Wolfscript: An Educational Programming Language for Android

Erick Bauman

A Thesis Submitted in Partial Fulfillment of the Requirements for Graduation with Honors in Computer Science

Department of Mathematics and Computer Science
Southwestern University
Georgetown, TX 78626

May 6, 2013

Approved _____
Dr. Richard Denman
Honors Advisor
Mathematics and Computer Science

Approved _____
Dr. Barbara Anthony
Committee Member
Mathematics and Computer Science

Approved _____
Mr. Andy Ross
Committee Member
Economics and Business

Wolfscript: An Educational Programming Language for Android

Erick Bauman

May 6, 2013

Department of Mathematics and Computer Science
Southwestern University
Georgetown, TX 78626

Thesis Committee:
Dr. Richard Denman
Honors Advisor
Mathematics and Computer Science

Dr. Barbara Anthony
Committee Member
Mathematics and Computer Science

Mr. Andy Ross
Committee Member
Economics and Business

Copyright ©2013 Erick Bauman

Abstract

Wolfscript is an educational programming language designed to be easily learned by anyone proficient with a scientific calculator. The key focus of the language is simplicity joined with the flexibility of other high-level languages. There are only two data types in Wolfscript so far: numbers and lists of numbers (arrays). Numbers are arbitrary-precision; all numbers are stored as a numerator over a denominator, so all rational numbers can theoretically be represented with no overflow. Strings are represented as arrays of characters, which are stored as numbers. Wolfscript contains all the features one expects from a procedural programming language: if/elseif/else, for/while, and methods. The syntax and IDE are designed for ease of use on any Android phone or tablet. This paper will cover the design philosophy of the language and the IDE, as well as some of the technical details.

Contents

Part I

Wolfscript

Chapter 1

Introduction

1.1 Why Wolfscript?

Prior to starting work on Wolfscript, I had seen several examples of programming languages designed exclusively for educational purposes. While each language had its own advantages, I felt there was a void that could be filled with a new language.

While I intended Wolfscript to be educational, I wanted it to be similar in at least its syntax to actual languages in use. Two similar visual programming languages are Alice [7] and Scratch [3], which are intended to teach programming to younger students and are substantially different from most languages taught in upper-level computer science courses. While ideal for teaching young children and people not intending to advance in computer science, the visual environments of Alice and Scratch, along with the inflexible code blocks, make them insufficient for a smooth transition into more "serious" languages.

Another programming environment, Greenfoot [6], allows students to easily write 2D games and applications using Java. I was introduced to Greenfoot in high school, and I considered it to be very good at teaching concepts and allowing students to put together satisfying projects. However, it still uses a full-featured programming language, requiring students to learn its conventions before they are comfortable using it. Therefore it is better to use Greenfoot after spending some time learning basic programming concepts.

I hoped to fill the gap between these two types of educational environ-

ments by providing a language similar to current popular languages while keeping it simple and easy to approach. I aimed to make something to teach concepts in a more traditional way than Alice or Scratch, but in an easier way than using a professional-level development language.

In addition, I sought to make programming possible literally anywhere a person might be. By placing a development environment on a phone and tablet, one can program on the bus, while taking a walk, or in between classes. Several applications for writing code on mobile devices do exist, but almost all of them rely on servers to do the actual compilation. While this may change, it is still a fact that phones and tablets rely on an internet connection for much of their functionality. Since Wolfscript's interpreter runs on the device, a user can run their applications with zero reliance on any other device or network.

I also hoped to build a development environment that would make writing code on a small touchscreen device a tolerable task. Using a simple text editor to write code on a tiny touchscreen is an exercise in frustration, and I knew it could be made easier with the right interface. The syntax of the language was also designed with easy typing in mind, shunning as many characters as possible that were hard to type on a mobile keyboard.

For the effort required to build an interpreter in two different languages, I felt it was best not to attempt to build Wolfscript for multiple mobile platforms. I chose Android as the mobile environment on which to build Wolfscript for two reasons. First, I was interested in contributing to a promising new mobile operating system, and since Android applications are written in what is essentially Java, I had experience that would allow for an easy transition. I had no interest in learning Objective C specifically for developing iOS applications. Second, I preferred Google's strategy with Android's marketplace to Apple's iPhone App Store because of Google's lower entry fee and more open attitude to developers.

The name "Wolfscript" comes from my middle name, Wolff, and the fact that it is an interpreted language like most scripting languages. I needed a catchy name, and the language spent a long time nameless while I contemplated the right name to give it.

1.2 Before Wolfscript

Before it had been decided that this project was going to be an honors topic, I spent several independent studies investigating the development of Android applications [2] [4] [5]. The first independent study, inspired by a Linear Algebra class, yielded a simple phone application called "Elementary Row Operations" that used elementary row operations to reduce matrices and find determinants and inverses. This application inspired the first ideas that eventually became Wolfscript.

After that simple application to modify matrices, I wanted more powerful matrix-modifying capabilities on my phone, similar to Matlab or Mathematica. However, I wanted something with a more feasible scope, though still with the capability to understand complicated mathematical expressions. The Android application allowed numbers to be entered as either fractions or decimals, and I wanted any future project to expand on that flexibility. Since I was familiar with graphing calculators and assumed that many people had at least a basic understanding of calculators, it made sense to begin there.

The first portion of what would become Wolfscript was the calculator. In it, I incorporated the convention from graphing calculators of keeping an “ans” variable, which contains the value of the last calculation performed. This feature remains in the language even now. However, as I progressed on the language, I shifted my focus away from programmable calculators, since I never intended to use GOTOs as in TI Basic. Instead I took inspiration from Java and scripting languages such as Python and Ruby. I adapted syntax from those languages due to their simplicity and clarity. The process of adding these language features lasted a long time, and specific details changed as I learned more. After I finished adding those features, however, Wolfscript mostly matched the form that it is in now. Since then I have focused on topics other than its architecture, such as finding faults and developing the IDE.

Chapter 2

The Interpreter

2.1 Overview

The interpreter portion of this project is comprised of the code that takes lines of Wolfscript code and executes them and the environment that the lines of Wolfscript code affect. For example, the interpreter maintains a list of variables, and assignments will change this list. The interpreter can exist completely independently of the IDE, and when it was originally written, it ran on Java in the command line. I still maintain an independent version of the interpreter that runs on any Java-enabled computer and uses standard console I/O.

The computer version of the interpreter is used for testing purposes as opposed to using the Android IDE because the command line is a faster, more flexible way to run files. This version also has a simple test suite, which takes a list of files to run, a file of simulated console input, and a file of correct console output. When the tests are run, the program compares the actual output with the correct output and displays an error if the two mismatch.

Since a good programming language needs useful, intuitive error handling, I have incorporated exceptions into Wolfscript that produce a stack trace when encountered. However, the interpreter currently has a limited set of exceptions that it recognizes, and it frequently has to resort to a default catchall exception. The system of exceptions needs to be expanded greatly to be useful.

As I developed Wolfscript, I decided to build each feature of the interpreter in stages, attempting to ensure that each stage was bug-free before

proceeding to the next stage. I started with the calculator stage, which provided the core of the language. I then created the interpreter itself and added variables. After variables were added, I created a second variable type, arrays. Once variables of both types could be defined, I added control flow statements to add the capabilities of an imperative programming language. Finally I added methods to incorporate procedural programming concepts.

2.2 Calculator Stage

The very first stage of the project was to build the calculator, which was to be the base level of all calculations as the language developed. I designed it to work as a command-line calculator, accepting input one line at a time and immediately returning the output of each line after it was submitted. It worked on desktop computers using Java, and it had no Android-related code.

The calculator operates in infix notation, and it works by maintaining two stacks, one for values and the other for operators. Operators, including parentheses, are arranged by precedence and therefore complicated expressions are allowed. The original version had operators that some programming languages only implement in methods, such as factorial, modulus, and exponents. Later in development, I added more operators such as logical and, logical or, equality, inequality, and comparison. A full list of operators is in the Beginner's Guide.

The calculator also has the unusual feature of allowing implied multiplication, which means a user is allowed to omit the multiplication symbol when using parentheses (2(4) vs. 2*(4)). This feature may eventually be eliminated, as it causes ambiguity in method calls, but for now it remains because when I added it I felt that the flexibility and intuitiveness of the calculator was paramount. Now, although this makes entering mathematical expressions very intuitive, it can cause confusion with method calls, which have a similar appearance.

Potentially the most important feature of the calculator is the way numbers are stored. All values are stored internally as fractions, using a custom-written BigFraction class to store Java BigIntegers as the numerator and denominator of the fraction. Every time a fraction is set, it is reduced using Euclid's algorithm to find the greatest common denominator, meaning that all fractions are always stored fully reduced. Using fractions allows for arbitrary precision, restricted only by the physical limits of the device. This

means that the result of all basic operations (except for fractional exponents) will contain the exact result of the calculation with no loss in precision.

2.3 Variables Stage

In the variables stage, I added the ability to store values in variables. At this time, variables were only numerical, as arrays did not exist. In keeping with the focus on the calculator, I added an "ans" variable, which automatically received the value of the last calculation performed. All variables are dynamically typed; no type is declared, and a variable is only created when a value is assigned to it. The number of statement types was expanded to two: expressions and assignments. An expression only assigns a value to "ans," while an assignment either creates a new variable or sets the value of an existing variable.

Also added in this stage was the project's interpreter component, which processes all statements and replaces all variables before passing expressions to the calculator component. It stores a sorted list, from shortest to longest, of all the variables. This is because the first way that variables were replaced was by traversing the list and replacing each variable as it was found. When a new variable is created, it is inserted into the sorted list of variables in the correct location.

After the insertion of the interpreter, statements gained the ability to change the state of the interpreter object, and the interpreter accepted only one statement at a time. Every statement returned a value so that "ans" would always contain the previous calculation. Assignment returned the value assigned to the variable, so after any assignment, "ans" is equal to the variable that has just been set.

I copied the code into an Android project and added a prototype user interface that allowed me to demonstrate that the calculator worked on Android devices. However, it remained a side project and was never incorporated back into the main project.

2.4 Arrays Stage

I added arrays to the interpreter as a second variable type, distinct from numbers. Arrays and numbers are still the only two variable types in the

language, and the only other type I intend to add is objects. Since variables are dynamically typed, a variable name can refer to either type over the course of a running program.

Arrays are variable length, and they are initialized with a set of initial values. An array can be initialized with zero initial values, but nothing can be retrieved from it until values are added. If a value is assigned to a non-negative index outside the range of the array, the array is expanded to that index and any intervening indices have their values filled with zero. Arrays can contain a mix of numbers and other arrays, making multidimensional and ragged arrays possible.

Arrays are implemented in a custom Array class that uses a Java ArrayList to store values of type Variable. The Variable class is a wrapper for both numbers and arrays, allowing both to be stored in the ArrayList. The interpreter adds all needed zeros when a value is assigned outside the array's size.

When a user attempts to access nested arrays inside an array, the recursive replacement of values requires that there be a temporary variable that represents the nameless arrays inside an array. Since expressions are stored as Strings even partway through being interpreted, an actual variable name needs to be placed into the string to represent the temporary variable. The variable name is currently called "NAMELESS," and if a user defines a variable by the same name, it will be overwritten whenever they access nested arrays. I intend to avoid this problem in a future update when I refactor the interpreter, but for now this is an unfortunate quirk of the language.

2.5 Control Flow Stage

The control flow stage took the interpreter from behaving like a calculator into becoming a Turing-complete programming language. This stage added statements that altered the behavior of the interpreter by allowing lines of code to run only if a condition was met. Prior to this, every statement was guaranteed to execute when it was passed to the interpreter.

The three statements added were "if" statements, "while" loops, and "for" loops. They require no parentheses or curly braces. However, they must be terminated with an "end" statement. I decided not to use whitespace to determine scope because the interpreter does not look at the overall program

but rather at individual statements, and measuring indentation in something like the interactive terminal seemed unintuitive. Instead, whitespace is trimmed by the interpreter, so indentation is irrelevant to proper operation.

When I added these statements, I had to consider the issue of scope. For example, if a variable is created inside an if statement, it must cease to exist when the if statement is exited. In order to do this, every control flow statement creates a new instance of the interpreter, which has its own set of variables.

When a program is run, a main interpreter is created, which exists for the entire duration of the program's run. When it encounters a control flow statement that evaluates to true, it creates a sub-interpreter and passes all statements that it receives straight to the sub-interpreter until the sub-interpreter encounters an "end" statement and notifies its parent. The main interpreter then destroys the sub-interpreter and takes back control until another control flow statement is encountered. If a sub-interpreter encounters a control flow statement while it is in control, it creates its own sub-interpreter. This happens recursively.

If a control flow statement evaluates to false, the active interpreter will discard the subsequent lines it receives while still checking for control flow statements. It only does this so that it can count how many interpreter levels the code could have descended to. This way, it knows which "end" statements to ignore until it encounters the one matching the original control flow statement.

Loops have an additional step. The first time the loop runs it acts mostly like an if statement. However, as the interpreter receives each line, it adds it to a String containing all the lines in the statement's code block. Upon reaching the end of the first iteration of the loop, the interpreter checks the original condition again and runs the stored code as long as the condition remains true. This means that when any parents of the interpreter pass the "end" statement for the loop to the interpreter, they must wait for all iterations after the first one to complete before regaining control. Therefore, all the lines from a program are only given to the interpreter once. Any time that any code is run after the first time it is received, an interpreter is re-running stored code.

For loops have two special statements. A while loop only has a condition, but a for loop has code that it runs the first time (the initialization code),

the condition, and the update (the code it runs at the end of each loop).

As I implemented loops, I originally intended every statement's return value to somehow "bubble up" to the top. Every type of statement prior to loops could easily display its return value in the interactive terminal because it was simply the value of a single variable. However, loops could generate huge amounts of return values; a loop with a million iterations and two lines would produce 2 million return values. This proved incredibly slow, so instead I set the return value for loops to be "Return value is too big" as a temporary placeholder. Eventually I changed it to an empty String, and if I refactor the interpreter in the future, I intend to make it return some indication of success, such as 1.

Around this time, I added input and output statements, which allowed for interactive programs to be written. Prior to this, the only way to interact with the interpreter was via the interactive terminal, which accepted input as lines to be executed by the interpreter. However, adding input and output statements allowed programs to be run from files; the input from the file was the code to be executed, and the only thing the user typed or saw was via "in()" or "out" statements.

2.6 Methods Stage

Methods behave similarly to control flow statements. When a method is first defined with the "def" keyword, it skips the lines of code, similar to skipped loops, except it also stores all the lines in a Method object as they are encountered. Methods are stored at the interpreter level in which they are defined, as are variables. Since both methods and loops store lines of code that will be replayed later, they function similarly in the interpreter. However, when statements for a loop are encountered for the first time, they are both run immediately and stored, while the statements in a method are merely stored and are not run until the method is called. In addition, the lines of code for a loop disappear immediately after a loop terminates, while lines of code for methods can only be run after its definition. When a method is called, a new interpreter is created and the stored statements are then run. Another major difference between loops and methods is the presence of parameters. The actual parameters that are passed to a method are assigned to variables in the method's interpreter with the same names as the parameter names in the method definition.

2.7 Future Stages

One feature that I expected to add before the project's completion was classes and objects. However, I found enough faults in the interpreter, especially with methods, that I decided to wait to add classes. I felt that the interpreter needed a complete refactoring before classes could be added, and if they were added to the current version of the interpreter, the resulting code would be incredibly difficult to debug. Since refactoring the interpreter may prove to be a significant task, it is the barrier to many of the features I would like to add to the interpreter. Most future features will require the concept of objects to be in place because of the design philosophy I am seeking.

Two examples of features that would require the presence of objects are graphics and networking capabilities. Although these features could potentially follow a procedural model, I feel that this is an outdated approach and is not a good model to follow when considering programming conventions today.

I did some research into OpenGL [1] as preparation for adding graphical capabilities to the language. After building test applications in C++ for desktop machines, I made a couple of proof-of-concept Android programs that utilized OpenGL for 2D drawing. When I reach the point of adding graphics to Wolfscript, it will likely use OpenGL.

I also want to eventually add a package/import system, so that both Java and Wolfscript libraries can be imported to applications. This way, features such as networking will not be included as an overwhelming amount of default functions in Wolfscript.

Some Android-specific features would access capabilities unique to mobile devices, but besides those, I intend all libraries to work the same on the computer as on phones or tablets.

Chapter 3

The IDE

3.1 Design Philosophy

When I first decided that I wanted to make it easier to program on a mobile device, I considered the two major disadvantages of mobile devices. First, phones and smaller tablets have a very limited amount of screen real estate to display a program. No matter how high-resolution the screen, the fact that the device is small makes it difficult to view small details and to select things using a touchscreen. Second, touchscreen keyboards can be difficult to use, and typing uncommon symbols requires drilling down via several modifier keys.

Considering these problems, I sought to design an IDE that would reduce typing and would save screen space when possible. To do this, I decided to avoid the traditional plain-text editing environment and made every line be represented internally as an object. This made it possible for the IDE to recognize different types of lines, color-code them accordingly, and represent them in alternate ways.

I designed the IDE to have colorful, distinctive buttons to reflect the feel of Scratch or Alice, which have color-coded draggable code blocks. However, I wanted my code buttons to have an obvious connection to the underlying code, and I wanted the lines to not restrict choices as a compromise for simplicity. Therefore, expressions themselves are still typed.

In order to make a compromise for small screens, I originally made it very difficult to obtain a big-picture view of a program by making it impossible to view nested code blocks alongside each other. I did this so that each

code block would have a reasonable number of buttons and the line count would not become overwhelming. However, this did cause some issues with navigation, so I added a button that allowed a user to expand the code inside a control flow statement as read-only. This allowed the child code to be smaller than it would need to be if it was editable.

3.2 Features

The IDE provides color-coded line types for expressions(gray), assignments(gray), comments(green), if/elseif/else(light blue), while/for(dark blue), def(yellow), and ret(orange). Each line type has a dialog window that reflects the inputs required for that line. For example, an assignment dialog window has two fields: the first is the variable name, and the second is the value to be assigned.

Every line in the IDE is represented as a button, and each line is placed in a vertical list of buttons. Pressing a button brings up a dialog allowing the line to be edited. However, control statement objects have a list of lines as their children, and in these cases pressing the button instead changes the view to a list of the children. This allows buttons to take up more space than a line of text normally would without occupying a massive amount of space. It also means that the buttons do not have to reflect scope through indentation.

When lines are edited, the text that sets that line type apart from the others (such as "if" for an if statement) is automatically inserted, meaning that users only have to enter the expressions they need to, without typing any boilerplate code.

An interactive terminal is also included in the IDE. It allows for quick code entry when a problem is too simple to require a program. The terminal can be used as a calculator, and it will be useful for the early stages of teaching the language. Students can immediately see the results of their efforts, making it easier to explore concepts.

The IDE has a built-in file browser that I originally wrote as an experiment before starting this project. It allows users to create and rename files or folders and save and load program data. The application will load all files, regardless of extension. If the IDE finds a defect in a program that makes it impossible to correctly load the program, it will display an error message showing the offending line.

3.3 Future Features

I intend the IDE to eventually keep track of all variables and methods that have been defined. This may end up sharing code with a refactored interpreter, because the IDE will essentially be performing the first step of interpreting the code. Keeping track of the variables will allow the IDE to quickly suggest variable names for expressions. Each text field in line dialogs will have a button that will provide a quick shortcut to existing variables, built-in functions, and methods.

The current method of inserting and moving lines of code is stiff and somewhat cumbersome. I intend to add line cutting, copying, and pasting, as well as the ability to change a line's type if it has children. This way, an if statement could be converted to a while loop without having to cut the code inside the former and paste it after the latter is created. I may also look into the ability to drag lines up and down to change their order by inserting a small strip along the right side of each code button.

Chapter 4

Educational Use

4.1 The Language

Throughout the previous sections, I have emphasized the educational potential in both the language and the IDE. I feel that the most important reason that the language can be educational is its simplicity. If something is simple to use, then learning it should come naturally. Therefore, instead of trying to find some sort of educational gimmick, I decided to reduce complexity as much as possible while reducing functionality as little as possible.

Classes that use languages such as Java or C++ have students write boilerplate code such as Java's "public static void main(String[] args)" before students are capable of understanding the purpose of such statements. In addition, console input and output are neither simple nor consistent. Programming students should not have to focus on the strange intricacies of language syntax. A much more important focus for a student of programming is language-independent programming concepts. With Wolfscript, I tried to make the language's syntax simple and intuitive enough to not interfere with teaching programming concepts.

The calculator focus at the lowest levels of the language allows it to respond similarly to a graphing calculator for basic calculations, and the interactive terminal allows users to type in expressions without assigning them to anything. This means that they can begin getting hands-on experience with Wolfscript with minimal prior experience.

4.2 The IDE

Installation and use of the IDE is straightforward. Once the application is placed on the market, users will be able to download it straight to their devices and immediately begin using it.

The bold, color-coded visuals of the IDE should make it easier to recognize different blocks of code, and the removal of the need to remember language syntax will make it easier to focus on program logic. Since each line type automatically inserts the keywords, all students have to remember is variable names and how to write expressions. Also, by automatically omitting the need to type end statements, students will not need to worry about the "mismatched curly braces" problem often encountered when learning other languages.

Part II

Beginner's Guide

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

Beginner's Guide 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!

Part III

Bibliography

Bibliography

[1] Edward Angel. *Interactive Computer Graphics: A Top-Down Approach with Shader-Based OpenGL (6th Edition).* Addison-Wesley. 2011.

[2] Google. Android developers. `http://developer.android.com/index.html` Retrieved May 6, 2013.

[3] MIT Media Lab. Scratch. `http://scratch.mit.edu/` Retrieved May 6, 2013.

[4] Mark Murphy. *Android Programming Tutorials: Easy-To-Follow Training-Style Exercises on Android Application Development.* CommonsWares. 2009.

[5] Mark Murphy. *The Busy Coder's Guide to Advanced Android Development.* CommonsWares. 2011.

[6] University of Kent. Greenfoot. `http://www.greenfoot.org/home` Retrieved May 6, 2013.

[7] Carnegie Mellon University. Alice. `http://www.alice.org/index.php` Retrieved May 6, 2013.