CMPS 203 - Final Project Report

Matthew J. Howard
Computer Science and Engineering
University of California, Santa Cruz
matthoward@ucsc.edu

Alexander S. Williamson
Computer Science and Engineering
University of California, Santa Cruz
alswilli@ucsc.edu

June 9th, 2019

1 Introduction

In this project, we implemented a records interface and key-value store with support for adding, removing, finding, and undoing removals of records in 7 different programming languages. Specifically, we implemented our project in Java, Python, Scala, Dart, Lua, Kotlin, and F#, of which we were familiar with Java and Python prior to the project. In the following sections, we highlight our motivation for implementing this project, provide details of our implementation design, and summarize our accomplishment.

1.1 Motivation

Our motivation for implementing a key-value store with accompanying interface stems from the desire gain practical, hands-on experience with various aspects (data structure implementation, I/O, data manipulation, etc.) of several interesting and unique programming languages. Further, we were motivated to study the learning curve of language syntax, paradigms, and typing systems as well as language performance to properly evaluate how suitable each language is for a potential large-scale code base.

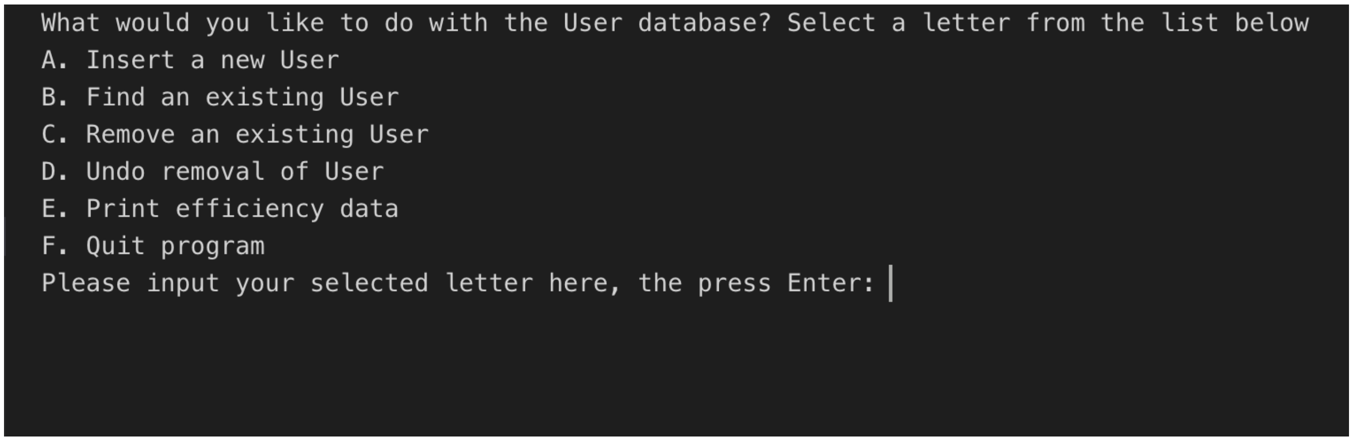
1.2 Project Description

1.2.1 User Records

For records, we designed a simple 'User' entry class which holds four string parameters: *firstname, lastname, username*, and *password*. When a new 'User' entry is created and added to the hash table, the *username* field is utilized as the unique key, and thus duplicate usernames are not permitted. The User entry class simulates practical record entries that consist of several string fields.

1.2.2 Command-Line Interface

Initially, when first loading up the program we created, the hash table is filled with pre-defined User entries from a text file. For each line in the text file, we list the four separate class variables for each User object that we want to parse and load. Each of these values is comma separated for convenience. Once all of these predefined User objects are loaded into the hash table, we prompt the user to respond to a command line interface menu of commands, as shown in Figure 1. If a user chooses option A from this list of commands, they can specify a new User object to add to the hash table using inputted text for each class variable. All other functions require simple responses from the user to perform actions with the hash table. A loop allows the user to continue responding to the options in from the menu until they decide to quit the program.



```
What would you like to do with the User database? Select a letter from the list below
A. Insert a new User
B. Find an existing User
C. Remove an existing User
D. Undo removal of User
E. Print efficiency data
F. Quit program
Please input your selected letter here, the press Enter:
```

Figure 1: Command-line interface menu options

1.2.3 Hash Table

For the key-value store implementation, we implemented a hash table with separate chaining. More specifically, the hash table consists of a finite number of buckets that are index by an entry hash, each containing a 'head' entry that may or may not be empty, depending on the state of the hash table. Each bucket head entry serves as the entry point to a linked-list of nodes that are stored within the bucket, and contains one field 'next' which represents any entries that were added to that bucket after the head. Further, if an entry hashes to a bucket that already has a head element, the new entry will be set as the 'next' element of the first entry in the linked list that has an empty 'next' element. In Figure 2, we highlight this separate chaining process for hash tables and the formation of the linked-list when hash collisions are found. To decide where hash entries are to be stored in the hash table, we apply a cryptographic hash function to the *username*, which produces a unique number based on the string value. This value also can be reproduced later to allow for the hash table to find the bucket the key being requested is stored at. Since this value will be within a larger range of numbers than the usual size of the hash table, we then apply a modulus of the size of the hash table to the value to generate a final hash index for the key. From here, the key-value pair to be stored becomes the data in the node of the linked list for the assigned bucket. In order to access this data later, each bucket keeps a 'head' entry that links the subsequent entries together. When the number of hash table entries exceeds 70% of the number of available buckets in the hash table, the entire data structure is resized with twice the number buckets as before. We allow for several types of operations within the hash table. These operations include 'PUT', 'GET', 'REMOVE', and 'UNDO-REMOVE'. Starting with 'PUT', this function takes in a key and a value and inserts them into the hash table. We built the hash table to allow for any type of input, but for our implementation we use *username* as the key and the corresponding User object as the value. Moving on the 'GET', this function takes in a key to retrieve and retrieves the value corresponding to the key in the hash table. If the value does not exist due to an invalid key or faulty key-value store, then an error message is printed to the console. Next, the 'REMOVE' function takes in a key and removes its corresponding value in the hash table. Similar to the 'GET' function, if the User object being searched for is not found, an error message is printed. Additionally, if the value is found, the key-value pair is added to a local stack for potential retrieval later on in case the user wants to undo a removal. This is where the 'UNDO-REMOVE' function comes in, and simply pops the top entry from the stack and uses 'PUT' to insert it back into the hash table.

1.3 Stack

To implement our stack, we use a linked list and define several important functions as well. These functions include 'ISEMPTY', 'PUSH', 'PEEK', and 'POP'. Beginning with the 'ISEMPTY' command, the function returns whether the size of the stack is greater than 0 or not. This function is used to check whether any removals exists to be undone when called 'UNDO-REMOVE' on the local stack in the hash table. The function 'PUSH' take in a key-value pair and insert them as a new entry to the top of the stack. The function 'PEEK' then returns the most recently added entry, while the function 'POP' removes the most recently added entry instead.

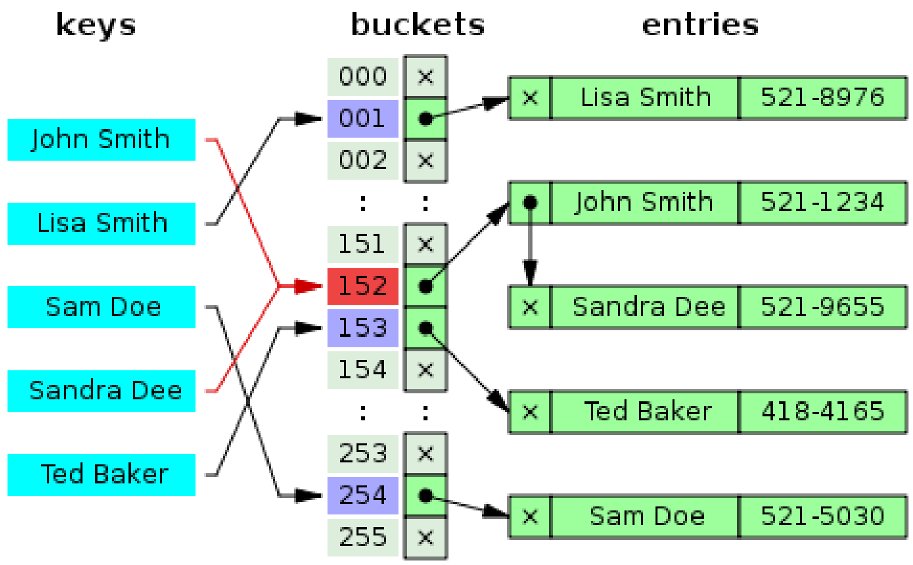


Figure 2: Hash-table with separate chaining (Source: `https://en.wikipedia.org/wiki/Hash_table`)

1.4 Summary of Accomplishment

We achieved many of our starting goals with the completion of this project. Most importantly, we were able to build a working implementation of our project using each one of the languages. Furthermore, we gained quite a bit of practical experience with their different paradigms and typing methods. Moreover, we achieved a much better understanding of the advantages and disadvantages to using each of the language features. Finally, the simultaneous hands-on experience with many languages provided better understanding of the nuances and bugs in other languages when debugging them.

2 Comparison Analysis of Languages

2.1 Code Design Comparisons

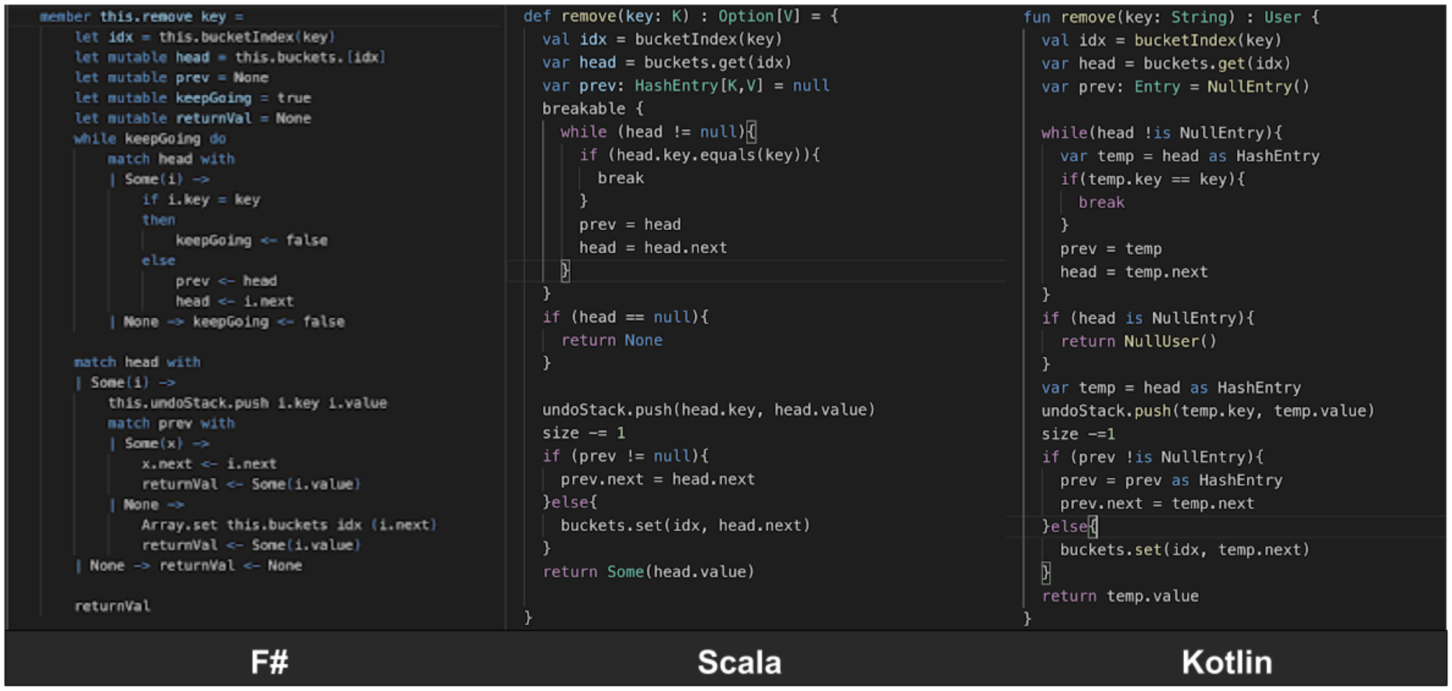
Many of our languages utilize several different programming paradigms other than just pure object-oriented design. A summary of the main paradigms and typing systems for each language can be seen in Table 1. Due to the variety of of programming paradigms found in the languages used, we often had to adapt code to be able to represent object-oriented logic that we were used to coding already. For example, in the more functional languages like Scala, F# and Lua, we had to create similar structures to 'structs' from C to allow for class functionality. Additionally, we had to adapt for certain syntax in languages to perform more simple operations. In Figure 3, we highlight some of the changes we had to make across 3 different languages to achieve the same functionality for the 'REMOVE' function in the hash table. Starting with F#, we had to utilize the 'match' statement several times in order to account for places where we checked if the value of a variable was equal to 'null' on 'None'. Option types also had to be utilized as a return value in both F# and Scala in order to handle when the variable return type changes to null. More over, F# supports mutable and immutable variables, so we had to define that functionality for each variable declaration, unlike the other two languages pictured. Scala and Kotlin were much more similar other than a few subtle differences in syntax involving conditional statements.

2.2 Language Performance Comparisons

We also compared the performance across all languages from 3 different hash table operations, 'PUT', 'REMOVE', and 'UNDO-REMOVE', which can be seen in Figures 4, 5, and 6, respectively. With each plot, we observed how the function in question performed over time with 1000 iterations of the function. Starting with Figure 4, we notice that

Language	Paradigm	Typing
Python	Multi-Paradigm	Dynamic
Java	Object-Oriented	Static
Scala	Functional / Object-Oriented	Static
Dart	Object-Oriented	Static
Kotlin	Object-Oriented	Static
Lua	Multi-Paradigm	Dynamic
F#	Functional / Object-Oriented	Static

Table 1: High-level comparison of programming paradigms and typing systems for each language used during implementation.



```fsharp
member this.remove key =
    let idx = this.bucketIndex(key)
    let mutable head = this.buckets.[idx]
    let mutable prev = None
    let mutable keepGoing = true
    let mutable returnVal = None
    while keepGoing do
        match head with
        | Some(i) ->
            if i.key = key
            then
                keepGoing <- false
            else
                prev <- head
                head <- i.next
        | None -> keepGoing <- false

    match head with
    | Some(i) ->
        this.undoStack.push i.key i.value
        match prev with
        | Some(x) ->
            x.next <- i.next
            returnVal <- Some(i.value)
        | None ->
            Array.set this.buckets idx (i.next)
            returnVal <- Some(i.value)
    | None -> returnVal <- None

    returnVal
```

```scala
def remove(key: K) : Option[V] = {
  val idx = bucketIndex(key)
  var head = buckets.get(idx)
  var prev: HashEntry[K,V] = null
  breakable {
    while (head != null){
      if (head.key.equals(key)){
        break
      }
      prev = head
      head = head.next
    }
  }
  if (head == null){
    return None
  }

  undoStack.push(head.key, head.value)
  size -= 1
  if (prev != null){
    prev.next = head.next
  }else{
    buckets.set(idx, head.next)
  }
  return Some(head.value)

}
```

```kotlin
fun remove(key: String) : User {
  val idx = bucketIndex(key)
  var head = buckets.get(idx)
  var prev: Entry = NullEntry()

  while(head !is NullEntry){
    var temp = head as HashEntry
    if(temp.key == key){
      break
    }
    prev = temp
    head = temp.next
  }
  if (head is NullEntry){
    return NullUser()
  }
  var temp = head as HashEntry
  undoStack.push(temp.key, temp.value)
  size -=1
  if (prev !is NullEntry){
    prev = prev as HashEntry
    prev.next = temp.next
  }else{
    buckets.set(idx, temp.next)
  }
  return temp.value
}
```

Figure 3: Code for remove in 3 different languages: F#, Scala, and Kotlin

the shape of the curves plotted resemble a step function. This shape is due to the time it takes to resize the hash table buckets as more and more entries are added. Lua and F# seem to come out as the most efficient language in this case, almost avoiding the same step function shape found in the other languages graph due to its speed, while Scala and Dart ended up being very inefficient, especially during resizing. Moving on to Figure 5, we see a similar trend to the one found in Figure 4. Scala and Dart seem to be very slow with removal, while F# and Lua perform much faster operations. Interestingly, some weird shaping from Kotlin and Dart appeared past 500 operations, which may be do to some other background operations occurring during runtime on the test machine. Finishing up with Figure 6, many of the languages perform within 0.5ms of each other. However, Python seems to have a much steeper curve that we are unsure about. Again, this outlier could be attributed to other background tasks being performed on the host computer rather than poor optimization from Python itself. Overall, it seems that Lua and F# are the best languages for complitation speed, while Scala, Dart, and Python are some of the worst. Perhaps languages with more Object-oriented paradigms are less efficient in general.

2.3 Challenges

We encountered a slew of challenges when coding in the various languages for this project. One big challenge we had to overcome constantly was learning the correct syntax for each language. Many of the ways simple operations were performed in each language differed by a wide margin. Common types of operations that differed include file and standard input and output, Loops, if-then-else statements, array operations, list operations, pattern matching, the

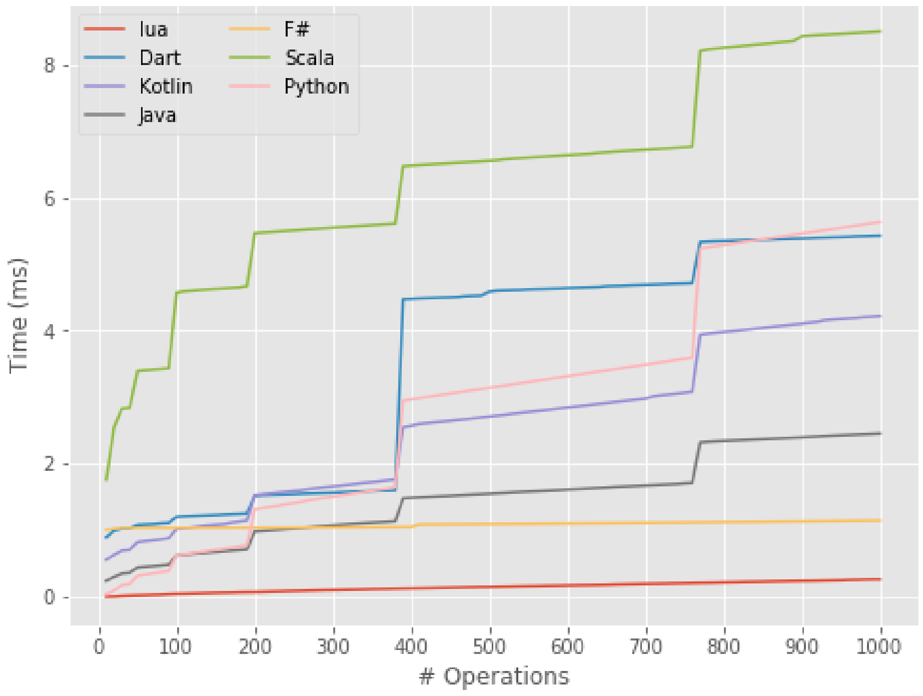


Figure 4: Performance graph for the put command across all languages

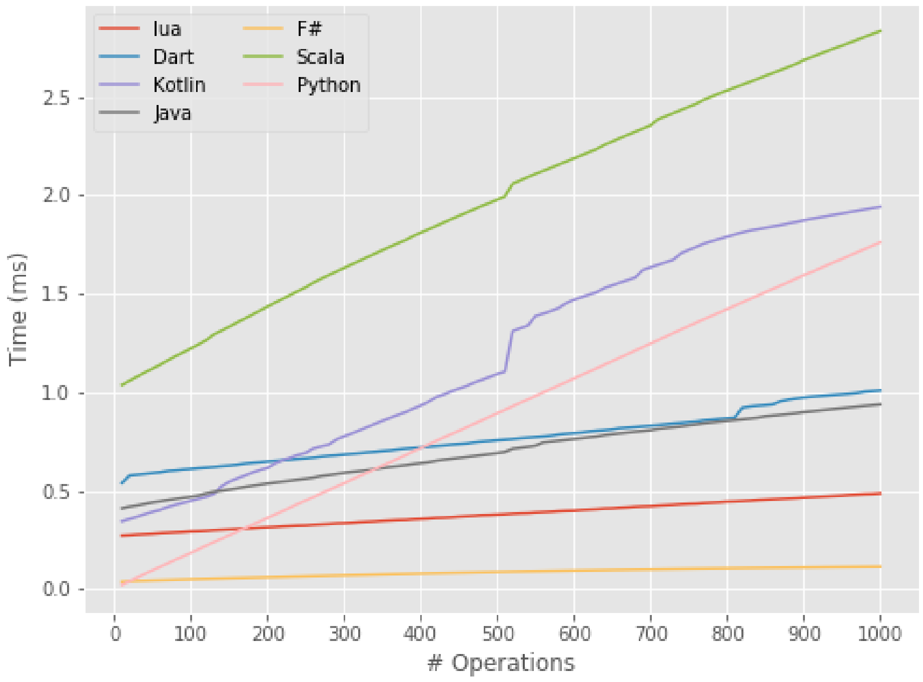


Figure 5: Performance graph for the remove command across all languages

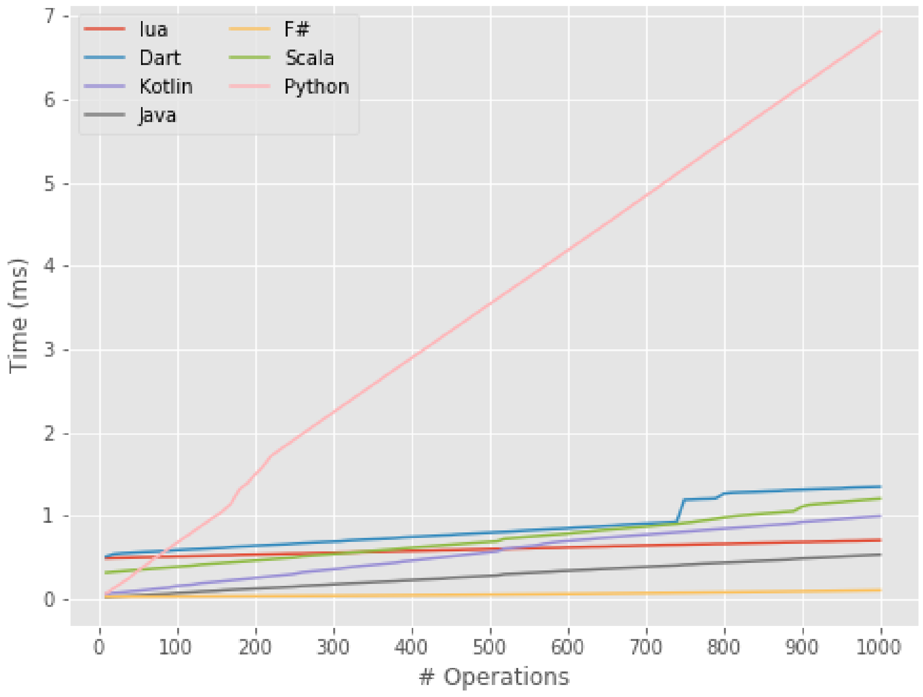


Figure 6: Performance graph for the undo-remove command across all languages

various variable access and function declarations, handling of null values, and option types. We has to utilize each of these operations in several different ways throughout the project, so we were constantly trying to figure out how to use them to generate the same logic across all languages. Another big challenge we faced was implementing object-oriented design for certain languages. As Table 1 shows, not all the languages we used were purely object-oriented, meaning that we had to come up with work-arounds in these types of languages when trying to emulate object-oriented logic already implemented in other languages. Some of the more specific operations we had to overcome here was properly typing and casting variables when type inference was unsuccessful, and handling several difference approaches to mutating class variables and/or properties. A third challenge we encountered was having to debug code in certain languages that had less documentation, where it was difficult to quickly resolve errors (such as array indexing) and required a fair amount of diving into the main documentation of the programming language.

3 Conclusion

In conclusion, from implementing this project we learned that program require different abstractions depending on the paradigms and features of the programming language. When choosing a language to code an application in, programmers should decide based on breadth of documentation, familiarity with similar programming designs (pointers or no pointers, functional or object-oriented, etc.), and performance on the specific task to be accomplished. From coding the 7 languages in this project, the languages we prefer the most are Python, for its ease of use, flexibility, and familiarity, and Scala, for its interoperability with Java and powerful functional features.

4 Project Code

The final project code can be found at the following Github commit-hash link: `https://github.com/matthewjhoward/cmps203/commit/d5c0a49b9ccf01784d5f887db07e5197169f3ee5`

5 References

References

[1] "Python-Tutorial." https://www.tutorialspoint.com/python/. Accessed May 2019.

[2] "Java Tutorial." https://www.tutorialspoint.com/java/. Accessed May 2019.

[3] "Scala Tutorial." https://www.tutorialspoint.com/scala/. Accessed May 2019.

[4] "Dart Programming Tutorial." https://www.tutorialspoint.com/dart_programming/. Accessed May 2019.

[5] "Kotlin Tutorial." https://www.tutorialspoint.com/kotlin. Accessed May 2019.

[6] "Lua Tutorial." https://www.tutorialspoint.com/lua/. Accessed May 2019.

[7] "F# Tutorial." https://www.tutorialspoint.com/fsharp/. Accessed May 2019.