**Familiarity Review**

**Name:** Jeremy James Alkire

**Date:** March 14th, 2020

**Week:** 10

**Coding Topic:** MVC

**Description of Understanding:** The MVC is a three-part architectural design pattern used for developing user interfaces. The three components are the model, view, and controller. The model manages the data and logic of the application. The view displays a visual representation of the data. It handles both input and output and is what the user interacts with. The model and view components do not communicate with each other directly, they are both managed by the controller, which acts as a medium between the two. The controller is essentially the engine of the program. It calls methods from the view and the model and passes data between them as necessary.

**Explanation of Changes:**

You were right, my model shouldn’t have been making print statements, as that’s the view’s responsibility. Years ago, I got into the habit of using constructors to state when an object has been created, and often liked implementing visual feedback for my setter methods. It helped me (as a programmer, not an end-user) to verify that things were functioning as I had intended. Those were habits developed as a young programmer who wasn’t writing code within any specific design pattern, or for any intention of deployment to a user. So, it wasn’t suitable for an MVC example! To resolve this, I removed all print statements from my model class.

In addition to this, I created two new methods (one in the model and one in the view) to apply the recommendations in your feedback.

*isConfigured()* was my approach to implementing your suggestion: “Rather it (the model) should return some kind of success message to the controller”. Since the model has no main method, it does not call this method on its own. It relies on the controller to call this method itself, and once the controller has done that, the method returns a true or false value boolean variable to tell the controller whether the athlete object was configured successfully. Provided that the method returns *true*, the controller then calls the view’s *announceCreation()* method, which prompts the view to notify the user of success.  
  
In the table below, I detailed how the application functions, as well as how responsibilities are separated between the model, view, and controller. This was updated to reflect the changes listed above.

**Also Integrated with:** Use Case Diagrams, Use Case Documents, Sequence Diagrams

|  |  |  |  |
| --- | --- | --- | --- |
| **File** | **Git Link** | **What should I be looking for?** | **Sandbox or Your code?** |
| RequestData.java |  | This class operates as the **view** within the MVC architecture. The method *requestAthleteData()* outputs a prompt to the user that requests that they input data into the console. That data is handled and stored in method variables and returned to the method caller. *requestAthleteData()* calls other methods, such as *getWeight()* and *getHeight()* which communicate with the user via the console to request and accept user input. Lastly, it has an *announceCreation()* method that notifies the user once they’ve successfully created & configured an Athlete object. The **view**, in this case, functions as a text-based UI. By itself, it does nothing. The **controller** calls its methods. The **view** is the medium through which the user can interact with the application. | My code. |
| Athlete.java |  | This class operates as the **model** within the MVC architecture. It specifies the attributes of the Athlete object. It has private non-static variables, public getters & setters, and a public void configuration method *configureAthlete()* that allows you to set the value of all variables with a single function call (and three arguments.) In addition to this, it has a public boolean method called *isConfigured()* (lines 48-57) that determines whether the name, age, and weight variables have been assigned values other than their default.  By itself, this class does nothing. It merely holds data. Object instances of this class get created elsewhere (via the **controller**). The **controller** can then call methods from the model object to set and get the values of its variables. The **model** never provides any output and the user never interfaces with it directly. | My code. |
| Engine.java |  | This class operates as the **controller** within the MVC architecture. It creates an object instance of the Athlete (**model**) class, calls *requestAthleteData()* from the RequestData (view) class and stores its output in a string variable. It then splits the string into a list of strings by comma (,) characters. From there, it calls *configureAthlete()* from the **model**, and passes each element in the list to **model’s** method via its parameters. After all of the above, it calls the **model’s** boolean *isConfigured()* method (line 30) to determine whether the Athlete object was configured properly. If so, it tells the **view** to notify the user of this by calling the **view’s** *announceCreation()* method (line 31). | My code. |

**Coding Topic:** Java Collections

**Description of Understanding:** A collection, sometimes called a container, is simply an object that groups multiple elements into a single unit. Collections are used to store, retrieve, manipulate, and communicate aggregate data. These typically represent items that form a natural group, such as telephone directories. A collections framework is a unified architecture for manipulating collections. Collection frameworks contain interfaces, implementations, and algorithms.  
  
Interfaces are abstract data types that represent collections. These allow collections to be manipulated independently of the details of their representation, and generally form a hierarchy. Nearly all interfaces inherit methods from the Collection superclass and pass these methods down to their implementations. Core interfaces in Java include: Collection, Set, List, Queue, Deque, Map, Sorted Set, and Sorted Map.

Implementations are concrete implementations of the collection interfaces. They are reusable data structures, and children of the interfaces, whom are their parents, and whom they inherit methods from. General-purpose implementations in Java include: HashSet, HashMap, ArrayList, ArrayDeque, TreeSet, TreeMap, LinkedList, LinkedHashSet, and LinkedHashMap.  
  
Algorithms are methods that perform useful computations, such as searching, sorting, inserting, and deleting elements inside of a collection. They’re described as polymorphic, meaning they’re able to be used in many different implementations. These are typically implemented via methods, such as add() and put().

**Explanation of Changes:**

There are frankly too many to list. However, in summary:

I added a substantial amount of code to most of these examples. I decided to format each example as somewhat of a little explanatory tutorial on the collection that it was intended to demonstrate. In each example I tried to use methods that made that collection unique, usually as a product of the interfaces it implemented that other collections do not.

I chose to use TreeMap for a leaderboard list as that’s a situation in which ascending key order makes sense, and where you also need key-value pairs (key = placement, value = name), unless you’re using objects to store both. Additionally, there were a wealth of methods at my disposal for interacting with the leaderboard list, such as getting a sub map from a specified key range. And furthermore, if a leaderboard is very large, TreeMap’s smaller memory usage over other maps (such as HashMap) may be of benefit when performing large operations.

I chose to use a HashSet to list the winners of a sports team when I did not want duplicate elements, as the set was going to be used (hypothetically) to determine who gets trophies. Having two instances of the same player name would result in two trophies being assigned to one player, which would be unintended. This would prevent the risk of user error (duplicates, specifically) when adding values into the collection. Furthermore, this allows cloning operations on the set (not the elements inside of it) in case it needs to be cloned for usage elsewhere. This is backed by a hash table, which means the order of elements are not guaranteed (at least not in the standard sense: not alphabetical / sequential) and are instead determined by the hashing algorithm. At the end of this example I printed the hash codes of three different sets, one of which was a clone of another, to showcase that a cone would have the same hash code as the set it was cloned from.

The examples themselves and my descriptions of the other collections detail what makes each of them unique, and when someone may want to use them. I promised that I’d get on call with another of my group members tonight to try and help them with their last familiarity submission before the end of week 10, so I’m submitting this now to do that before it’s too late to be of further assistance, as it’s already 10:37 PM here, otherwise I’d have gone into more detail!

**Teaching Video:**

**Starting at:**

**Also Integrated with:** ArrayList is used briefly in my controller class (Engine.java) in my MVC coding examples.

|  |  |  |  |
| --- | --- | --- | --- |
| **File** | **Git Link** | **What should I be looking for?** | **Sandbox or Your code?** |
| ArrayListExample.java |  | This class utilizes the ArrayList implementation of the List interface.  ArrayLists take up as much memory as is allocated for the specified capacity, regardless of whether elements have been added. Adding/removing from a position in an ArrayList that is anywhere but the end requires it to shift elements over.  ArrayLists have a capacity that is at least as large as the list size, but typically larger. It is always as large as the array used to store elements in the list. The size of an ArrayList grows automatically as elements are added to it. It is not fixed, like an array.  Unlike a normal array, in which methods are accessed with [], elements in an ArrayList are accessed by methods or iterators.  ArrayLists only support object entries, not primitive data types. If you insert a primitive variable into an array it is converted to an object. | My code. |
| CollectionExample.java |  | This class also utilizes the ArrayList implementation of the List interface.  This is a simple example of a collection. It is a collection of Strings. It contains a conversion constructor which initializes a new collection and uses some of the basic methods inherited from the Collection interface.  All true collections are descendants of the Collection interface. | My code. |
| HashMapExample.java |  | This class utilizes the HashMap implementation of the Map interface.  Unlike a TreeMap, it only implements Serializable, Cloneable, and Map, and does not implement SortedMap or NavigableMap.  Like TreeMap, HashMaps are not true collections as they do not implement the Collection interface.  HashMaps are backed by a hash table. They permit null values and null keys and make no guarantee as to the order of its elements. Programmers must pay close attention to capacity and load size when creating and using HashMaps, lest they run into performance problems.  The capacity is the number of buckets in the hash table and the load factor is a measure of how full the hash table can get before its size is increased automatically.  A HashMap requires more memory than is needed to hold its data. If it gets close to 70-75% full, it gets resized and its entries get rehashed. The order of elements is determined by a hashing algorithm.    A HashMap is an optimal choice when you want key-value pairs, the backing of a hash table, and the ability to store null values and null keys. | My code. |
| LinkedListExample.java |  | This class utilizes the LinkedList implementation of the List and Queue interfaces.  A LinkedList stores pointers to the next and previous elements and is often considered more memory intensive than other alternatives, such as ArrayList.  As such, LinkedLists are often considered more performance heavy than ArrayList, though this is sometimes a controversial statement.  Elements in a LinkedList, by default, are inserted in FIFO (first-in-first-out) order. However, because LinkedList inherits from Deque, it can be configured to insert elements in LIFO (last-in-first-out) order.  LinkedLists implement Serializable, Cloneable, Iterable, Collection, Deque, List, and Queue.  As such, LinkedLists have access to a wealth of methods inherited from each of the above interfaces. “Operations that index the list will traverse it from the beginning or the end, whichever is closer to the specified index.” (<https://docs.oracle.com/javase/7/docs/api/java/util/LinkedList.html>)  LinkedLists are an optimal choice for when you want to use your list as both a Queue and a List, as it has inherited from each of those interfaces. As such, it has access to methods that many other collections do not. Many methods operate on either the head or tail of the queue. When FIFO or LIFO order is important to you, you can consider using a LinkedList. | My code. |
| TreeMapExample.java |  | This class utilizes the TreeMap implementation of the Map interface.  A TreeMap is not a true collection as it does not implement the Collection interface. Unlike a HashMap, elements in a TreeMap are sorted according to the natural ordering of their keys, or by a Comparator.  This implements NavigableMap and SortedMap, as well as Cloneable and Serializable.  TreeMaps do not contain null keys but may contain many null values. TreeMaps are not as fast as HashMaps. TreeMaps can save memory and use less of it than HashMaps as it only uses the amount of memory needed to hold its elements.  A TreeMap is the preferred choice over other collections when we want key-value pairs, objects to be sorted in their natural order, and are uncertain about memory limitations, or want to simply save memory. Barring key-value pairs, it has the above advantages over HashMaps as well. | My code. |
| HashSetExample.java |  | This class utilizes the HashSet implementation of the Set interface.  A set is a collection that cannot contain duplicate elements. This is particularly useful for situations in which duplicates would not make sense.  Elements are not ordered sequentially or alphabetically. In a HashSet in particular, the set is backed by a hash map.  It’s possible to clone a set, however, this only clones the set itself, it does not clone the objects contained inside of it. It will retain references to the same objects in the original set, though. | My code. |

**Diagram**: Sequence Diagram

**Description of Understanding:** A sequence diagram demonstrates the interaction between messages and objects within a system. It displays these relationships in a sequential manner, so that people can clearly see how messages flow throughout the system. Components of a sequence diagram include objects, messages, and sequence numbers, which indicate how methods are called one after another.

**Explanation of Changes:**

You mentioned that it looked like the view was calling the getData() method on the user. I looked at my diagram and I can see what you were referencing.

To resolve this, I removed the arrow drawn from the View to the User on the fifth step of my sequence diagram, the one that displayed the call of the getData() method.

You recommended showing that a page is displayed where the user can add/modify data. I considered adding a new participant to the diagram called “Console” to display this, and having the getData() line drawn from the View to the Console, but then it’d appear as if it was calling a method on the console. Ultimately, I decided against it, because this approach could cause even more confusion, and would also not be technically accurate. getData() is a method owned by the view, but it is also called by the view, so it is calling the method on itself.

To demonstrate this, I used a Self-Message arrow, as explained at <https://www.javatpoint.com/uml-sequence-diagram>

**Teaching Video:** None

|  |  |  |
| --- | --- | --- |
| **File** | **What should I be looking for?** | **Example**  **Or Your code?** |
|  | The link on the left links directly to an image of my sequence diagram.  This diagram sequentially shows how data flows through the program by outlining what methods are called and what data is passed to them as arguments. It also displays what data is returned by methods (such as getters) and attempts to showcase this flow of information chronologically. | My code. The same code used to demonstrate the MVC architectural pattern. |