Wolfscript: A Beginner's Guide

Erick Bauman

June 2, 2013

Contents

0.1 Introduction

What is Wolfscript? Wolfscript is a simple and straightforward programming language designed to be easy to learn and to be programmed directly on Android devices. Wolfscript has been designed for ease-of-use and simplicity, even for those unfamiliar with programming.

Why learn Wolfscript? Because it's easy! If you are just learning to program, Wolfscript offers a simple, easy-to-learn way to program on the go, and if you already have experience programming, you should be able to learn most of Wolfscript in a day. What's great about Wolfscript is that you only need an Android device to write programs, and there is no need for an internet connection to run programs. So next time you find yourself with nothing to do and your phone in your pocket, practice making games instead of playing games!

Chapter 1

Learning the Basics

1.1 Using the Interactive Terminal

The interactive terminal is basically just a command prompt or console that you can type expressions into, and the terminal will respond back with the results. In order to run the interactive console on your Android device, press the “Interactive Terminal” button at the main menu. When in the interactive terminal, enter text into the text field and submit it using the “submit” button or by pressing enter.

1.2 Calculator

Wolfscript has an interactive terminal mode that should be very familiar to anyone who has used a scientific calculator. The interactive terminal is a perfect place for someone with no programming experience to start. It behaves very much like a calculator and can even be used as one.

1.2.1 Expressions

Let’s begin with something very simple. In the interactive terminal, type

```
2+2
```

and hit enter. The terminal should return to you the obvious answer 4. Try a few more expressions! Wolfscript supports complex expressions with nested parentheses, such as

```
5*(2+3/5)
```

with full support for order of operations. The numerical results of these expressions are exactly what you would expect: the exact value that you would get if you used a calculator. Wolfscript supports many operators, some of which you may or may not have used before. Not familiar with some of these operators? We will come back to some of them later, especially the logical operators and comparisons. Here is a table of the operators and what they are, arranged by decreasing precedence:

Operator(s)	Description
()	grouping subexpressions together
!, -	factorial and unary negative (NOT subtract)
&, |	logical AND and OR
<, >, ==, <=, >=, <>	less than, greater than, equal, less than or equal to, greater than or equal to, not equal
*, /, %	multiplication, division, and modulo (remainder)
+, -	addition and subtraction

1.2.2 Assignments

Simple expressions are easy. However, it is important to note that these expressions merely are equal to a certain value; they do not *do* anything (with the exception of setting the "ans" variable discussed in the next paragraph). In order to store the result of an expression for future use, you must assign the value to a variable. This can be compared to the M+ button on a basic calculator, except variable assignment is far more powerful and versatile.

Wolfscript adopts a few conventions from graphing and scientific calculators, and one convention it borrows is an "ans" variable. This variable always contains the value of the last calculation. For example, if you entered the lines

```
2+2
ans*5
```

you would get 20, because after the 2+2 line, ans was 4. After the ans*5 line, ans is 20.

You can assign your own variables instead of relying on ans. Although you can assign variables on graphing calculators, many of them do not allow you to name them. However, here you can name the variable *almost* anything you want, provided it is not reserved for use somewhere else.

Rules for Variable Names
Variables cannot contain spaces
Variables cannot contain operators such as + or *
Variables cannot be identical to words that Wolfscript uses for special purposes
Variables CAN have uppercase or lowercase letters
Variables CAN contain characters that are not reserved such as _ (underscore)

Try this:

```
a = 2+2
b = a+2
a+b
```

Now, a is 4, b is 6, and ans is 10. Besides assigning a value to ans, the line a+b did not do anything, unlike the other two, which explicitly assigned the value "2+2" to a and then "a+2" to b. It is important to emphasize that the equal sign in this case is not declaring an equation but rather putting the value on the right into the variable on the left. When writing an assignment, you should always have a variable on the left and an expression on the right.

What is a variable? It is essentially a container for a value that has a convenient name for referencing it. Therefore, you should name variables something intuitive and relevant to the problem at hand. For example, if you are keeping track of the number of cats for something, you might name a variable "numCats" or "cats":

```
cats = 77
```

Now I can always check how many cats there are by looking at cats.

1.2.3 Arrays (Lists of Numbers)

If you hear the word array, you may feel somewhat intimidated. What is an array? Well, in this case it is just a list of numbers that can be accessed by referring to where they are in the list. It can be very convenient to have a list of numbers that you can refer to in this manner. Let's take a look at an array:

```
[2,4,8,1]
```

If you type this into the interactive terminal, you will get the same thing back as a result. All you have done is said "I have this list of numbers. They are 2, 4, 8, and 1." Wolfscript has basically said, "Ok." So let's actually do something with this array. Let's use what we learned about variables and assignments to put this array into a variable so we can use it:

```
myArray = [2,4,8,1]
```

The variable name “myArray” has no particular significance. You can choose any name you wish. Now, we have an array, and we have saved it as myArray. How can we access the numbers in it?

First, there is a very important property of arrays to understand. The values in arrays are not numbered from 1, but instead from 0. Why? It can make math more straightforward in certain circumstances, and it has been a programming convention for years. Regardless of the reasons, this means that the number 2 in myArray is in “slot” number 0. Instead of using the terms slot or location, we will use the term index. Here is a representation of the array:

Index	0	1	2	3
Value	2	4	8	1

Therefore, in myArray, the number at index 3 is 1, and the number at index 1 is 4. Let’s access one of these numbers!

```
myArray[2]
```

This accesses the value at index 2, which is 8. Now let’s change the value to something else!

```
myArray[2] = 9001
```

Remember, for assignment, a variable must be on the left. In this case, the variable is myArray, and we are modifying its second index. Now, myArray looks like this:

Index	0	1	2	3
Value	2	4	9001	1

Let’s say we need another slot in our array. If we need another number, we need a bigger array. No problem! Simply assign the new index a value:

```
myArray[4] = 9
```

The array is expanded and now includes that value:

Index	0	1	2	3	4
Value	2	4	9001	1	9

What happens if we expand it by more than 1?

```
myArray[8] = 2
```

The array expands, and the values at indices 5, 6, and 7, which we never defined, become zero by default. This is something to remember for convenience.

Index	0	1	2	3	4	5	6	7	8
Value	2	4	9001	1	9	0	0	0	2

Notice that the array can contain duplicates.

Arrays may not seem useful now, but if you do any significant amount of programming, you will find out how useful they can be.

Another fact to take note of is that an array can contain more than just numbers. It can also contain more arrays! We will come back to this later.

1.2.4 Equality and Booleans

Sometimes you want to know if an expression is equal to another expression, or maybe you want to know if it is greater or less than the other. In this case, conditionals are very useful. For example, if you wanted to know whether 2+2 was equal to 8/2, you could enter

```
2+2 == 8/2
```

This is asking Wolfscript whether 2+2 is equal to 8/2. However, what is the value you get back? If you enter this into the interactive terminal, you get

```
1
```

which may seem puzzling. However, the computer needs to answer in a consistent way. If it were a person, it might respond "Yes, 2+2 is equal to 8/2." However, that is a bit inconvenient for a computer, so we have to use something a bit simpler, if possibly more confusing at first sight. This is made more clear if you enter

```
2+2 == 42
```

and get

```
0
```

Since 4 is not equal to 42, 0 must mean “no,” or more accurately “false.” Likewise, 1 must meant “yes,” or more accurately “true.” As a matter of fact, many programming languages do use true and false in situations such as this, and they are known as booleans. However, Wolfscript tries to keep everything as a number if possible, and thus 0 is treated as false, and every other number is treated as true.

You can check for more than just equality. Here is a list of ways to compare values:

Symbol	Comparison
==	equal
>	greater than
<	less than
<=	less than or equal
>=	greater than or equal
<>	not equal

You may wonder under what circumstances you would want to check if an expression was true. The best example for this is if statements, which are indispensable for programming. We will discuss these shortly.

1.3 Simple Programs

1.3.1 Writing Your First Program

Up until now, we have been looking at the interactive terminal, but in order to write real programs, you want to be able to save your program in a file so that you can run a sequence of commands. While the interactive terminal is like a calculator, writing a program is like automating all the commands you were giving to the calculator.

This will assume you are using the Android-based WolfscriptIDE, but if you are not, you can type the programs in a text editor.

For your first program, you are going to create the classic “Hello World” program. All it does is prints out the words “Hello world,” and it is often used as the first program one writes when using a new programming language.

First, create a new program, and then press the “+ Add Line” button. Choose the “Expression” option and type

```
outs "Hello world!"
```

Press “OK” and then press “Run.”

Congratulations! You just wrote your first program!

1.3.2 Input and Output

In the last section, we displayed the words "Hello world!," but how did we do that? Let's look closer at how to display words and numbers, as well as allowing the user (the person running your program) to input values.

You can display words by using the command outs, used like this:

```
outs "These words will display."
```

or, if you want to display words from a variable,

```
words = "Hello!  These words will also display."
outs words
```

To just display a number, use out:

```
abc = 4
out abc
```

This will display the value of abc. Remember that if you don't use quotes when displaying words, Wolfscript will think you are wanting to display a variable, and if that variable does not exist, your program will not work.

In order to input values, use in() like this:

```
userInput = in()
```

Now, the user's input is stored in variable userInput. Simple! Note that it is a good idea to prompt a user for input before using in() so that the user will know that they are supposed to enter something and what it is supposed to be.

1.3.3 Comments

Sometimes you will want to tell either yourself or someone else what you are doing in your program. This is what comments are for, as they allow you to say things that the computer would not understand. They are essentially asides directed at yourself and others that the computer completely ignores. To make a comment in Wolfscript, start a line with a #, like this:

```
#This variable will track the number of cats in the house
cats = 70000
```

As you can tell, the comment explained the use of the variable for another person. By convention, it is discouraged to write comments for things that are self-evident, such as

```
#Set cats to 70000
cats = 70000
```

because it is quite obvious that the line does that. Keep in mind that comments can be a powerful tool if used correctly, but if they become redundant, they will be useful to nobody.

1.3.4 If Statements

Here is what an if statement looks like:

```
if num == 5
    #Contents of the if statement go here
end
```

Before we begin to use if statements, recall the section "Equality and Booleans." If the variable num is indeed equal to 5, then num==5 will evaluate to 1. If it is not, it will evaluate to 0. The if statement will run the lines between its start and end if the expression after if evaluates to a number that is not zero, and if the expression does evaluate to zero, we jump to the end of the if statement.

```
num = 4
if num == 5
    #The code here is skipped because 4 is not equal to 5
    #4==5 evaluates to 0
end

num = 5
if num == 5
    #The code here is run because 5 is equal to 5
    #4==5 evaluates to 1
end
```

As you can tell, this allows you to choose whether certain code runs or not, which is very useful.

Something to note is that, when using the Android IDE, you will not have to worry about the closing "end" line, as they are automatically added.

All you have to do is add the if statement's condition, and it does the rest. There is a specific section that goes into using the IDE in more detail.

There is more to if statements than simply an ordinary if. Wolfscript also contains elseif and else. Elseifs and elses only run if the preceding if (or elseif) did not:

```
num = 4
if num == 5
    #The code here is skipped because 4 is not equal to 5
    #4==5 evaluates to 0
elseif num == 4
    #The code here is run because 4 is equal to 4
    #4==4 evaluates to 1
elseif num > 3
    #The code here would have run if the previous elseif had not
else
    #This would have run if the if had not run
    #and none of the above elseifs had run
end
```

Note that the elseifs replace the "end" normally at the end of an if statement. You can have as many elseifs as you want, but only one if and one else can be attached to them. To have an elseif, you must start with an if, and an else requires at least an if. An else is always optional:

```
#The following is valid
num = 4
if num == 5
    #The code here is skipped because 4 is not equal to 5
    #4==5 evaluates to 0
else
#This code is run because the if did not run
end

#The following is valid
num = 4
if num == 5
    #The code here is skipped because 4 is not equal to 5
    #4==5 evaluates to 0
elseif num > 5
    #This code is also skipped because num > 5 evaluates to 0
```

```
end

#The following is INVALID
num = 4
elseif num == 5
    #You cannot start with an elseif
end

#The following is INVALID
num = 4
else
    #You cannot start with an else
end

#The following is INVALID
num = 4
if num == 5
    #The code here is skipped because 4 is not equal to 5
    #4==5 evaluates to 0
else
    #You cannot have two elses attached like this.
else
    #An else is always terminated by an "end," because it is what
    #runs when all the other conditions do not.
end
```

Here is an example of an if statement in action:

```
num = 1
if num < 5
    num = num+1
end
```

When the code runs, num is incremented by 1 because it is less than 5.

Exercise 1

Write a program that asks for the user's age, and then prints out a different message for several different age ranges. A sample solution is at the end of this document.

1.3.5 Loops

Sometimes you want to perform a similar task multiple times. Fortunately, loops allow you to perform a task an arbitrary number of times without typing it over and over. They behave very similarly to if statements; they only will run if the provided expression evaluates to a nonzero value. Let's start with while loops:

```
num = 5
while num > 0
    num = num-1
end
```

This while loop runs 5 times, subtracting 1 from num every time.

run	num	num > 0?	Does the loop run?
1	5	yes	yes
2	4	yes	yes
3	3	yes	yes
4	2	yes	yes
5	1	yes	yes
6	0	no	no

Since a loop will run until the condition is zero, a loop such as

```
while 1
    #This loop will never terminate
end
```

will run forever, or until the program is forcefully terminated. Try to avoid infinite loops. It may not always be so obvious that a loop's condition does not allow it to terminate.

The other type of loop in Wolfscript is the for loop. For loops are similar to while loops, and a while loop can always be used where a for loop is used. They are simply more convenient than while loops in certain situations.

For loops have three parts, separated by semicolons:

```
for i = 5; i > 0; i = i-1
    #Contents of for loop
end
```

This for loop actually does the same thing as the first while loop shown: it counts down from 5 to 0 and stops. Let's take a closer look at each part.

```
i = 5
```

The first part is used for initialization; it is run once at the beginning of the first loop. Therefore, we now have a variable named "i" with a value of 5. By convention, the letter "i" is often used in for loops, but this initialization can take the form of any expression or assignment.

```
i > 0
```

The second part is the same as the condition for the while loop; it is used to determine if the loop will run. It runs immediately after the initialization code is run when the for loop is first started, and it then runs immediately before each run to determine if the loop will run again.

```
i = i+1
```

The third part is the code that runs after each loop. You can think of it as a line of code that is automatically appended after the last line of the loop.

As another example of loops, here is a for loop that prints out the numbers 25 through 50:

```
for i = 25; i <= 50; i++
    out i
end
```

A Quick Shortcut: ++ and - -

In the for loop example, variable i was incremented using the code

```
i = i+1
```

but there's a shorter way to write this. You can instead type

```
i++
```

for the same result. Think of it as shorthand for the same thing. In addition,

```
i--
```

subtracts one from a variable. This is a common feature in many languages.

Exercise 2

Write a program that accepts a number from the user and then counts up from 1 to that number. Have the program print out the number each iteration of the loop so it visually counts up. For example, if the user entered 4, the output would be:

```
Enter a positive number:
4
1
2
3
4
```

The first 4 is the user's input, and the rest of the numbers are from the loop.

1.3.6 Methods

Sometimes you will find yourself wanting to reuse code you have written. Fortunately, you don't have to just copy the code over and over. Instead, you can store it in a method, where it can be run over and over from other places in your code! A method is a block of code that has been packaged together under a name so that it can be called at any time. Different languages may call this code structure different things, such as subroutines or functions, but the important thing to remember is the concept behind it.

Methods are defined like this:

```
#This method merely prints out "Hello World!" when it's called
def hello()
    outs "Hello World!"
end
```

This method, which prints out "Hello World!" when run, does not do anything when defined besides store itself for later use. All you are doing when defining a method is telling Wolfscript that you have a block of code that you will want to use later.

To call this method, simply write

```
#"Hello World!" will be printed
hello()
```

All methods return values so that they can be part of mathematical expressions. You can use the "ret" keyword to specify what value to return. To give an example,

```
#This rather useless method always returns 42.
def giveMe42()
    ret 42
end

#When this is called, 42 is printed out.
out giveMe42()
```

What does it mean when a method "returns" a value? It simply means that the method call (e.g. giveMe42()) is replaced with the value it returns after it is run. Therefore, if you typed

```
giveMe42()+5
```

into the interactive terminal, it would result in the expression 42+5, which would give you 47.

You may be asking why there are parentheses at the end of the method. That is because methods allow for more functionality than simply being called. You can also give variables to them so that you can perform different operations on them. For example, let's say that you wanted to write a method to add two numbers:

```
#This is not recommended since + is a faster and shorter way,
#but this is important for explaining the concept.
def add(a,b)
    ret a+b
end

#Prints 4
out add(2,2)
#assigns 7 to num
num = add(2,5)
#Prints 8, which is num+1
out add(num,1)
```

Methods can contain more than one "ret" statement, but they will stop running as soon as one is encountered. Combining methods with if statements allows for the possibility that one of several "ret"s will be run. For example, here's a method that returns 1 if the variable passed to it is greater than 9000 and 0 otherwise:

```
#Returns 1 if a > 9000.  Returns 0 otherwise.
def large(a)
    if a > 9000
     ret 1
    else
     ret 0
end

#Prints 1
out large(9001)
#Prints 0
out large(2)
```

Methods will also return the value of the last line of the method if no "ret" is encountered first.

Wolfscript Functions

Wolfscript already has some built-in methods that are needed for basic functionality. However, to keep these distinct from user-made methods, they are called functions. There is no reason for this other than for clarity. You have already used the in() function, which allows for console input. However, there are several others. There is a list of them at the end of this document, and several will be mentioned in future sections as they are needed.

Exercise 3

Write a method that determines if the one parameter passed to it is even or odd. If it is even, return 0. If it is odd, return 1. Add a few lines that run the method to test it. (Hint: using modulus, which is represented by %, can help determine whether a number is even or odd. If you need help, read about remainders.)

1.3.7 Arrays Revisited

Now that you have learned about loops and methods, it is time to discuss the power of arrays. As mentioned at the end of the first arrays section, an array can contain arrays as values. This allows for "multidimensional" arrays, which are very useful. Here is an example of a multidimensional array being defined:

```
arr = [[2,4],[5,6]]
```

This array contains two smaller arrays, both of length 2. However, arrays can contain a combination of both numbers and arrays of varying lengths:

```
arr = [709,[2,4,6,8],[19,29],[20]]
```

Sometimes you will want to know the length of an array. For this, Wolfscript has a built in function: len(). Just pass in the array's name as a parameter, and it will return the length.

Here is a for loop that prints out the contents of an array:

```
myArray = [1,1,2,3,5]
for i = 0; i < len(myArray); i++
    out myArray[i]
end
```

Exercise 4

Write a method that, when given two arrays of numbers, will return an array containing the sums of the numbers in the corresponding indices up to the length of the shorter array. For example,

```
arr1 = [2,4,6]
arr2 = [7,8]
answer = addArr(arr1,arr2)
out answer
```

would print [9, 12], because 2+7 is 9 and 4+8 is 12. The 6 is never added to anything because there is no corresponding number in arr2.

Chapter 2

Appendices

2.1 Sample Answers to Exercises

Exercise 1

```
outs "What is your age?"
age = in()
if age < 18 & age >= 0
    outs "You are too young to vote in the US."
elseif age < 65 & age >= 0
    outs "You are of a working age and can vote."
elseif age < 100 & age >= 0
    outs "You are either retired or are approaching retirement age."
elseif age < 150 & age >= 0
    outs "You are incredibly old."
elseif age < 5000 & age >= 0
    outs "You are older than is physically possible.  I suppose you could be
         a tree."
else
    outs "Liar."
end
```

Exercise 2

```
outs "Enter a positive number:"
num = in()
for i = 1; i <= num; i++
    out i
end
```

Exercise 3

```
def evenOrOdd(num)
    if num%2 == 0
        ret 0
    else
        ret 1
    end
end

#Output should be 1, 0, 1, 0, 1, 0
out evenOrOdd(5)
out evenOrOdd(2)
out evenOrOdd(1)
out evenOrOdd(42)
out evenOrOdd(709)
out evenOrOdd(0)
```

Exercise 4

```
def addArr(a1,a2)
    shorter = len(a1)
    if( len(a2)<len(a1) )
        shorter = len(a2)
    end

    #Create an empty array that we will fill with sums
    sum = []

    for i = 0; i < shorter; i++
        sum[i] = a1[i]+a2[i]
    end

    ret sum
end

arr1 = [2,4,6]
arr2 = [7,8]
answer = addArr(arr1,arr2)
out answer
```

2.2 Wolfscript Keywords

In Wolfscript, keywords do not need parentheses after their use if they accept any parameters.

Keyword	Parameters	Use
clr	variable name	Remove a specific variable from the interpreter.
clrloc	nothing	Remove all variables from the current interpreter (not its parent).
loc	nothing	Display a list of all variables in the current interpreter (works in interactive console only).
met	nothing	Display a list of all methods in the current interpreter (works in interactive console only).
frac	nothing	Display the fractional value of ans (works in interactive console only).
out	expression	Print out the value of the expression to the console.
outf	expression	Print out the fractional value of the expression to the console.
outs	expression	Interpret and print to the console the string representation of the expression.
end	nothing	End a code block started by if, while, for, or def.
ret	expression	Return the value of the expression as the return value of the method this ret is in.
status	nothing	Print some debug data (works in interactive console only).
if	expression	Execute the code after this line up to a closing end or elseif or else if the expression evaluates to anything that is not 0.
elseif	expression	Execute the code after this line up to a closing end or elseif or else if the expression evaluates to anything that is not 0 unless an if or elseif has already executed.
else	nothing	Execute the code after this line up to a closing end unless an if or elseif has already executed.
while	expression	Execute the code after this line up to a closing end and continue looping as long as the expression evaluates to anything that is not 0.
for	expression1; expression2; expression3	Execute expression1 immediately. Then execute the code after this line up to a closing end and continue looping as long as expression2 evaluates to anything that is not 0, executing expression3 at the end of every loop.
def	methodName(param1, param2, ... , paramN)	Store the code after this line up to a closing end under the name methodName. Save all parameter names as the variable names for the values that will be passed when the method is called.

2.3 Wolfscript Functions

Wolfscript functions look similar to user methods when used.

Function	Input	Output	Use
sin()	number	number	Find the sine of a degree in radians.
cos()	number	number	Find the cosine of a degree in radians.
tan()	number	number	Find the sine of a degree in radians.
len()	number or array	number	Find the length of an array. If a number is passed, the result is -1.
in()	nothing	number or array	Accept input from the user.
str()	number or array	array	Return the string representation of a number (e.g. 2→[50]→"2" or 3/2→[49,46,53]→"1.5"). If an array is passed, the same array is returned without modification.
num()	number or array	number	Convert a string representation of a number into that number. If the array values or number do not represent the ASCII values of numbers (48-57), then behavior is undefined.
time()	nothing	number	Find the current system time in milliseconds since January 1, 1970. Equivalent to Java's System.currentTimeMillis().
str()	number or array	array	Return the string representation of a number, except the representation is in fractional form instead of a truncated decimal (e.g. 2→[50]→"2" or 3/2→[51,47,50]→"3/2"). If an array is passed, the same array is returned without modification.
eq()	number or array, number or array	number	Determine if the two parameters are equal. Returns 1 if they are. Returns 0 if they are not. Arrays cannot be equal to numbers and a combination will always return 0.

2.4 Wolfscript Exceptions

These are the potential exceptions you may encounter if you make a mistake in your code.

Exception	Possible cause
Unparsable expression	The expression entered may contain a variable that does not exist, or may have operators in invalid places.
Index out of bounds	A request for a value in an array has an invalid index, or you may be trying to assign a value to a negative index.
Variable is not an array	You may be treating a variable that is not an array as an array (such as trying to obtain a value from an index when the variable is a number).
Division by zero	Dividing by zero is not allowed.
Invalid array declaration	The definition for an array may have mismatched brackets. This may also show up for Strings since they are stored as arrays.
an unknown error has occurred	Wolfscript has no idea what went wrong. Report these errors so they can be fixed!