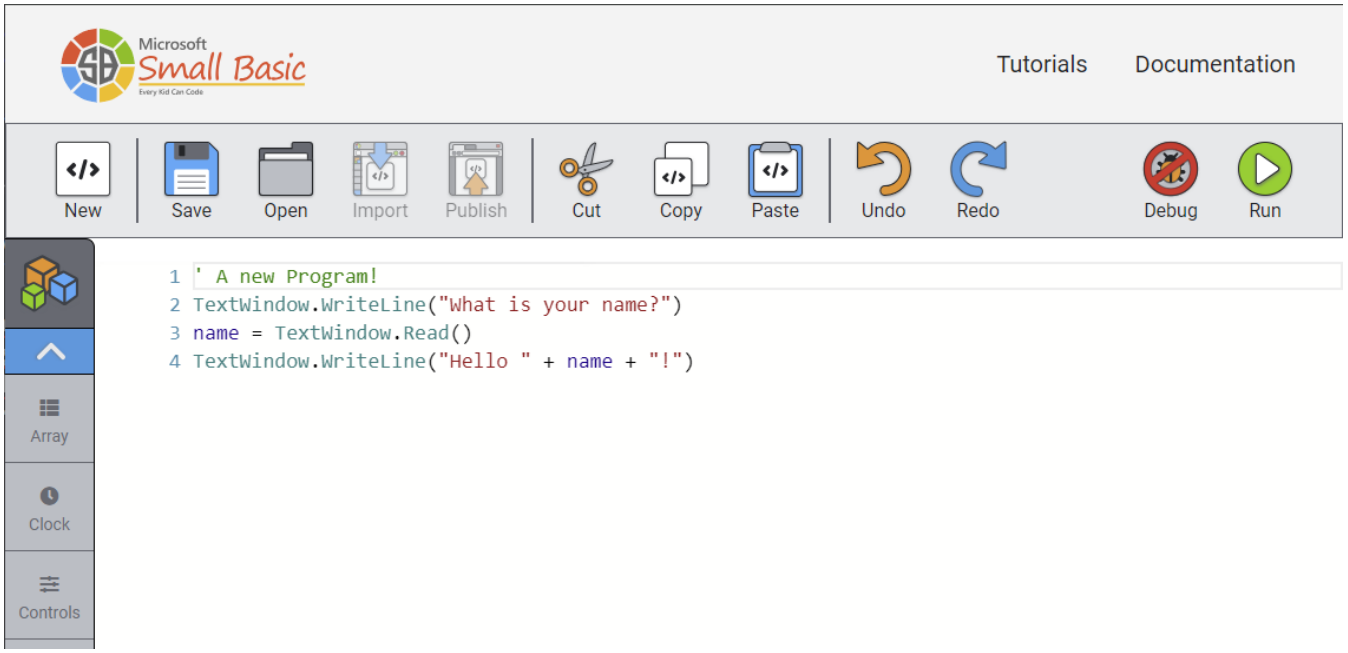
We've seen the distinction between a literal and a variable, let's think about another kind of literal that we may need. Numbers are not the only kind of data that computers process. Computers also process text. Text can be represented as a literal by placing it between double quotation marks. A person's name is an example of where text is needed.

The diagram above shows an unassigned variable on the left and a string literal on the right. The string literal is a sequence of characters making up the name Doug Lenat. We may assign this string literal to the variable.



In this example, in the interest of saving time and space, we've combined the assignment diagram with the result diagram. After we execute the assignment, the variable **name** contains a copy of the string **"Doug Lenat"**.

Small Basic's Default Web Portal Includes an Example of a Variable



Upon visiting Small Basic's web portal, you will be greeted with a code editor with code already inserted. This is Small Basic's substitute for the "Hello World!" program that most introductory programming documents provide as an example of a simple program.

The first line, which reads >>`' A new Program!` <<, is a comment line. It is not part of the Small Basic language. A comment begins with a single quote and terminates at the end of the line. Everything between the single quote and the end of the line is ignored by Small Basic. This enables programmers to explain to readers information about the code that may be important to readers, but not to Small Basic.

In the previous section, we read about numbers, strings, and variables. Can you spot the string in the second line of the program? Strings begin and end with double quotation marks. >>`"What is your name?"`<< is the string you were looking for. At this end of our Journey Through Small Basic, we will take a closer look at objects. On line 2, this sample program uses an object to send output to the screen. The object's name is TextWindow. Next to TextWindow, separated by a dot, we see WriteLine. WriteLine belongs to TextWindow. The dot between TextWindow and WriteLine shows that WriteLine is part of TextWindow. When using an object in Small Basic, all interactions with the object are conducted by these parts that belong to the object. We call these parts either methods or properties. Methods perform actions. Think of them as the arms and legs of an object. WriteLine is a method. It is the part of TextWindow that writes strings of text to a text window. There are no properties used in this example, so we will talk about properties later.

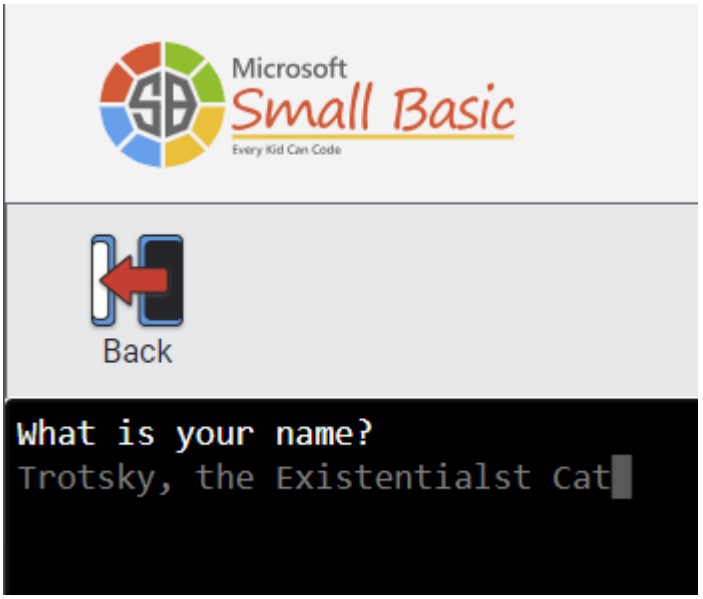
Line 3 of the example features a variable, the object TextWindow, and another method; Read. The Read method of TextWindow sits around waiting for you to type something on the keyboard and then hit enter. As soon as you hit enter, it returns a string to your Small Basic program. The string contains the characters that you typed, up to the enter key, but not including the enter key. The variable name is on the left hand side of an assignment statement. The text string returned by the Read method is assigned to the variable name.

Line 4 looks a lot like line 2. The difference is found between the parentheses following the WriteLine method. Within these parentheses, there is a string, a plus operator, a variable name, another plus operator and another string. The plus operators "add" the string inside of the name variable to the strings, forming a new string. If the user gave the name "Bob" as his name, this new string will be "Hello Bob!"

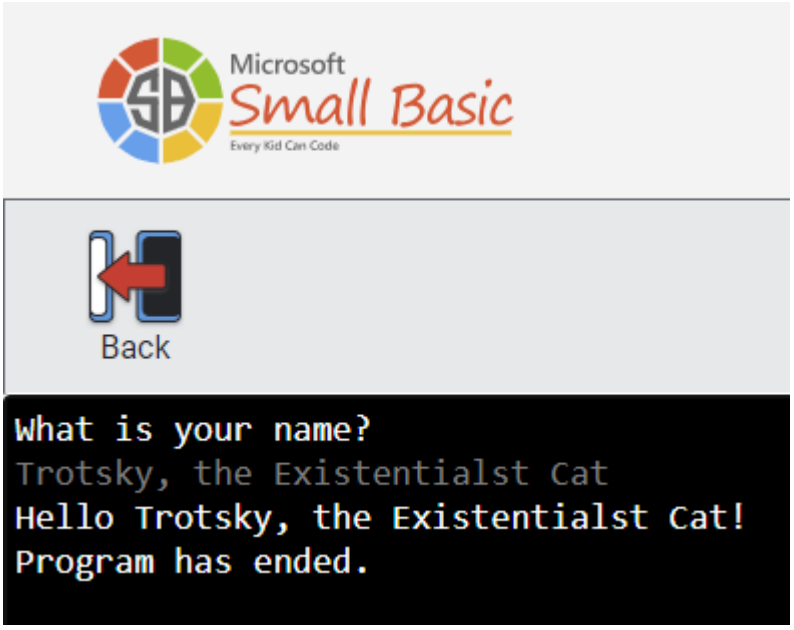
On the Small Basic web portal page, there is a round green button labeled "run". If you click that button, Small Basic will run the program.

TextWindow's WriteLine method will output the text "What is your name?" to a text window.

TextWindow's Read method will sit around waiting for you to type something and hit enter. Type "Trotsky, the Existentialist Cat", and then hit enter.



TextWindow will respond by displaying the response, crafted by line 4 of the program.

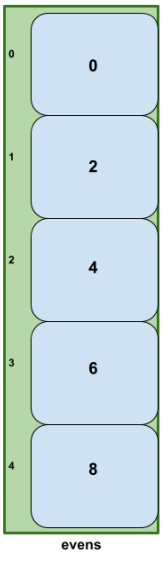


Click on the "Back" button to return to the program editing screen.

Arrays

Arrays are a means to store multiple values together. You can think of an array as a variable with multiple locations to store things. Each storage location within a Small Basic array has an index. An index organizes different items stored in the array similar to how a library may organize information about books within a card catalog. Each card in a card catalog has a label. The label is analogous to an array index. The information stored on any card is analogous to the information stored in an array at that index. Like a card catalog, the labels can be text or they can be numbers. Within a card catalog, there should be no two cards with identical labels. Identical labels would make it impossible to uniquely identify a card.

Let us suppose that we wish to store the first five even whole numbers into an array. With Small Basic, every variable is potentially an array. What transforms a variable into an array is the act of using the variable as an array. Below, on the left, is an example of Small Basic code assigning values to an array. To the right of the code, is a diagram showing the structure of the array within the variable **evens**.

evens[0] = 0 **evens[1] = 2** **evens[2] = 4** **evens[3] = 6** **evens[4] = 8**	

The variable **evens** contains an array that associates each index with a value. The values stored in an array are called elements. The element at index **3** of array **evens** is **6**. The element at index **1** of array **evens** is **2**. The array was not created until we used it. Prior to the statement **evens[0] = 0**, the **evens** array did not exist. Each time we added a new element at a new index to the array, the array grew in size.

Values can be stored in an array or retrieved from an array. To retrieve a value from array evens, use it in an expression or on the right hand side of an assignment. For example:

x = evens[2]

With evens initialized as shown in the figure above, **x** will be set to the contents of the evens array retrieved from index **2**. The new value of **x** will be **4**.

Objects

You may have encountered the term *Object Oriented Programming*. Small Basic is not an object oriented programming language, but it does use objects. You cannot create objects using Small Basic. At this point, you can think of an object as a collection of actions that a Small Basic program can invoke in order to perform tasks outside of the native functionality of the Small Basic programming language. As you encounter other programming languages, you will learn that a program does not run in isolation. It communicates with other programs and the computer's accessories through a layer called an *operating system*. The operating system is a program that manages your program and stands between your program and other programs running on your computer. These other programs handle output to your screen or printer, input from your keyboard and mouse, and other tasks that Small Basic is unable to do.

One example of an object in Small Basic is TextWindow. If you wish to send text output to your screen or read text input from the keyboard, TextWindow handles these actions on

your program's behalf. For each action that an object can perform, there is a corresponding method. For example, among the methods offered by TextWindow are WriteLine and Read. To use a method offered by an object, you must call that method from within your Small Basic program. Methods use a syntax that is foreign to Small Basic but common in many other programming languages. Below is one line of Small Basic code that sends the text "Hello World!" to your screen.

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

Notice the string "Hello World!". This string is like any other string in Small Basic. Notice also the TextWindow object and its method WriteLine. To tell an object which method it should call, you name the object, follow its name with a dot (i.e., "."), and follow the dot with the name of the method. To send information into the method, you include that information by enclosing it with parentheses. The open parenthesis immediately follows the name of the method and the closing parenthesis immediately follows the data that you are sending to the method. In the case above, we are sending the string "Hello World!" to the WriteLine method of the TextWindow object. TextWindow will respond by creating an output window and then displaying "Hello World!" within it. We can just as easily pass the value of a variable into the WriteLine method.

```
annoying = "Doug Lenat"
TextWindow.WriteLine(annoying)
```

The code directly above will display "Doug Lenat" in the text window.

Boolean Values

According to the reference manual, in addition to numbers and strings, Small Basic supports boolean values. A boolean value is a truth value. A boolean value is always either True or False. A variable can be set to True or False. I have tested Small Basic using variables with boolean values, and I am not convinced that they are truly supported by Small Basic.

Whether or not Small Basic supports variables with boolean values, it does use boolean values in boolean expressions. We will cover boolean expressions in the next section.

Operators and Expressions

An expression in Small Basic is anything that can produce a value. All numbers, strings, and booleans are values. The string "Doug Lenat" produces the value "Doug Lenat" when it is used as an expression or within an expression. The number 21 produces the value 21 when it is used as an expression or within an expression. A variable that has a value produces that value when it is used, as or within an, expression. An array indexed by an index produces the value stored at that index when it is used as, or within, an expression. We can think of all literals, variables that have been assigned a value, and a reference to an element within an array as expressions. These are all examples of base expressions.

All expressions may be used on the right hand side of an assignment statement. The table below provides examples of assignment statements and the expressions used in each assignment.

Small Basic Assignment Statement	**The Expression that Provides a Value for the Assignment**	**Description**
X = 10	**10**	The expression 10 is evaluated to 10 and then X is set to 10.
Y = X	**X**	As X was assigned 10 in the previous example, the variable X is evaluated to 10 and then Y is set to 10.
angryPerson = "Doug Lenat"	**"Doug Lenat"**	The expression "Doug Lenat" is evaluated to "Doug Lenat"

		and then angryPerson is set to "Doug Lenat".
angryPeople[0] = <u>angryPerson</u>	**angryPerson**	As the variable angryPerson was set to "Doug Lenat" in the previous example, the 0th element of the angryPerson array is set to "Doug Lenat".

In the table above, the expression on the right hand side of the assignment statement is underlined.

If computers were only capable of moving data items from one position to another, we would be able to replace them with abaci, and simply push beads from one place to another. However, computers are also capable of performing operations on data items. Operations are performed by operators. You are already familiar with most of the operators within the Small Basic language, as most of them are identical to the operators you learned in grade school arithmetic. Below is a table of Small Basic's arithmetic operators.

Small Basic Arithmetic Operator	Grade School Equivalent	Example Expression	Description
+	**+**	**14 + 6**	The addition operator **+** adds the expressions on each side together. The value of this expression is 20.
-	**-**	**365 - 14**	The subtraction operator - subtracts the value of the expression to its right from the value of the expression to its left. The value of this expression is 351.

*	x	**20 * X**	The multiplication operator * multiplies the value of the expressions on each side together. The value of this expression is 20 times whatever value is stored in X.
/	÷	**Y / X**	The division operator / divides the dividend to its left by the divisor to its right. The value of this expression is the result of dividing the value of Y by the value of X.
^	**superscripted number**	**10^2**	The exponent operator ^ raises the value of the expression to its left to the power of the value of the expression to its right. The value of this expression is 100.

From the above, we can see that an expression can be either a base expression (a literal, variable, or an array element reference) or two expressions conjoined by a binary operator. In the previous sentence, an observant reader will see that an expression can be composed of other expressions, by means of operators.

Any expression can be bracketed by parentheses. For example, we could substitute **365 - 14** with **(365 - 14)**. Both **365 - 14** and **(365 - 14)** evaluate to the same value: 351. If parentheses do not change the value of the expression that they wrap, what is the point of having them?

Parentheses force an expression to be evaluated as a single unit. Consider the expression below:

20 - 10 * 5

Which operators should be evaluated first? If we evaluated the operators from left to right, **20 - 10** would evaluate to **10** and then **5** would be multiplied by **10**, resulting in **50**. On the other hand, if we evaluated **10 * 5** first, and then subtracted the result from **20**, the result would be **-30**.

Operators are evaluated by precedence. An operator with higher precedence is always evaluated before an operator with lower precedence.

The table below ranks operators by precedence. When operators of the same precedence follow in sequence, without parentheses, the rules on whether to evaluate the operators from left to right vs right to left varies according to the third column in the table.

Precedence	**Operators**	**Order for equal precedence**	**Example**	**Description**
Highest	( )	N/A	5 * (10 + 2)	The order of evaluation is: 5 * (10 + 2), 5 * 12, 60
High	^	right to left	10 + 2 ^ 2	The order of evaluation is: 10 + 2 ^ 2, 10 + 4, 14
Middle	* , /	left to right	4 + 100 / 10 * 5 - 1	The order of evaluation is: 4 + 10 * 5 - 1, 4 + 50 - 1 54 - 1 53
Low	+, -	left to right	7 + 9 - 10	The order of evaluation is: 7 + 9 - 10, 16 - 10, 6,

[Insert text and graphic showing the order of evaluation of a complex expression]

Arithmetic operators are intended to operate on numeric values. In most cases, if an arithmetic operator has one or two string operands (operands are the values of the expressions that the operator conjoins), Small Basic will attempt to convert the strings to numbers and then apply the operator to the results of those conversions. This works well when the string contains only digits, but it fails if the character sequence within a string does not conform to the format of a number. If Small Basic fails to convert the string into a number, while applying an arithmetic operator, it returns a nonsense numeric value.

The one exception is the **+** operator. In Small Basic, the **+** operator may be used to append strings together. When the expression to the left of the **+** operator is a string, it will convert the expression to the right to a string and then append the two strings. If the expression to the left of the **+** operator is a number, it will attempt to convert the expression to the right into a number and then perform arithmetic addition. The table below shows examples.

Assignment Statement	Conversion Attempts	Description
name = "John" + "Spencer"	none	Both are strings so the **+** operator is used to concatenate the strings. name is set to "JohnSpencer"
id = name + 11	11 is converted to the string "11"	The value to the left of the **+** operator is a string and the value to the right is a number. **+** converts 11 to the string "11" and then concatenates "11" to "JohnSpencer". id is set to "JohnSpencer11"
age = 20 + "7"	"7" is converted to the number 7	The value to the left is a number, so Small Basic attempts to convert "7" to a number. "7"

		conforms to the expectations of a number, so it is converted to the number 7. **+** adds 20 and 7. age is set to 27
age = "20" + 7	7 is converted to the string "7"	The value to the left is a string, so Small Basic converts the number to the right to a string and then concatenates them. age is set to "207"
age = 20 + "seven"	"seven" cannot be converted to a number and the attempt to convert it to a number returns 0 or some other random number	The value to the left is a number, so Small Basic tries to convert the number to the right to a number and fails. The conversion results in an unpredictable numeric result, which is added to 20. age is set to some unknown value added to 20

Boolean Values

Boolean values are truth values. A boolean value can only be True or False. Comparison operators, which will be discussed promptly, compare two values (usually a number or a string) and return either True or False.

Below is a table of Small Basic comparison operators.

Operat	Meanin	String	Description	Numeric Example	Description

or	g	Example			
=	equal	**"Food" = "food"**	"Food" and "food" are compared alphabetically. Since 'F' in "Food" is not equal to 'f" in "Food" (they are of different cases), the expression evaluates to False.	**101 = 102**	101 and 102 are numbers, so they are compared numerically. Since 101 is less than 102, the result of the comparison is False.
<>	not equal	**"Food" <> "food"**	Food" and "food" are compared alphabetically. Since 'F' in "Food" is not equal to 'f" in "Food" (they are of different cases), the expression evaluates to True.	**10 <> 11**	10 and 11 are numbers, so they are compared numerically. Since 10 and 11 are not equal, the result of the comparison is False.
<	less than	**"John" < "Steve"**	"John" and "Steve" are strings. "John" comes before "Steve" alphabetically, so "John" is less than "Steve". The result of the comparison is True.	**100 < 0**	100 and 0 are numbers, so they are compared numerically. Since 100 is not less than 0, the result of the comparison is False.
>	greater than	**"Steve" > "Mary"**	"Steve" and "Mary" are strings. "Steve" comes after "Mary" alphabetically, so "Steve" is greater	**100 > 0**	100 and 0 are numbers, so they are compared numerically. Since 100 is greater than 0,

			than "Mary". The result of the comparison is True.		the result of the comparison is True.
<=	less than or equal	**"John" <= "Steve"**	"John" and "Steve" are strings. "John" comes before "Steve" alphabetically, so "John" is less than "Steve". The result of the comparison is True.	**99 <= 100**	99 and 100 are numbers, so they are compared numerically. Since 99 is less than 100, the result of the comparison is True.
>=	greater than or equal	**"food" >= "food"**	"food" and "food" are strings. "food" is equal to "food" alphabetically, so "food" is greater than "food". The result of the comparison is True.	**100 >= 100**	100 is a number, so 100 and 100 are compared numerically. Since 100 is equal to 100, the result of the comparison is True.

Comparison expressions may be combined by using Small Basic's logical operators. Small Basic has only two logical operators: **And** and **Or**. As this book uses Small Basic only as a stepping stone to the next language, it is worth noting that Small Basic is the only programming language without a **Not** operator.

And and **Or** may only be used to conjoin two expressions with boolean values. In Small Basic, the only expressions with boolean values are comparison expressions and other boolean expressions composed of comparison expressions conjoined by logical operators.

A Small Basic boolean operator combines the boolean value of the expression to its left with the boolean value of the expression to its write. In the case of **And**, the resulting boolean value is **True**, if, and only if, both of its operands are **True**. For **Or**, the resulting

boolean value is **True**, if, and only if, either one or both of its operands are **True**. Otherwise, the result is **False**.

In English, if I say that it is raining and I am wet, I am being truthful only if it is true that it is raining and it is true that I am wet. Small Basic's **And** operator works in the same way that the English word "and" works.

In English, if I say "John is late or dead", I imply that John must either be late or dead, but not both late and dead. This kind of "or", frequently used in English, is not the same as the Small Basic's **Or**. The English "or" is usually exclusive in the sense that we assume that one, but not both, of the facts that it combines, are true. The Small Basic **Or** operator is inclusive, only one of its operands needs to be true for an **Or** expression to be **True** , but it is also **True** when both are **True**.

Below are truth tables for Small Basic's logical operators.

And				
Value of Expression to the Left	*Operator*	*Value of Expression to the Right*	*Result*	*Description*
False	**And**	**False**	**False**	Both operands are **False**, so the **And** expression evaluates to **False**.
False	**And**	**True**	**False**	The operand to the left is **False** and the operand to the right is **True**. Since it is not the case that both are **True**, the result is **False**.
True	**And**	**False**	**False**	The operand to the left is **True** and the operand to the right is **False**. Since it is not the case that both are **True**, the result is **False**.
True	**And**	**True**	**True**	Both operands are **True**, so the **And**

				expression evaluates to **True**.

Or				
Value of Expression to the Left	*Operator*	*Value of Expression to the Right*	*Result*	*Description*
False	**Or**	**False**	**False**	Both operands are **False**, so the **And** expression evaluates to **False**.
False	**Or**	**True**	**True**	The operand to the left is **False** and the operand to the right is **True**. Since at least one of the operands is **True**, the **Or** expression is **True**.
True	**Or**	**False**	**True**	The operand to the left is **True** and the operand to the right is **False**. Since at least one of the operands is **True**, the **Or** expression is **True**.
True	**Or**	**True**	**True**	Both operands are **True**, so the **Or** expression evaluates to **True**.

Comparison expressions may be joined by boolean operators to make complex boolean expressions. We will return to this topic in the next section.

Control Structures

Unless directed to do otherwise, a Small Basic program will execute one statement after another, from the top of the program to the bottom, unless it is either directed to do otherwise or it encounters an error. If all programs were run from top to bottom without repeating some sequence of statements, computers would not be able to accomplish most of their tasks. Calculators can get away with evaluating a single expression each time the enter or "=" key is pressed, toasters do just fine toasting a single load of slices with each

lever push, but computers are valued because they not only perform tasks quickly, but because they can process large quantities of data, over and over, without assistance when they are instructed to do so.

To do a task repeatedly, a program needs to have a mechanism to control which instructions should be repeated and how many times they should be repeated. Likewise, a computer program must decide which code to execute and which code to not execute. Deciding how many times to execute a section of code or whether to choose one section of code over another, requires language elements that control the processing sequence of a program. Small Basic has a small number of control structures which you may use to control which sections of code to execute, which to avoid, and which to repeat.

The If Structure

To control whether to execute or avoid a section of code, the Small Basic programmer uses an **If** structure. In its simplest form, the **If** structure is provided with a boolean expression and a section of code. If the boolean statement is **True**, then the section of code is executed. Otherwise, the section of code is skipped.

Consider the code below:

```
If person = "Mary" Then
    TextWindow.WriteLine("You're fired!")
EndIf
```

Somewhere in our human resources program, we may have a section of code similar to the above. When the **If** section is processed, Small Basic evaluates the boolean expression **person = "Mary"**. If this expression is true, the statement **TextWindow.WriteLine("You're fired!")** is executed. If the person is someone else, perhaps someone we do not want to fire, then the WriteLine statement is skipped.

An **If** structure must always end with **EndIf**. However, between **If** and **EndIf**, other optional parts of the **If** structure may be deployed. Let's suppose we not only want to fire Mary, but we also want to reward any employee that is not Mary.

```
If person = "Mary" Then
    TextWindow.WriteLine("You're fired!")
Else
    TextWindow.WriteLine("Without Mary, we can afford to give you a raise.")
EndIf
```

This next **If** structure is exactly like the previous **If** structure, except that it has a second section of code that is to be executed when **person = "Mary"** is **False**. Depending upon who walks in the door, that person is either fired or given a raise, contingent upon whether the person is Mary or not Mary.

Logically, we could survive with these two variations of the If statement and use them to cover all possible situations. For example, if we wished to fire Mary, give a warning to John, and reward everyone else, the next example would accomplish this task.

```
If person = "Mary" Then
    TextWindow.WriteLine("You're fired!")
Else
    If person = "John" Then
        TextWindow.WriteLine("You are warned!")
    Else
        TextWindow.WriteLine("Without Mary, we can afford to give you a raise.")
    EndIf
EndIf
```

Examining the code above, we can see that when the employee is not Mary, we can choose between warning that employee or rewarding the employee, depending upon what that employee is John.

Fortunately, Small Basic gives us the option of simplifying the above by using yet another feature of the **If** structure. The previous snippet of code can be rewritten thus:

```
If person = "Mary" Then
        TextWindow.WriteLine("You're fired!")
ElseIf person = "John" Then
        TextWindow.WriteLine("You are warned!")
Else
        TextWindow.WriteLine("Without Mary, we can afford to give you a
raise.")
EndIf
```

The **If** structure can have any number of **ElseIf** substructures. There can only be at most one **Else** structure and, if it is present, it must be the last substructure of the **If** structure.

The While Loop

Sometimes, you want Small Basic to repeat the same section of code so long as some condition is true. In Small Basic, the **While** loop is provided for exactly this purpose.

The while loop begins with the **While** keyword followed by a condition. The **While** loop ends with the **EndWhile** keyword. Between the **While** keyword and condition at the top, and the **EndWhile** keyword at the end, are statements you wish to repeatedly execute. A group of statements between a section of a control structure is called a **block** of statements. When this block is within a loop, it is called the **body of the loop**. Within the body of the loop, there should be at least one statement that alters the state of one or more variables mentioned in the controlling condition in such a way that eventually the condition becomes **False**. Otherwise, the Small Basic will execute the statements within the While structure until the universe ends, the power goes out, the computer is shut off, or Small Basic is interrupted by the user.

Imagine that we wish to fire all friends of Mary. We know that Mary has 10 friends and that the names of all of her friends are elements of the array **friendsOfMary**. The names of these friends of Mary are at indices 1 through 10 of the **friendsOfMary** array. The following use of the **While** structure could be used to accomplish this important task.

```
i = 1

While i <= 10
	friendOfMary = friendsOfMary[i]
	TextWindow.WriteLine(friendOfMary + ", you're fired!")
	i = i + 1
EndWhile
```

On the first pass through this **While** loop, **i** is **1**, when the condition **i <= 10** is evaluated. The statements inside of the block between **While I <= 10** and **EndWhile** are executed. The last statement of the block is **i = i + 1**. It changes the value of **i** to **2**, and the condition **i <= 10** is evaluated again. As it is still true, the statements inside of the block are executed again, and again, until finally, the value of **i** is set to **11**, the condition **i <= 10** becomes **False**, and the loop is ended. When the loop ends, execution continues at the first line of code of the **EndWhile** keyword.

In our previous example, **i <= 10** is what we call the control condition of the loop. We used a simple example here because this is your first encounter with Small Basic's **While** loop. Usually, the **While** loop is utilized for more complex terminal conditions because there is a better control structure for cases where a single variable's value, which changes in a predictable way, determines how many times a loop's body is executed. This alternative control structure is the **For** loop.

The For Loop

The For loop is a handy control structure designed with loops that rely upon a single variable, which changes incrementally, to decide when the loop should be terminated. The For loop is perfect for the task that we accomplished in the previous While loop example. The same example, written using the For loop, is listed below.

```
For i = 1 to 10
    friendOfMary = friendsOfMary[i]
    TextWindow.WriteLine(friendOfMary + ", you're fired!")
EndFor
```

In this example of the **For** loop, i is set repeatedly to increasing values, beginning with 1 and ending with 10. With each repetition, the body of the loop is executed. After the tenth execution of the body of the loop, the loop terminates.

Just as an **If** structure ends with **EndIf** and a **While** loop ends with **EndWhile**, a **For** loop ends with **EndFor**.

The **For** loop has variations that allow the programmer to control how **i** changes. The default behavior is to change the value of its control variable, in this case **i**, by adding one on each iteration of the loop. We can change this by adding a **Step** clause to the **For** loop. The **Step** clause is optional and when it is present, it comes at the end of the **For** line of the **For** loop. Below is another variation of our previous example, but in this case we only wish to fire every other one of Mary's friends.

```
For i = 1 to 10 Step 2
    friendOfMary = friendsOfMary[i]
    TextWindow.WriteLine(friendOfMary + ", you're fired!")
EndFor
```

With **Step 2**, on each repetition of the loop's body, **i** will be incremented by **2** (i.e. i = i + 1). This loop fires the friend's of Mary whose names are within the **friendsOfMary** array at indices 1, 3, 5, 7, and 9. This specialized version of our loop only fires Mary's odd friends.

We could get even by firing Mary's even friends in reverse order with the following variation of our For loop.

```
For i = 10 to 1 Step -2
    friendOfMary = friendsOfMary[i]
    TextWindow.WriteLine(friendOfMary + ", you're fired!")
EndFor
```

This loop fires the friend's of Mary whose names are within the **friendsOfMary** array at indices 10, 8, 6, 4, and 2.

The Subroutine

The BASIC programming language was created by John G. Kemeny and Thomas E. Kurtz at Dartmouth College in 1963. Like Small Basic, it was intended as an educational tool, and no one set out believing that it would become a programming language that professionals would use to write code. In some ways they were right. Most of the amazing uses to which BASIC was later applied were by professionals without a degree in computer science (Bill Gates is an example). Serious programmers tended to shun BASIC after their earliest courses because BASIC, though easy to learn, is not a very powerful language in terms of giving the programmer control over the computer and all it can do. In fact, this is why Small Basic gives you predefined objects that do a lot of the hard work for you, like drawing on the screen.

In keeping with the above, Small Basic's subroutines are a simplified, but inadequate, attempt to allow programmers to create sections of code that have names. These named sections of code can be called from anywhere in your program, allowing you to avoid writing the same lines of code, over and over, in many places within your program. Don't get me wrong, they are useful, but they are frustrating to more advanced programmers that know how much more could be accomplished if, instead of subroutines, real functions and procedures were offered instead (functions and procedures differ from subroutines in many ways, the most important of which is that you can pass values into them and they can return a value in response). Microsoft Visual Basic does have functions and procedures. Perhaps the hope at Microsoft is that after mastering Small Basic, you will want to move onto Visual Basic.

Defining a subroutine in your code is very simple. A subroutine is a block of code, not much different in appearance from the While loop. It's basic form is shown below:

```
Sub YourSubroutineName
    …
EndSub
```

When you define a subroutine, you need to give it a name. The name you chose should replace *YourSubroutineName*. The three dots are there to show you where a block of code should be inserted.

Let's suppose my program repeatedly asks the user the same question and then reads a response. Instead of repeating those two statements everywhere in my program, I could define a subroutine to perform that task and then call that subroutine whenever I wanted information from the user.

```
Sub NagTheUser
    TextWindow.WriteLine("Create another? (Y/N)")
    userInput = TextWindow.Read()
EndSub
```

Immediately above, I've defined a subroutine named "NagTheUser". Whenever I'd like to get input from the user, I could call the subroutine, as I do immediately below:

```
NagTheUser()
```

That is all there is to it. A subroutine, in Small Basic, is just a name for a block of code. When a user calls that block of code, Small Basic executes the instructions inside of the block and then returns to the place immediately after it was called. You can think of a call to a subroutine as a jump to the block of named code, the execution of the named code, and an immediate jump back to where we were.

Putting it All Together

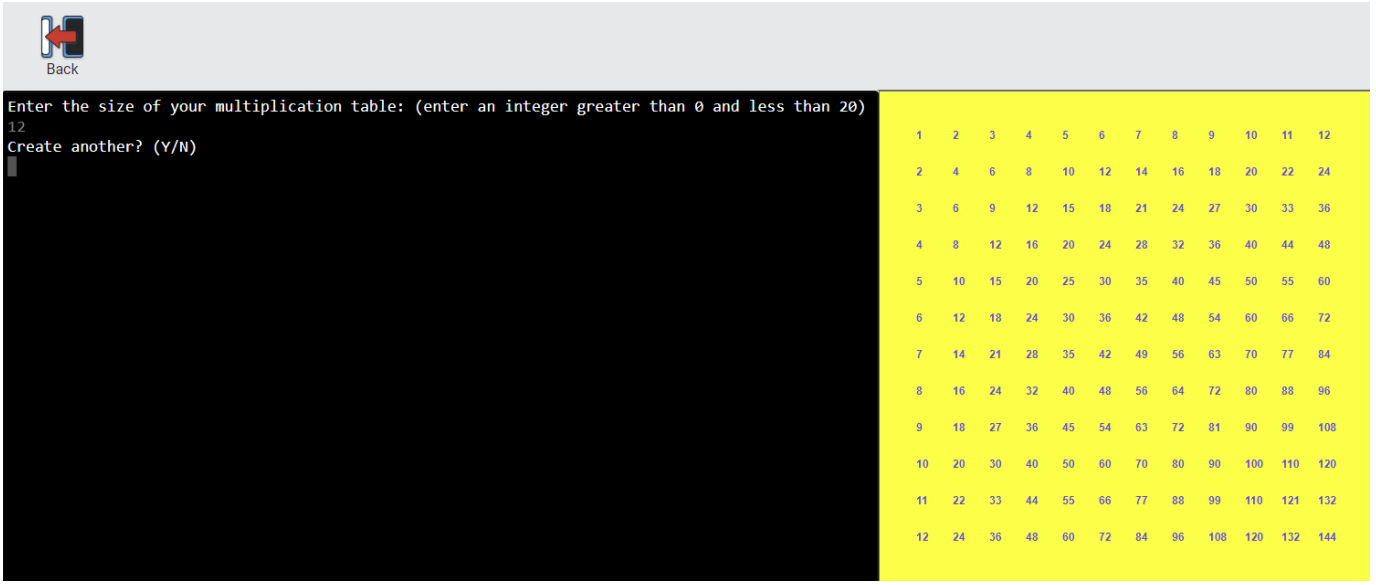
Go again to Small Basic's code editor! Select the code immediately below and copy it and then paste it into the code editor. If you are reading a hardcopy of this book, type the code you see below into the code editor.

Small Basic's code editor may be found online here: https://smallbasic-publicwebsite-code.azurewebsites.net/

```
GraphicsWindow.BackgroundColor = "Yellow"
GraphicsWindow.FontName = "Arial"
GraphicsWindow.FontSize = 12
GraphicsWindow.FontBold = "true"
GraphicsWindow.PenColor = "Blue"

tableCellWidth = 40
tableCellHeight = 40

continue = "True"

While continue = "True"
  TextWindow.WriteLine("Enter the size of your multiplication table: (enter an integer greater than 0 and less than 20) ")
  userInput = TextWindow.Read()
  n = userInput

  DrawMultiplicationTable()

  TextWindow.WriteLine("Create another? (Y/N)")
  userInput = TextWindow.Read()

  If userInput = "Y" Or userInput = "y" Then
      continue = "True"
  Else
      continue = "False"
  EndIf
EndWhile

Sub DrawMultiplicationTable
  GraphicsWindow.Clear()
  For y = 1 to n
      For x = 1 to n
          product = x * y
          xDrawingPosition = tableCellWidth * x
          yDrawingPosition = tableCellHeight * y
          GraphicsWindow.DrawText(xDrawingPosition, yDrawingPosition, product)
      EndFor
  EndFor
EndSub
```

After you've copied the text to Small Basic's code editor, hit the run button.

After pressing the "run" button, Small Basic will display both the TextWindow and the GraphicsWindow, side by side. If you provide an answer to its request for input, it will respond by drawing a multiplication table on the GraphicsWindow. By answering "Create another?", you can create as many multiplication tables as your wish. When you are finished, press the "back" button.



If you examine the code, the first five lines set up the GraphicsWindow by telling it what background color, font, font size, font style, and pen color to use. Each of these assignments is an assignment to a property of the GraphicsWindow. Until these properties are reset, your program will continue to use these settings.

Next, we created variables to govern how much space we will reserve, each time we write a product to the screen, for the text of the product. These variables could have any name, as they are not part of GraphicsWindow. I chose meaningful names as variable names, as we should always do.

The program uses a while loop to run the main part of the program repeatedly. As long as the user replies to the prompt with either "y" or "Y", it will ask the user for another table size and then draw the table.

We use the cell width and cell height defined near the beginning of the program to determine where to draw the products in the table. There are no real cells in our programs. The cells are imaginary, but we use them to place all of the output in a grid-like pattern. For any product x * y, we draw the product at position (x * tableCellWidth), (y * tableCellWidth). You can think of the GraphicsWindow as a coordinate system, like one used in math class. Each position in this coordinate system represents a screen pixel. Unlike in math class, the origin of this coordinate system is the top left corner of the GraphicsWindow and not the center. Also, unlike coordinate systems used in math classes, the value of y increases in a downward direction, not upward.

The GoTo Statement

The GoTo statement directs Small Basic to jump to another location in a program. To let Smart Basic know what other location to jump to, it needs a target. Think of the GoTo statement as a clown cannon. The target is called a label and it could be thought of as a net or cushion that the clown lands on after being shot through the cannon. For every GoTo statement, there should be a label, somewhere in the program, to receive the clown. Below is an example using the GoTo statement.

```
clowns[1] = "John Despencer"
clowns[2] = "Doug Lenat"
clowns[3] = "Josh Hagen"
maxClowns = 3

TextWindow.WriteLine("Enter the name of a person: ")
userInput = TextWindow.Read()

For i = 1 to maxClowns
        If clowns[i] = userInput Then
                GoTo dumpster
        EndIf
EndFor

GoTo endOfProgram

dumpster:

TextWindow.WriteLine("Another clown has been put in its place.")
```

```
endOfProgram:

TextWindow.WriteLine("Thank you for not littering!")
```

The GoTo statement is the most flexible of all control statements. It is a vestige of the history of programming, harkening back to the time when programmers wrote in assembler language. I have introduced you to the GoTo statement without prefacing my introduction with the traditional sneering required and expected of all books on programming. I found it easier to write my description of the GoTo statement as I imagined professors, teachers, and bosses sweating, red faced, and with throbbing veins in their neck (under those neckbeards of course), fuming that I had the audacity of not prefixing my introduction to this clown cannon with condemnation of the GoTo. I found the notion amusing.

Using a GoTo statement is considered blasphemy in the world of programming. Other control structures have evolved out of the need to eliminate the GoTo. In this way, the GoTo is like a grain of sand stuck inside of the shell of a living oyster. With an irritating grain of sand, natural pearls would not exist. Without the GoTo, For and While loops would never have come into existence.

The GoTo is universally abhorred because liberal use of the GoTo leads to messy, difficult to read, difficult to debug, and difficult to validate code. When GoTo was a thing, programmers felt free to write code that would jump around their program. Many problems were "solved" by "just jumping somewhere". The lines of execution became as messy as your junk drawer full of old USB cables and malfunctioning wired earbuds. Code that employed the GoTo took on the disparaging name of "spaghetti code", not a single person taking a moment out to think of how Italians might be offended by this term!

While the GoTo continues to exist, openly, nakedly, and wantonly within the specifications of many programming languages, teachers and professors try to keep its existence secret. As Microsoft Small Basic exists as an educational tool, many feel that it should not be included in the language. I disagree. Without the GoTo, what would teachers and

professors have to sneer at? Instead of spending an entire class on how evil it is, they'd need to find some other topic to fill that space.

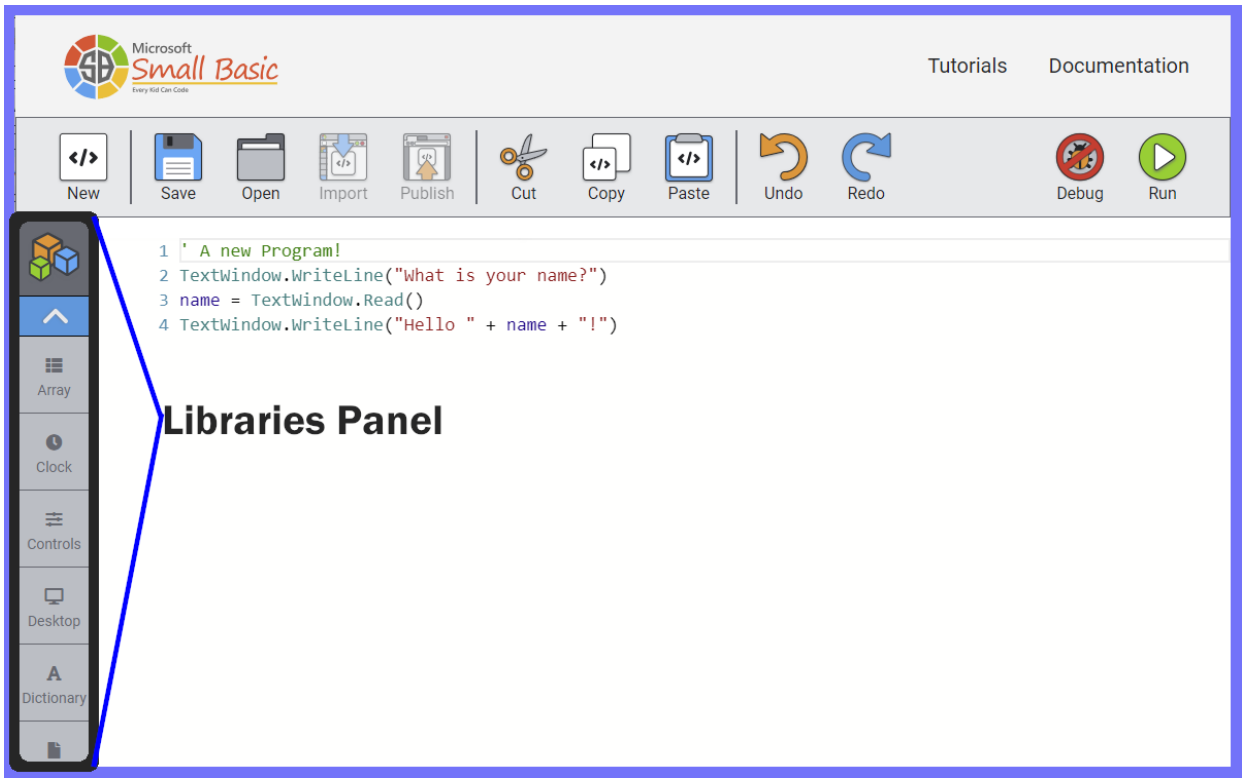
Despite the evils of the GoTo statement, it exists for a reason. Sometimes it is necessary to "Get the hell out of Dodge". Other programming languages have put lipstick on the GoTo pig by renaming it and its target a "catch throw", an "exception handler", or a "break statement", but Small Basic has none of these, so it is left with the lowly GoTo.

Yes, you should not use the GoTo statement in Small Basic, unless you find a brilliant reason to do so, that no one has thought of before, or you just want to launch some clowns into a dumpster.

More About Objects

We mentioned that objects cannot be created by writing code in Small Basic and that objects exist in Small Basic only to enable Small Basic programmers to access functionality that Small Basic itself cannot add, such as communication with the operating system or the creation of graphics. Small Basic's platform includes a suite of objects that were created using other .NET languages, more often than not in the C# language.

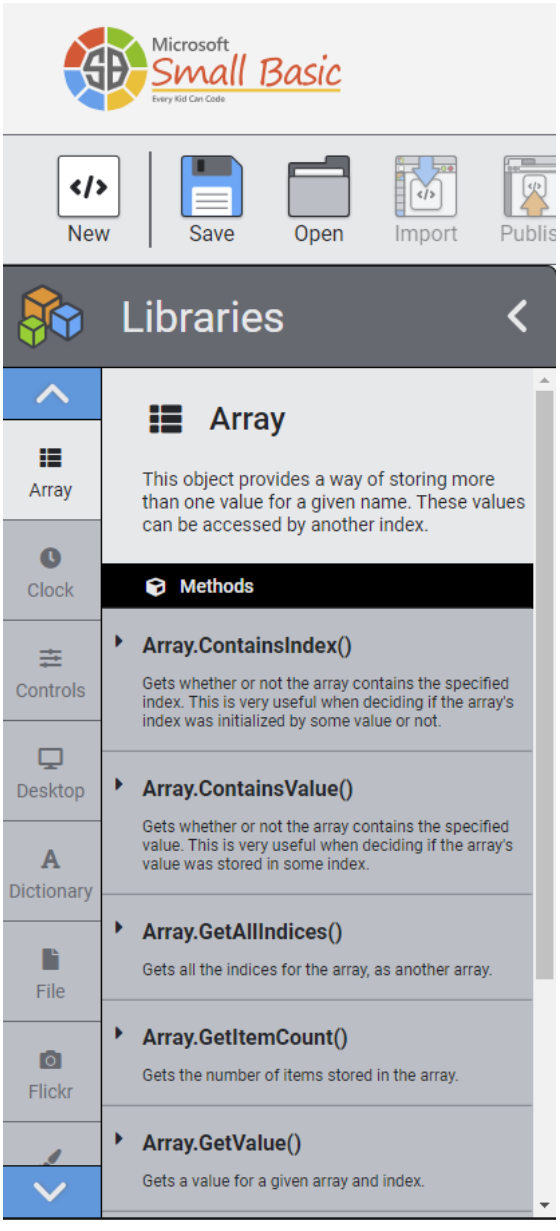
Objects included with Small Basic are presented to the user as libraries. The image below shows the reader where to find the Libraries Panel of Small Basic's web based interface.



The items listed in the Libraries Panel are objects available to Small Basic programs. Click on "Array" to view details about the Array object.

Communicating with an Object Using Small Basic

Methods



Every object has methods. Methods are the means by which a programmer accesses the functionality of an object. Small Basic programmers make calls to an object by typing the name of the object and then the name of one of the object's methods, the the object name and method name separated by a dot (e.g., ".").

For example, to specify the GetValue method of the Array object, the Small Basic programmer types **Array.GetValue**.

Every method requires a list of information in order to do its task. The information comes in the form of zero or more Small Basic values. These values are passed to the method through a comma separated list of values between an open and closed parentheses pair immediately following the method's name. For example, the GetValue method expects a single value indicating the index of the item in the array that the user wishes to access. If the Small Basic variable n has a value, it could be used to retrieve a value stored in Array at index n.

Array.GetValue(n)

	If a method does not need external information to do its task, the list passed to the method should be empty. An example of a method requiring no additional information to perform its task is the GetAllIndices method. **Array.GetAllIndices()** Explore the details panel of the Array object to learn more about its methods.

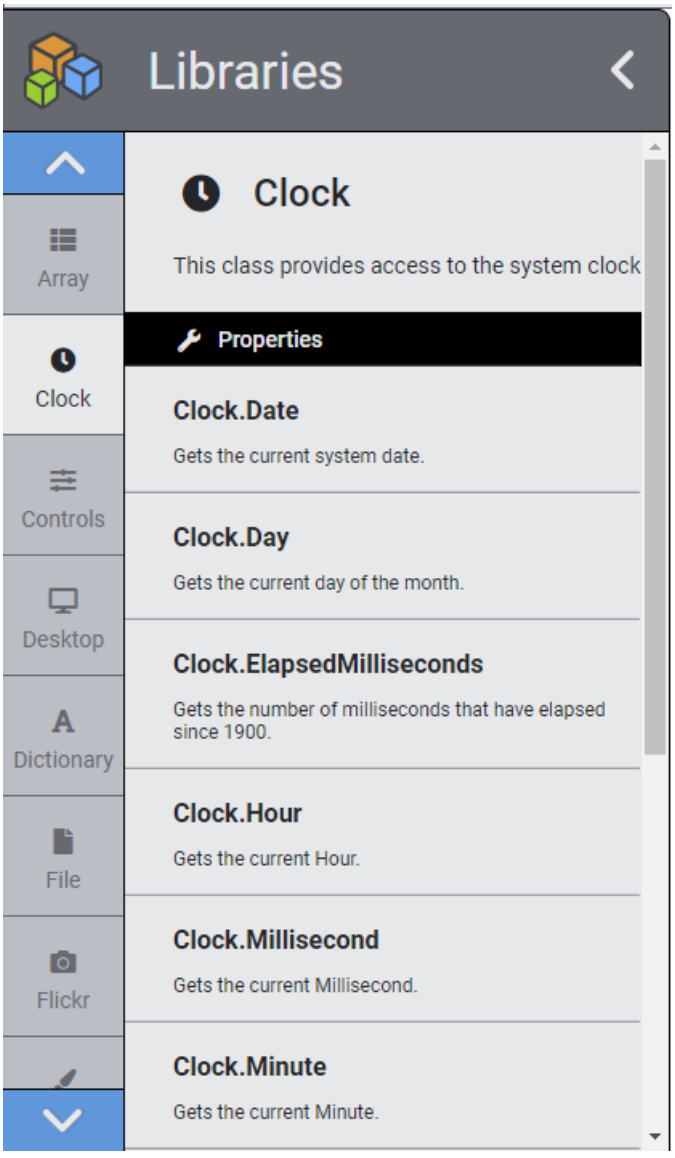
When a method is called by a Smart Basic program, the method may return a value. The value will be a Small Basic value. For example, if 1 is an index of Array in which the value 10 is stored, a call to Array.GetValue(1) would return the value in Array stored at that index. This returned value could be assigned to the variable v as shown below.

v = Array.GetValue(1)

This statement sets **v** to the value stored in array at index **1**. The value may then be used by the Small Basic program that made this assignment.

Properties

In addition to methods, many objects have properties. A property looks like a method, but it does not accept additional information. It is an error to access a property by following its name with an open and closed parentheses pair. A property is similar to a variable. You use its name to retrieve a value, only in the case of properties, you must designate the property using the same pattern that is used to designate the name of a method.



The Small Basic Clock object provides an interface into your computer's system clock. A Small Basic program can retrieve information about the current date and time. The clock object provides only properties as an interface between a Smart Basic program and the system Clock.

If a programmer wished to display the current date, the following line of code would access the Clock object and retrieve the current data by reading its Clock.Date property.

TextWindow.WriteLine(Clock.Date)

Explore the details panel of the Clock object to learn more about its properties.