Brooke Bailey and Jessica Mott
CSCI 324: Programming Languages
25 April 2023

Ruby

Historical Account of Ruby

The programming language Ruby was developed by Yukihiro “Matz” Matsumoto in 1995. The driving factor behind the development of the language was to combine features of the already existing languages of Perl, Smalltalk, Eiffel, Ada, and Lisp in order to create a balance between functional and imperative programming. The design of Ruby is meant to highlight the scripting and object-oriented nature of several of the aforementioned languages by combining the two features into a simple object-oriented approach [5]. This choice allows Ruby to adopt the object-oriented language paradigm as well as others such as functional [8].

Language Elements of Ruby

Similar to most other programming languages, Ruby possesses fundamental elements through reserved words, data types, and data structures. Ruby has about forty keywords reserved for specific functionality when used in a program. The majority of these phrases are consistent with other languages; however, some diverge to support features particular to Ruby. The most noticeable differences are the words BEGIN, END, yield, and redo, which relate to Ruby’s usage of blocking. The BEGIN and END keywords label blocks of code to be run before or after all other code in the file, respectively. Using the redo reserved word, a block can be restarted, and the usage of yield allows the passing of additional instructions during the execution of a block or method [3].

In terms of distinct primitive versus structured data types, Ruby has none due to every value inherently being an object. This fact means that every declared value can be assigned its own properties and actions. However, Ruby does have basic types that are typical data type representations, such as numbers, booleans, and strings. One more distinct option in Ruby is symbols. This type is similar to a string except it is seen as an identifier, and once it is declared, it cannot be changed. Once the symbol is created, no modifications are allowed, and its size in memory is consistent. Symbols are useful for hash keys and representing method or instance variable names since they will not be changed in the program. Symbols can utilize a certain subset of string methods, such as changing to upper or lower case, and if necessary, a symbol can be converted into a string [7].

Syntax of Ruby

The syntax of Ruby was designed with the idea of simplicity in order to allow programmers to easily learn and utilize the language. The usage of English keywords and relatively simple structure of essential components such as loops, conditionals, and functions makes the language fairly easy to understand. All loops begin with their reserved keyword, such as “while” or “for”, followed by the expression or conditional control statement, and then by the loop body ending with the keyword “end”. Ruby also has an alternative way to achieve the same effects as a for loop by using the times method. For example, this method allows the programmer to type a statement such as “3.times {puts “Hello, World!”}”, which will print “Hello, World” three times [5].

Different from other programming languages, Ruby doesn’t have functions, but rather methods. These methods are defined by the keyword “def”, followed by the method name, and then any parameters in parentheses. Methods must be defined before a call is made to invoke the action. Similarly to functions, methods have a return value, which can be explicitly stated or assumed as the last evaluated expression [6].

Evaluation of Ruby’s Writability, Readability, and Reliability

Overall, as a language, Ruby contains several features that allow for successful writability. One of these elements is expressivity. Due to Ruby's flexible nature, programmers are able to define and redefine the functionality of basic operators and essentially manipulate the language to suit their preferences. For instance, in Ruby, the "+" operator symbolizes mathematical addition. However, if a user desires, they could create a class that defines the word "plus" to achieve the same functionality [5]. Additionally, since everything in Ruby is an object, writability becomes easier as the programmer can apply any method or attribute to any value. The coder does not need to remember which types can and cannot have certain features, allowing them to express their ideas with simpler lines of code. For example, by applying the "times" method to a print command, the result of a for loop can be achieved in a single line of code as opposed to three [5]. By having this adaptability, Ruby grants the programmer the ability to code in a manner that allows for endless options and the capacity to alter the language for each different application it is used for. Other aspects that improve writability are the usage of English keywords, the limited use of punctuation, and the absence of variable declarations. Instead of requiring the type of a variable to be declared with the variable name, such as "int x = 2", the programmer can simply say "x = 2". This characteristic, paired with the limited use of punctuation, such as no end semicolons after every statement, allows programmers to complete code more efficiently.

In terms of readability, Ruby possesses both advantages and disadvantages. By using familiar English keywords that correspond to straightforward meanings of their applications, readers of Ruby programs can likely know what the code is doing without having any prior background knowledge in programming. For instance, if an individual read the following code:

```
numbers = [1, 2, 3, 4]
if (numbers.include? ‘2’)
        puts(“2 is in the list!”)
end
```

they would likely be able to tell that there is a list of numbers and the conditional statement is checking whether or not the number two is present. The simplicity of the wording and punctuation causes little distraction and confusion for someone reading a Ruby program.

However, this simplicity can also cause readability-related problems. For instance, every block, method, loop and conditional statement ends with the reserved keyword “end”. Therefore, in more complex programs, especially ones with nested blocks, it can become difficult to identify which “end” is associated with which statement. Similarly, due to the fact that Ruby has such flexibility in terms of writing code, there becomes an absence of a set standard of what to expect in these types of programs. For example, since programmers can redefine operators and typical functions, a reader of the program needs to adjust to the definitions and cannot always follow a predefined set of expectations. Furthermore with the omission of variable types in declarations, it is not always clear what type a variable is. For instance, if a Ruby programmer defines a value as “x = 1”, the reader cannot necessarily distinguish between if the number is an integer or a floating point representation which can affect how they would interpret it in other parts of the program.

Finally, Ruby is a fairly reliable language. This programming language is extremely portable as it can function on a wide range of operating systems such as Linux, UNIX, macOS, and Windows [5]. By having this feature, Ruby programs can be transferred to and from different machines and still be able to perform its functionality. Additionally, Ruby has decent exception handling capabilities which allow it to easily identify and handle errors in the code. One way in which Ruby accomplishes this element is through its raise and rescue blocks. The raise block begins when an error occurs and temporarily stops the execution of the program. Then, the rescue block runs code to handle the exception and then returns back to the normal execution of the code. Apart from this typical process, Ruby also has other blocks it can use such as the retry and ensure statements. These pieces of code re-run the rescue block after the exception and provide code to always be run no matter the circumstance or error that occurs during the handling process, respectively [1]. Ruby’s relative simplicity in terms of writability and readability also improves its reliability since during creation of the program there are not a lot of complex rules and standards that need to be followed. Therefore, this quality makes it so that there is a higher chance the program is correct and has less mistakes.

Major Strengths and Weaknesses of Ruby

Ruby’s simplicity and flexibility are two of its most important strengths. By being a clear and understandable language, Ruby is a great option for beginner programmers and easy for more experienced coders to learn as well. Its capabilities of plainly being able to express ideas and functions through understandable reserved keywords and straightforward processes allow Ruby to help programmers accomplish problems quicker than other languages. Additionally, its flexibility makes it so that Ruby can be applied in a variety of contexts and fit each coder’s preferences and needs. Another advantage to Ruby is its connection to various imports and libraries, often referred to as Gems. One example of this idea is its graphical extension Ruby2D, which allows programmers to use Ruby to build visual displays, particularly in relation to gaming.

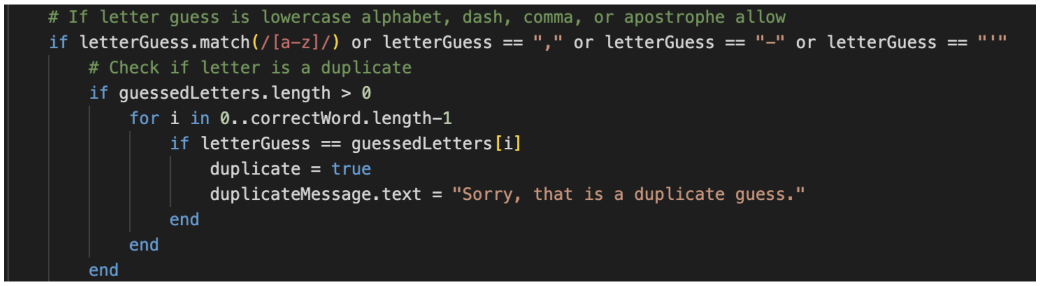
While Ruby allows for a wide range of programming styles and paradigms, this same flexibility makes it challenging to debug when things go wrong. It can lead to inconsistencies in code structure and behavior, making it harder to pinpoint where an error is occurring. For example, methods can be defined with different argument types and numbers, which can make it difficult to track down errors that occur when calling them. Another weakness of Ruby is the lack of documentation for gems such as Ruby2D. There is an adequate amount on how to apply all of the fundamentals of the gem. However, for some of the more involved processes such as updating a text element or resetting a screen there was less information which makes incorporating window elements more difficult.

General Program: Snowman

Overall, Ruby's key features made the implementation of the Snowman game easily achievable. The program utilized basic components of the language such as arrays and loops as well as more specific elements such as graphics and regular expression matching. Although Ruby supports a variety of data structures, arrays worked well in this application since only a small amount of data in the form of characters need to be tracked throughout the program. The main usage of this structure was to record which letters the user has guessed and compare them to the correct word in the game. Other fundamental elements of Ruby such as loops and conditional statements were used frequently to update the arrays and to check if the user's letter guess was both a valid alphabet character and if it was included in the winning word. Additionally, conditional statements were employed to check which stage of the game the user was at so that the appropriate part of the snowman was drawn to the screen as well as to know if the user wanted to play again after ending a game. Finally, the element of reading in files was vital to the game as it allowed the program to access a large dictionary for game play. Being able to access these core aspects of the language was crucial in the development of this program and made the process fairly straightforward.

The implementation of the Snowman program also highlights some more distinct aspects of Ruby. One of these features is the usage of Ruby 2D for the graphical display of the game's user interface. This form of graphical elements is mostly meant to produce visuals in digital games that use Ruby as the main language. The Snowman program exploits several of the features Ruby 2D has to offer, including color, text, shapes, and input. The combination of these aspects allowed the text-based nature of the game to be transformed into a visual experience. The color and shape characteristics were applied in illustrating the state of the snowman during the game as well as for hint and replay buttons. The textual features aided in displaying the letter aspects of the game such as the user's letter guesses and where correct guesses are located in the winning word. Two different types of user input through both the keyboard and mouse allowed functionality to accept both user guesses as well as recognize if any of the buttons were selected. Overall, Ruby 2D granted a reasonably easy implementation of a graphical game.

The other main unique feature of Ruby that the Snowman program emphasizes is regular expression matching. This quality is utilized in the accepting of user input for which letters are being guessed. Statements check to see if the character that is inputted is one of the alphabet keys. By having access to this feature, it simplifies the writability and readability of the program by not having to add dozens of conditionals to check if the letter matches one of the letters. Regular expression matching allows the program to analyze the same result with one simple line of code.



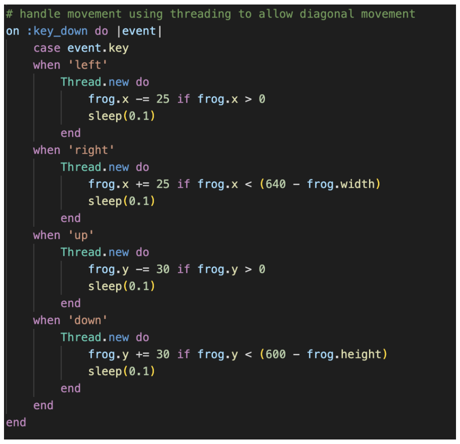
```ruby
# If letter guess is lowercase alphabet, dash, comma, or apostrophe allow
if letterGuess.match(/[a-z]/) or letterGuess == "," or letterGuess == "-" or letterGuess == "'"
    # Check if letter is a duplicate
    if guessedLetters.length > 0
        for i in 0..correctWord.length-1
            if letterGuess == guessedLetters[i]
                duplicate = true
                duplicateMessage.text = "Sorry, that is a duplicate guess."
            end
        end
    end
```

Creative Program: Frogger

The Frogger-style game uses features of Ruby to implement a time-based winning leaderboard. The basic premise of the game is for the player to move the frog from the bottom of the screen to the top as quickly as possible. However, there are different types of obstacles to traverse in order to make this possible. The player must avoid hitting the moving crocodiles and cars, or they will be brought back to the bottom of the screen. To reach the end, there are two moving logs that the player must hop onto for the frog to reach the other side. Time will run until the frog reaches the top of the screen, so the player must win in order for the game to finish. The time will be compared to a leaderboard of previous players with the three lowest times. This program implements a hash table, regular expressions, BEGIN blocking, graphics, audio, and threading.

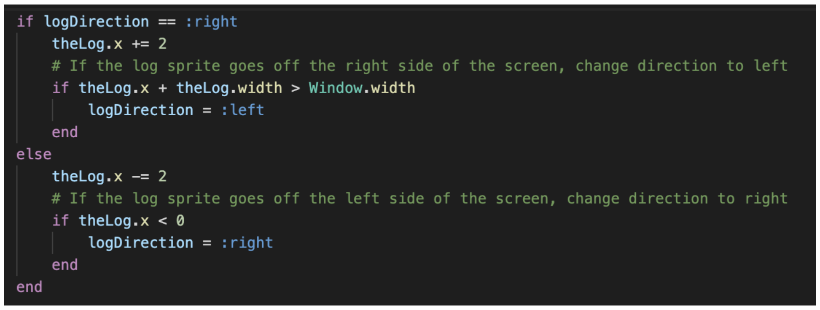
The program uses regular expressions to validate user input for a username that the player must enter in order to play the game. A text file called "usernames.txt" is read to populate a hash table with past player data. The formatted usernames are stored in the hash table, which helps with storing the timed score, preventing repeat names, and displaying the leaderboard after completion. All of this information is stored inside of the BEGIN block. The BEGIN block is a block of code that runs first as the file begins execution [4]. It is used to perform setup tasks before the rest of the program runs. This ensures that the code for creating the hash table and setting up the username is executed only once, regardless of how many times the program is run.

The Ruby2D gem has an object called a sprite, such as a PNG frog, that can be moved and interacted with across the screen. The movement is implemented using threads, which allows the game to continue processing while the frog is moving. Moving objects that are not user-controlled will move on their own accord, separate from user input. When a key is pressed (up, left, right, or down), a new thread is created to move the frog in the desired direction, and the thread sleeps for a short period of time (0.1 seconds in this case) before completing [2,9]. This ensures that the frog moves smoothly and does not jump too quickly across the screen. The use of threads allows the program to interpret two inputs simultaneously. For example, if the player presses the up and side keys at the same time, the frog will move diagonally in that direction.



```ruby
# handle movement using threading to allow diagonal movement
on :key_down do |event|
    case event.key
    when 'left'
        Thread.new do
            frog.x -= 25 if frog.x > 0
            sleep(0.1)
        end
    when 'right'
        Thread.new do
            frog.x += 25 if frog.x < (640 - frog.width)
            sleep(0.1)
        end
    when 'up'
        Thread.new do
            frog.y -= 30 if frog.y > 0
            sleep(0.1)
        end
    when 'down'
        Thread.new do
            frog.y += 30 if frog.y < (600 - frog.height)
            sleep(0.1)
        end
    end
end
```

The language made the implementation of Frogger relatively simple. With the Ruby2D gem, features such as audio, graphics, and sprites worked well for game integration. The window and sprite built-in attributes allowed for the movement of objects such as the logs to bounce from one side of the screen to the other without needing user input. For example, these lines of code move a log across the screen by setting the attributes ':left' and ':right'.



```ruby
if logDirection == :right
    theLog.x += 2
    # If the log sprite goes off the right side of the screen, change direction to left
    if theLog.x + theLog.width > Window.width
        logDirection = :left
    end
else
    theLog.x -= 2
    # If the log sprite goes off the left side of the screen, change direction to right
    if theLog.x < 0
        logDirection = :right
    end
end
```

The hardest part to implement and understand is the update do loop. This is an infinite loop in coding that is not a major problem. The concept of an infinite loop is what makes it difficult to understand. The contents of the loop will run until the window is closed [10]. This means that if an element of the window should only be run once, it either needs to be done before the loop or within a conditional in the loop. For example, the stoppage of the timer must be done within the loop because the window needs to be open to check the location of the frog. However, the timer cannot be stopped an infinite amount of times, so a Boolean must be used to ensure that the timer continues until the frog reaches the top of the screen. This leads to confusing and dense code for checking every possible condition for each sprite on the screen.

References

1. ankita_saini. (2018, October 12). *Ruby: Exception handling*. GeeksforGeeks. https://www.geeksforgeeks.org/ruby-exception-handling/. Accessed 2023 April 10.
2. *How to use Ruby Threads*. RubyGuides. (2020, January 31). https://www.rubyguides.com/2015/07/ruby-threads/. Accessed 2023 April 20.
3. *Keywords*. Keywords - documentation for Ruby 2.2.0. (n.d.). https://docs.ruby-lang.org/en/2.2.0/keywords_rdoc.html. Accessed 2023 March 24.
4. Khanna, M. "How to Use Ruby BEGIN and END Blocks." *Scout*, Scout APM, (23 Sept. 2021). https://scoutapm.com/blog/ruby-begin-end. Accessed 2023 April 20.
5. *Ruby*. About Ruby. (n.d.). https://www.ruby-lang.org/en/about/. Accessed 2023 March 1.
6. *Ruby Methods*. Techotopia. (n.d.). https://www.techotopia.com/index.php/Ruby_Methods. Accessed 2023 March 24.
7. Shivi_Aggarwal. (2022, May 23). *Ruby: Data types*. GeeksforGeeks. https://www.geeksforgeeks.org/ruby-data-types/. Accessed 2023 April 4.
8. RubyCademy, T.-. (2019, February 15). *Ruby is a Multi-paradigm programming language*. Medium. https://medium.com/rubycademy/ruby-is-a-multi-paradigm-programming-language-49c8bc5fca80. Accessed 2023 April 20.
9. *Threading*. Class: Thread (Ruby 2.5.0). (n.d.). https://ruby-doc.org/core-2.5.0/Thread.html. Accessed 2023 April 20.
10. *The Window*. Ruby 2D - The window. (n.d.). https://www.ruby2d.com/learn/window/ Accessed 2023 April 20.