FiM++ 1.0 (Sparkle) language specification

FiM++ is an esoteric programming language inspired by the “friendship reports” written by Twilight Sparkle in the 2010 television show *My Little Pony: Friendship is Magic*. It follows the same general structure as Java, but uses full English (generally past-tense) words and sentence structure.

The intricate behaviors of this language (such as how addition works) are not specified in this document, as these are platform-dependent.

Challenge to Readers: Once you’ve read through this, write a program in this language that reads so fluently that you can't tell it's a program (i.e. Tara Strong or Rina-Chan could read it as a fluent Friendship Report). Show them to a below-linked editor and the most readable will be featured in this document!

Don’t like documents? Want to contribute to the language? See the **FiM++ Wiki**!

Note that this document is **nearly complete**; there are still some undefined parts of the language. These are listed in the “To Do” section or marked with respective comments. Feel free to submit your suggestions on how to do these, or what else we should do!

Table of Contents

Foreword

(Space reserved for SingleCrystal's foreword)

Purpose of this document

This document is created to list the barebones structure and syntax of the FiM++ programming language. When completed, this document will only list what is *required* to write a FiM++ program, no more, no less.
This document does *not* specify good practices for programming in FiM++, nor does it define any libraries that provide ease-of-use in this language.

Purpose and philosophies of FiM++

- Should be human-readable as fluently as a regular letter.
- Keywords, variable names, class names, and method names can all be multiple words.
    - Even though they have spaces in them, they must be thought of as a single phrase. (For instance, "Did you know that" is a single keyword, not a set of four individual keywords.)
- Lines of code must end in a punctuation, and newlines are not required (it can all be typed on the same line if needed).
    - A lone punctuation mark represents an empty statement which will simply not be run. This means that you can type multiple punctuation marks to end a line and they will have the same effect as a single one. (such as "...")
- This language is case-sensitive.
    - "`Applejack's Hat`" is *not* the same as "`Applejack's hat`"

Syntax used in this document

Code examples are given in with the following syntaxes:

- Plain text is explicit and must by typed as it is shown
    - `Dear`
    - `Today I learned`
- This language defines that many different words can mean the same thing (we call them synonyms). In this case, the different words are presented in square braces with solidi separating them.
    - `[said/wrote/sang]`
    - an ellipsis (either ⋯ or ...) is used to signify to carry on in this pattern
        - `[P./P.P./P.P.P./...]S.`
- Arbitrary text is in the form "`<arbitraryText>`"
    - `<superClass>`
    - `<methodName>`
- Items highlighted in light yellow with the comment "Keep in?" are proposals and not finalized
    - `I invented <syntax>.`
- Items in bright red and underlined highlight where compile errors would be thrown
    - `Dear: Lesson Zero`
    - `1.2.3`
- Items in bold magenta are problems in the language specification that must be fixed. Try to not use them until they are fixed.
    - "`We can't use apostrophes to concatenate strings because that's confusing.`"

Reserved arbitrary text names

These arbitrary text names always represent the same thing in every context. Anything that is not listed here is meant to be interpreted.

- `<nothing>`
    - Represents the absence of any text whatsoever, including whitespace. The inclusion of this in a list of synonyms means that the list is optional.
    - Note that **this is distinct from the `nothing` literal**.
- `<whitespace>`
    - Represents any text that is not colored by the editor, such as: spaces, new lines, tabs, et cetera.
- `<whitespace:<whitespace character name>>`
    - Represents a specific type of whitespace. This is platform-independent (For instance, "`<whitespace:newline>`" can be any OS's line feed. See Wikipedia's article on newline characters for more info)

- <punctuation>
  - Represents any character used to end a line of code.
  - [./!/?/‽/…/:]
- <solidus>
  - Represents a solidus (forward slash: "/"). This is used to distinguish an actual solidus from the separator used to list synonyms.
- <type>
  - Represents a data type.
  - [<number>/<array>/<character>/<string>/<bool>]
- <value>
  - Represents any data value, be it a variable, literal, or return value of a method.
- <value:<type>>
  - Represents a specific type of value. For instance, <value:bool> represents a strictly boolean value.
- <operator>
  - Represents an operator
- <operator:<operator name>>
  - Represents a specific operator. For instance, "<operator:else>" represents any operator that defines the "else" statement.

To Do

- A way to get and set values in an array
  - A way to get individual characters from character strings
    - Idea: treat Strings like character arrays
- Constructors
- Try-catch statement
- Modulo Operator (remainder)
- Create a compiler
  - Propositions for compiled files are already completed, working on interpreters

Files

Source

Source files are plain-text files that contain the code specified in this document. They must be saved with this type of file name:

- `<file name>.fpp`

A source file should be referred to as a letter to be sent.

Compiled

The code can be compiled into any C-style language (Including Javascript), but the recommended is Java's `.class` file type for cross-platform capabilities. A proprietary compiled file type is in the works (Current versions are `.burp` and `.FR`), but no complete Sparkle compiler yet exists.

Compiler proposals are ongoing at http://fimpp.wikia.com/wiki/FiM%2B%2B_Wiki:Proposals.

A compiled file should be referred to as a sent letter. Therefore, compiling should be referred to as "sending".

Comments

Inline

An inline comment must be preceded by the text "`S.`" with at least 1 "`P.`" preceding it. Everything after it is ignored. It is ended with a newline character.

- `[P./P.P./P.P.P./...]S.<arbitrary text><whitespace:newline>`

Java equivalent:

- `//<arbitrary text>`

Example:

- `P.S. I don't know how well this will go`
- `P.P.S. That said, I'm pretty excited!`
- `P.S.You don't need a space after the "S.".`
- `I said "something"! P.S. replace "something" with the actual variable.`
- `P.S.I said "something else"! <= old, unused code kept for reference`

Block

A block comment must be preceded with the text "`(`" and followed by the text "`)`". Everything within these is ignored.

- `(<arbitrary text>)`

Java equivalent:

- `/*<arbitrary text>*/`

Example:

- `(I can say anything I want in here!)`
- `I said "something"! (replace "something" with the actual variable)`
- `Dear **Princess Celestia**( and Princess Luna and Princess Cadence)**: Hey, Celly**!`

Classes

Classes are files that contain runnable code.

Declaration

A class must be declared by the text "`Dear`" preceding the superclass and list of implementations, followed by a colon, followed by the class name. The class name may have spaces. Class names should be in Title Case (each word should have a capitalized first character). Interfaces that it implements can be listed with the text "`and`". Class names must ***not*** be the same as variable or method names. Class names are case sensitive ("Princess Celestia" is ***not*** the same as "princess celestia").

- `Dear <Superclass Name>[<nothing>/and<whitespace><Interface 1 Name>/and<whitespace><Interface 1 Name>and<whitespace><Interface 2 Name>/...]:`

`<Class Name><punctuation>`

Java equivalent:

- `class <ClassName> extends <SuperclassName>[<nothing>/<whitespace>implements <Interface1Name>/<whitespace>implements<whitespace><Interface1Name>,<whitespace><Interface2Name>/...]{`

Example:

- `Dear` **`Princess Celestia: Letter One.`**
- `Dear` **`Princess Luna`** `and` **`Shining Armor`** `and` **`Cadence: An Update:`**

Ending

A class must be ended by the text “Your faithful student, ”, followed by the programmer’s name, followed by punctuation.

- `Your faithful student, <programmer name><punctuation>`

Java equivalent:

- `} //<programmer name>`

Example:

- `Your faithful student, Twilight Sparkle.`
- `Your faithful student, Kyli Rouge!`

Superclasses

Any class can be a superclass of another class. However, all classes must be subclasses of the `Princess Celestia` class. This means that if your class does not extend another class, it ***MUST*** explicitly extend the `Princess Celestia` class. Neglect to do this will throw a compile-time error.

Interfaces

Interfaces are files that contain data on how to build a class, but do not contain runnable code.

Declaration

Interfaces must be declared with the text “Dear “, followed by the name of the interface, followed by punctuation. The interior of an interface may only have method declarations, but not the bodies.

- `<Interface Name><punctuation>`

Java equivalent:

- `interface <InterfaceName>{`

Example (see the bottom of the document for a full example):

- **`Princess Luna:`**
- **`Me:`**

Ending

Interfaces must end like classes.

- `Your faithful student, Digit Shine.`

Methods

Declaration

Methods must be declared with the words “`I learned `”, followed by the class name, followed by punctuation. The method name may have spaces. The return value can be set by appending the text “`with`”, followed by the return type, before the aforementioned punctuation. Arguments can be defined by appending “using”, followed by the type of the variable, followed by the name of the variable. Method names must ***not*** be the same as variable or class names. Method names are case sensitive (“how to say Hello World” is ***not*** the same as “how to say hello world”).

If you include the text “Today ” before the declaration, it marks it as a main method. There can be multiple main methods, which will be run in order of declaration.

- [<nothing>/Today<whitespace>]I learned<whitespace>***<method name>***[<nothing>/<whitespace>[with/to get]<whitespace><type>][<nothing>/<whitespace>using<whitespace><type><whitespace>*<variable:parameter 1>*[<nothing>/<whitespace>and<whitespace><type><whitespace>*<variable:parameter n>*...]]<punctuation>

Java equivalent:

- public void <methodName>(){
- public <type> <methodName>(){
- public void <methodName>(<

Example:

- I learned ***the importance of oral hygiene***!
- **Today** I learned ***how to say Hello World.***
- I learned ***how to do math*** with a number.
- I learned ***how to take the sum of a set of numbers*** with a number using the numbers *X*.

Returning

A value may be returned with the words "Then you get ", followed by a variable name or value, followed by punctuation.

- Then you get [<variable name>/<literal>]<punctuation>

Java equivalent:

- return <returnValue>;

Example:

- Then you get 99!
- Then you get *the answer*!

Ending

Methods must be ended with the text "That's all about ", followed by the method name, followed by "!".

- That's all about <method name>!

Java equivalent:

- }

Example:

- That's all about ***how to do math***!

Calling

In order for a method to be useful, they must be called. They can be called inline as a variable or literal can be, or on their own when preceded by "I remembered " and followed by a punctuation.

- <method name>[<nothing>/<whitespace>using<whitespace>[<value>/<value><whitespace>and<whitespace><value>/...]]
- I [would/remembered]<whitespace><method name>[<nothing>/<whitespace>using<whitespace>[<value>/<value><whitespace>and<whitespace><value>/...]]

Java equivalent:

- <methodName>([<value>/<value>,<value>/...])

Example:

- Did you know that *Spike* was ***Spike's age***?
- I said ***how to write Hello World***!
- I remembered ***how to write Hello World.***
- I would ***say some choice words.***

Variables and Constants

Names

A variable name must **start** with a Unicode character that is **not** a character involved in making literals (such as: "`"", "' ", "true", "7", etc.) and must not **contain** any reserved keywords (such as: "Dear", "is", "I learned", etc.). Variable names must ***not*** be the same as class or method names. Variable names are case sensitive ("Applejack's hat" is ***not*** the same as "applejack's hat").

Examples of valid variable names

- Applejack
- Applejack's hat
- Team Fortress 2
- Somepony's true identity

Examples of invalid variable names

problems underlined and in red

- "reality"
- 99 jugs of cider
- true facts
- My Dear Fluttershy
- Something valuable I learned yesterday
- the song I sang

List of types

Below is a bulleted list of all the types of variables in FiM++, with sub-bullets listing the type names. These are represented throughout the document as "<type>". The "a" and "an" words ***do not*** apply to array declarations.

- 64-bit floating-point numbers
  - [<nothing>/the<whitespace>/a<whitespace>]number
- data arrays
  - [<nothing>/many<whitespace>]<type>[s/es]
  - Only works once; "numberses" throws a compile-time error.
- characters
  - [<nothing>/the<whitespace>/a<whitespace>][letter/character]
- character sequences (strings)
  - [<nothing>/the<whitespace>/a<whitespace>][word/phrase/sentence/quote/name]
  - These should be treated as arrays of characters by the compiler, but input by the programmer as quoted text.
- boolean values
  - [<nothing>/the<whitespace>/a<whitespace>/an<whitespace>][logic/argument]

Data arrays

Arrays must be dealt with in a special way in order to read and write from each slot in the array

Declaration

- Did you know that<whitespace><variable name><whitespace>[is/was/has/had/like/likes/liked]

- ```
  Did you know that cake has many names?
  cake 1 is "chocolate".
  cake 2 is "apple cinnamon".
  cake 3 is "fruit".
  I said cake 2.
  ```
  - Prints "apple cinnamon".
- ```
  Did you know that cake has the names "chocolate" and "apple cinnamon" and
  "fruit"?
  ```

Declaration

A variable must be declared with the following syntax. Variables can be multiple words. Variables should be completely lowercase. A variable can be made into a **constant** by using the text " always ". Note that "is" is allowed here, even though

it is used as the equal operator. This is the only instance where this can happen (initial writing), and every use of "is" on this variable is considered to be the equal operator.

- `Did you know that <variable name><whitespace>[<nothing>/always<whitespace>][is/was/has/had/like/likes/liked ]<whitespace>[<nothing><literal>/<type>/<type><whitespace><literal>/<variable>]?`

Java equivalent:

- `<type> <variableName>;`

Examples:

- `Did you know that Applejack likes numbers?`
- `Did you know that Trixie has the name "Trixie Lulamoon"?`
- `Did you know that Kyli Rouge likes the phrase "inventing an esoteric programming language based on MLP:FiM is fun"?`
- `Did you know that Spike's age is the number 10?`
- `Did you know that Princess Luna is always the phrase "awesome"?`

Reading

A variable is read from memory simply by referencing its name.

- `<variable name>`

Java equivalent:

- `<variableName>`

Examples:

- `I said variable name!`
- `If that's what she said then…`

Null values

A null value is a special value that represents an undefined space in the computer's memory. If you try to do something to it, a run-time error occurs. If you try to print it, the word "nothing" prints. You can read it just fine.

- `nothing`

Java equivalent

- `null`

Examples:

- `Did you know that sasquatch is a word? I said sasquatch.`
  - Prints "nothing"
- `Did you know that Rarity's catchphrase is the phrase "The worst possible thing"? Rarity's catchphrase became nothing. I said Rarity's catchphrase.`
  - Prints "nothing"

Literals

List of types

Below is a bulleted list of all the types of literals in FiM++, with sub-bullets listing the possible ways to represent them. These are represented throughout the document as "<literal>".

- 64-bit floating-point numbers (<value:number>)
  - `[<whitespace>/<punctuation>][0/1/2/3/4/5/6/7/8/9][<nothing>/<number>/.<number>/...][<whitespace>/<punctuation>]`
  - **Note**: The "." only works once; "123.456.789" throws a compile-time error.
- characters (<value:character>)
  - `[<nothing>/<type>]['/'/']<Unicode character>['/'/'][<whitespace>/<punctuation>]`
- character sequences (<value:string>)
  - `["/"]<Unicode text>["/"]`
- boolean values (<value:bool>)

    - [yes/true/right/correct]
    - [no/false/wrong/incorrect]
- null value (<value:null>)
    - nothing

Declarations

A literal must be declared explicitly

- 64-bit floating-point numbers
    - 1
    - 512
    - 31.25
    - the number 99.
- characters
    - 'A'
    - '6'!
    - the letter 'T'.
- character sequences (Strings)
    - "Princess"
    - the word "adorable".
    - "^_^"
- boolean values
    - yes
    - no
    - incorrect

Operators

Operators are explicit pieces of text that modify a value or return a value based on a comparison

Arithmetic

Arithmetic operators take in multiple numbers and return a number.
Standalone arithmetic operators must all begin with "I would ".
All arithmetic operators feature the ability to use prefix or infix notation. That is to say, you can either place the operator BEFORE the values being operated upon, or you can place it between two.
Arithmetic operators can be chained together like so:

- <value><operator><value><operator><value>...

ORDER OF OPERATIONS IS NOT NECESSARILY GUARANTEED.

Addition

This is the only arithmetic operator that accepts "and" as an infix operator. This means that if you mean to use prefix notation but forget the prefix, it defaults to addition.

- Infix: <value:number><whitespace>[plus/and/added to]<whitespace><value:number>[<nothing>/...]
- Prefix: add<whitespace><value:number><whitespace>and<whitespace><value:number>
- Increment: *<variable:number>*<whitespace>got one more

Java equivalent:

- <value>+<value>
- <value>+<value>
- *<variable:number>*++;

Examples:

- I said add 2 and 3.
- Did you know that *twelve* is 2 plus *ten*?
- I wrote 8 and 7 plus 3 added to 19.
- *Spike* got one more.

Subtraction

- Infix: <value><whitespace>[minus/without]<whitespace><value>
- Prefix: [subtract<whitespace>/the difference between<whitespace>]<value><whitespace>[and/from]<whitespace><value>

- Decrement: *<variable:number>*<whitespace>got one less

Java equivalent:

- <value>-<value>
- <value>-<value>
- *<variable:number>*--

Examples:

- I said subtract 5 and 7.
- Did you know that *Spike's age* is *Rarity's age* minus *Applebloom's Age*?
- I wrote the difference between *the number of books in the Canterlot Archives* and *the number of books in the Treebrary*.
- *Applejack* got one less.

Multiplication

- <value><whitespace>[times/multiplied with]<whitespace><value>
- multiply<whitespace><value><whitespace>and<whitespace><value>

Java equivalent:

- <value>*<value>
- <value>*<value>

Example:

- I said multiply 8 and 16!
- Did you know that *Junebug's daily profits* is *Junebug's hourly wage* times 8?
- I wrote *my favorite number* times 100.

Division

- Infix: <value><whitespace>divided by<whitespace><value>
- Prefix: divide<whitespace><value><whitespace>[and/by]<whitespace><value>

Java equivalent:

- <value>/<value>
- <value>/<value>

Examples:

- I said divide 8 and 2.
- Did you know that *Spike's age* is *my age* divided by 2?
- I wrote divide 2 by 9.

Variable modifiers

Rewriting

- *<variable name>*<whitespace>[[is/are] now/now [like/likes]/become/becomes]<whitespace>*<variable name>*

Java equivalent:

- <variable name> = <value>;

Examples:

- *Spike's age* is now 11!
- *Applejack* now likes 99.
- *the number of books in Twilight's library* becomes 1000.

String modifiers

- Concatenation **(Does not use keywords)**
  - [<literal:character>/<literal:string>]<value>[<literal:character>/<literal:string>]

User interaction

Output

- `I [said/wrote/sang/thought]<whitespace><value><punctuation>`

Java equivalent:

- `System.out.println(<value>);`

Examples:

- `I said "Hello World"!`
- `I wrote 99.`
- `I sang Winter Wrap-Up!`

Outputs are printed with a new line after each:

- `I said "Hello"! I said "World".`
  - prints "`Hello`
    `World`"

Input

Values are taken in as the type of variable given. For instance, if the variable specified is a number, then the input must be a valid number or an exception is thrown.

- `I [heard/read/asked]<whitespace><variable name>[<nothing>/the next <type>]<punctuation>`

Java equivalent:

- `<variable> = new java.util.Scanner(System.in).nextLine();`

Examples:

- `I heard Applejack's speech.`
- `I read the scroll!`
- `I asked the first number.`

Prompt

A prompt is a special type of user interaction that combines in

- `I asked <variable name><whitespace><value><punctuation>`

Java Equivalent:

- `System.out.println(<value>);`
  `<variableName> = new Scanner(System.in).nextLine();`

Example:

- `I asked Spike "How many gems are left?".`
  - Prints "`How many gems are left?`" and then waits for the user to input a number, which it stores in the variable "`Spike`".

Comparison

Comparison operators return a boolean value based on the values passed to it. If these values are of different types, they are cast to a string and then compared.

All comparisons start with the text "`[is/was/has/had]`", which shall be represented here by `<comparator>`.

Unlike Java, these can be standalone statements. Their return values will be ignored, and some compilers may choose to ignore the statement entirely. This is done for readability.

Equal

Returns "true" if and only if the values surrounding it are exactly equivalent. If these values are arrays, then the operator will only return true if they are the same length AND each corresponding value also is equal.

- `<value>`<comparator>`<value>`

Java equivalent:

- <value>==<value>

Examples:
- *nine* is 9
- *Spike's age* was 10
- *Junebug's profit* was *Rose's profit*

Not equal

Returns "true" if and only if the values surrounding it are not exactly equivalent. If these values are arrays, then the operator will return true if they are not the same length, or if any corresponding value is not equal.
- <value><comparator>[<whitespace>not/n't]<value>

Java equivalent:
- <value>!=<value>

Examples:
- *Celestia's age* is not 10
- *the number of cupcakes* isn't 0
- *Trixie's name* wasn't "Luna"

Less than

Returns true if and only if the value to the left is less than the value on the right. If these values are arrays, then the operator will return true if the length of the left is smaller than that of the right, or if they are the same length and the sum of the values in the left array is smaller than that of the right.
- <value><comparator> less than<value>

Java equivalent:
- <value><<value>

Examples:
- *the coolness of Rainbow Dash's dress* was less than 0.83
- *the number of cupcakes* is less than satisfactory
- "Apple" is less than "Nothing"

Less than or equal

Returns true if the value to the left is less than the value on the right, or if they are equal. If these values are arrays, then the operator will return true if the length of the left is smaller that of the right, or if they are the same length and the sum of the values in the left array is smaller than or equal to that of the right.
- <value><comparator> [no/not/n't] [more/greater] than<value>

Java equivalent:
- <value><=<value>

Examples:
- *the number of flowers in Junebug's garden* isn't more than 50
- 10 is not greater than the number jugs of cider on the wall

Greater than

Returns true if the value to the left is greater than the value on the right, or if they are equal. If these values are arrays, then the operator will return true if the length of the left is larger than that of the right, or if they are the same length and the sum of the values in the left array is greater than that of the right.
- <value><comparator> [more/greater] than<value>

Java equivalent:
- <value>><value>

Examples:
- Did you know that *truth* is the argument *anything Trixie can do* is greater than *anything you can do*?
- If *Spike's crush on Rarity* is more than *Scootaloo's crush on Rainbow Dash* then: I said "It's Tuesday".

Greater than or equal

Returns true if the value to the left is greater than the value on the right, or if they are equal. If these values are arrays, then the operator will return true if the length of the left is greater than that of the right, or if they are the same length and the sum of the values in the left array is greater than or equal to that of the right.

- `<value><comparator> [no/not/n't] less than<value>`

Java equivalent:

- `<value>>=<value>`

Examples:

- `If Celestia's greatness is no less than Luna's greatness then: I would praise Luna some more.`
- `Chrysalis' greatness isn't less than Twilight Sparkle's greatness`

Boolean Operators

Boolean operators allow operations to be done with truth values.

And

The "and" operator returns true if and only if both values surrounding it are also true.

- `<value:bool><whitespace>and<whitespace><value:bool>`

Java equivalent:

- `<value:bool>&&<value:bool>`

Examples:

- `Did you know that whether I went outside is the argument the sky is clear and Pinkie's tail is still?`
- `If I have no pants and I must scream then: I would scream without pants.`

Or

The "or" operator returns true if any one of the values surrounding it are true.

- `<value:bool><whitespace>or<whitespace><value:bool>`

Java equivalent:

- `<value:bool>||<value:bool>`

Examples:

- [to be written]

Exclusive or

The exclusive or operator returns true if and only if exactly one of the surrounding values is true.

- `either<whitespace><value:bool><whitespace>or<whitespace><value:bool>`

Java equivalent:

- `<value:bool>^<value:bool>`

Example:

- [to be written]

Not

The "not" operator inverts any boolean value. That is to say, if it is true, not returns false. If it is false, not returns true.

- `not<whitespace><value:bool>`
- `it's not the case that<whitespace><value:bool>`

Java equivalent:

- `!<value:bool>`

Example:

- `As long as it's not the case that Applejack has 5 (apples), I said "Keep going!".`

Branching statements and loops

Each of these starts and ends differently, but all of them have conditional statements that define how or when they are run.

If

- Starts with: `[If/When]<whitespace><value:bool>[<nothing>/<whitespace>then]<punctuation>`
- Ends with: `[That's what I would do/<operator:Else>]<punctuation>`

Java equivalent:

- `if (<value:bool>){`
- `}`

Examples

- `If Spike was 10 then:`
- `When 10 was not the number:`

Else

- Starts with: `[Otherwise/Or else]<punctuation>`
- Ends with: `That's what I would do<punctuation>`

Java equivalent:

- `else{`
- `}`

Switch

A switch statement allows you to combine several "If,then/else" statements that all reference one variable into one simple switch statement. The first line takes in the single variable to compare after the phrase "`In regards to `", and following lines define what to do in different cases (denoted by "`On the xth hoof`", where x is a number.), where the variable has different values. If the variable is none of these, it goes to the default case ("`If all else fails`").

- Declaration: `In regards to<whitespace><value:number>`
  - Declares that a switch statement has started and it is using the given value. This value is saved and guaranteed to not change while the switch statement is evaluating.
- Case: `On the<whitespace><literal:type>[<nothing>/st/nd/rd/th]<whitespace>hoof<punctuation>`
  - If the original value is exactly equal to this literal of the same type, then all the code between this and another case (or the end of the switch) is run. There can be indefinitely many case statements within a switch.
- Default case: `If all else fails<punctuation>`
  - If none of the other cases match the original value exactly, then the code between this statement and the next case (or the end of the switch) is run. **There can only be one default case per switch!**
- Ending: `That's what I did<punctuation>`
  - This is the end of the switch

Java equivalent:

- Declaration: `switch(<value:type>){`
- Case: `case<whitespace><literal:type>:`
- Default case: `default:`
- Ending: `}`

Example:

- ```
  In regards to Pinkie's Tail:
  On the 1st hoof...
  I said "That's impossible!".
  On the 2nd hoof...
  I said "There must be a scientific explanation".
  On the 3rd hoof...
  I said "There must be an explanation".
  On the 4th hoof...
  I said "Why does this happen?!".
  I would flail.
  ```
  ```

```
If all else fails...
I said "She's just being Pinkie Pie.".
That's what I did.
```

While

- Starts with: `[Here's what I did while/As long as]<whitespace><value:bool><punctuation>`
- Ends with: `That's what I did<punctuation>`

Java equivalent:

- `while(<value:bool>){`
- `}`

Do while

- Starts with: `Here's what I did<punctuation>`
- Ends with: `I did this [while/as long as]<whitespace><value:bool><punctuation>`

For

Counting

- Starts with: `For every<whitespace><type:[number/character]><whitespace><variable name><whitespace>from<whitespace><value:[number/character] 1><whitespace>to<whitespace><value:[number/character] 2><punctuation>`
- Contains: 0 or more statements
- Ends with: `[That's/That's] what I did<punctuation>`

Java equivalent:

- `for(<type:[number/character]> <variableName> = <value:[number/character] 1>; Math.abs(<variableName> - <value:[number/character] 2>; <variableName>--){`
- `}`

Example:

```
For every number x from 1 to 100,
I said x!
That's what I did.
```

Iterating

- Starts with: `For every<whitespace><type><whitespace><variable name><whitespace>in<whitespace><value:type array><whitespace><punctuation>`
- Contains: 0 or more statements
- Ends with: `[That's/That's] what I did<punctuation>`

Java equivalent:

- `for(<type> <variableName> : <value:type array>){`
- `}`

Example:

```
Did you know that Berry Punch likes the phrase "Cheerwine"?
For every character c in Berry Punch...
I said c.
That's what I did.
```

Example programs

Print the text “Hello World”

```
Dear Princess Celestia: Hello World!

Today I learned how to say Hello World!
I said “Hello World”!
That’s all about how to say Hello World!

Your faithful student, Kyli Rouge.
```

Add all the numbers from 1 to 100 and return the result

```
Dear Princess Celestia: Numbers are fun! (Start the class, naming it and its superclass)

Today I learned some math. (Start the mane method. This is the first part that runs)
I wrote the sum of everything from 1 to 100. (Run the “the sum of all the numbers from 1 to 100” method and print the return value)
That’s all about some math. (and now the program has finished running)

I learned the sum of everything from 1 to 100 to get a number. (start of a new method)
Did you know that the sum was the number 0? (Declare a new variable named “the sum” to be a number, and initialize it to 0)
Did you know that the current count was the number 0? (Declare a new variable named “the current count” to be a number, and initialize it to 0)

As long as the current count was no more than 100: (Start a conditional statement that loops while “the current count” is less than or equal to 100)
I would add the current count to the sum. (Increment “the sum” by the value held in “the current count”)
the current count got one more. (Increment “the current count” by 1)
That’s what I did. (End of the loop)

Then you get the sum! (Return the value held in “the sum”)
That’s all about the sum of everything from 1 to 100. (End of the method)

Your faithful student, Kyli Rouge. (End the class and sign your name)
```

Applejack’s Drinking Song (99 Jugs of Cider)

```
Dear Princess Celestia: Letter One.

Today I learned how to sing Applejack's Drinking Song.

Did you know that Applejack likes the number 99?

As long as Applejack had more than 1…
I sang Applejack" jugs of cider on the wall, "Applejack" jugs of cider,".
Applejack got one less. (Jug of cider)

When Applejack had more than 1… (Jugs of cider)
I sang "Take one down and pass it around, "Applejack" jugs of cider on the wall.".

Otherwise: If Applejack had 1… (Jug of cider)
I sang "Take one down and pass it around, 1 jug of cider on the wall.
1 jug of cider on the wall, 1 jug of cider.
Take one down and pass it around, no more jugs of cider on the wall.".

Otherwise…
I sang "No more jugs of cider on the wall, no more jugs of cider.
Go to the store and buy some more, 99 jugs of cider on the wall.".
That's what I would do, That’s what I did.

That's all about how to sing Applejack's Drinking Song!
```

Your faithful student, Twilight Sparkle.

P.S. Twilight's drunken state truly frightened me, so I couldn't disregard her order to send you this letter. Who would have thought her first reaction to hard cider would be this... explosive? I need your advice, your help, everything, on how to deal with her drunk... self. -Spike

An interface

Princess Luna:

I learned ***how to fly***.
I learned ***how to do magic***.
I learned ***how to raise the moon***.

Your faithful student, Kyli Rouge.

Blame

The blame for this language goes to the following people:

- Cereal Velocity (Blogger)
    - For popularizing this language on Equestria Daily
        - Twice
- DeftCrow (deviantART) / SingleCrystal (Google)
    - For being the first to start formalizing the grammar in a deviantART journal
- Draco-Icarus (deviantART)
    - For criticising the language, therefore ironing out kinks
- Evan Rittenhouse
    - For inventing a nice, solid way to use arrays
    - For ironing out many other syntactical problems
- Digit Shine (Pony 'sona) / Kyli Rouge (RL) / Supuhstar (deviantART, et al)
    - For starting this very document and ensuring that changes keep to the philosophy
    - For owning the wiki
- Parcly Taxel (deviantART)
    - For the first idea on how to properly implement arrays

See Also

Still confused? Other resources will be linked here.

- Official FiM++ Community Wiki. Learn and share information about this hilarious thing!
- Unambiguous list of reserved phrases. What's in a name? None of these!
- Unofficial proposals/brainstorming document. See what's on the cutting edge!