**Lesson 7 – Bringing it all Together**

* Need to make the GUI respond to user activity
* So far we clumped our code into 3 basic groups
  + **Model**
  + **View**
  + **Startup**
* **Model**
  + Everywhere we manipulate our data
  + Describes how the program functions
  + Methods are used to manipulate data stored in instance variables
* **View**
  + All the display (no manipulation)
  + This code is separate from the Model
  + Responsible for displaying both the components for user interaction and the information about the state of the data contained in the Model
  + Methods are used to create and update the display
    - Update Method accesses attributes from the Model Class
* **Startup**
  + Very limited functionality
  + Creates Instance of the View and Model and links them so they can work together
  + This is the Class with a Main Method
  + A Frame containing the View is displayed
* **Events**
  + Events generally work the same way for all components
  + When something happens to a JComponent an event object is created describing the event
    - When it occurred
    - What keys (if any) were pressed
    - Which JComponent created it
  + We need to create a new class that requires this event object as a parameter
    - This class then updates the data in the model which indirectly updates the data in the view
      * This Class (or set of classes) are called the Controller Class
* **Controller Class**
  + A part of the GUI that contains code that responds to user actions/events
  + A separate class is created for every JComponent that will be manipulated
  + Must extend Object class and implement a Listener specific to the type of interaction it responds to
    - Usually ActionListener or MouseListener
  + Contains instance variables for the model and JComponent and one (or more) method(s) that respond(s) to the event(s)
  + Name of the Controller Class is named for the JComponent that is being manipulated
  + Pass in the Model and JComponent to the Class Constructor
  + The Methods in the class are those that are required by the Listener’s interface
    - These methods run at any time the JComponent is being manipulated
    - They are passed the event object that was generated by the JComponent
      * Example: ActionEvent Object
  + Before we can make this work, we must register the Controller
    - This is done by using the add(type listener)Listener to a JComponent
      * Provides an instance of the controller as an argument
    - Since most GUI’s have several controllers the View Class should have a helper method to add listeners
      * Must be called in the View’s Constructor
* Since the Controller calls methods in the Model to manipulate the data there needs to be a way for the Model to inform the View that data has been changed
  + This is done by adding an update() method in the View Class
    - This method redraws the GUI using accessor views from the Model Class
    - All methods in the Model that modify data must end by calling the View’s update() method
* We must now link the View and Model Class
  + This is done in the Startup Class
* **Startup Class**
  + Creates an instance of the Model and View Class
    - Creates a JFrame to display the View
  + The Model must have an attribute for the View
    - The Model Class contains a public method to instantiate the View
      * Allows the creation of multiple views for the same model
  + View Class
    - Has an attribute for the Model
      * Constructor instantiates the model and calls its setGUI method so the Model knows which View its working with

**Lesson 8 – Arrays**

* **Variable** = Symbolic name for a location in memory that is used to store data
  + Example:
    - int age;
      * Sets up a location in memory to store the integer value of a person’s age and calls this location age
* **Array** = Location in memory that stores a set of items of the same type under the same name
  + Example:
    - int[] studentAges;
      * Sets up a location in memory to store the integer values of a set of student ages and calls this location studentAges.
* **Advantages of Arrays**
  + Single location/name for a large amount of data
  + Individual items (elements) may still be accessed
  + Operations on entire data group can be carried out using repetition structures
* **Array Facts**
  + Items in an array are referred by the name of the array and their position in the array
  + An element’s position in the array is called an index, starting at 0
  + Items in an array can be any of Java’s Primitive Datatypes as well as any Object
  + Items in an array must all be of the same type
  + An array cannot have its length altered after creation (length must stay static)
* **Using Arrays (3 steps)**
  + **Create the Array**
    - **Array Declaration**
  + **Give the array a size**
    - **Memory Allocation**
  + **Setting Values of the Array Elements**
    - **Array Initialization**
* **Array Declaration** 
  + Write the type and the variable name
  + The type of the variable is the same as the type of information stored in each element
  + To indicate that it is an array add the [] after the type
    - Example: int[] studentAge;
* **Memory Allocation**
  + Declaring an Array does not create it, it only gives it a name to an array that will be created during memory allocation
  + Allocating memory for the array is like using a constructor for an object
    - E.g *studentAge = new int[12];*
      * Specifies that the array will have 12 elements
      * JVM allocates memory for the array at this time
      * Number inside the bracket can be anything that evaluates to an integer (e.g. another variable)
* **Array Initialization**
  + The array can now be filled with values, one element at a time
    - Process is sped up when using repetition structures
  + Initializer List:
    - *String[] studentNames;*
    - *studentNames = new String[]{“Shalee”, “Bob”};*
    - ***OR***
    - *int[] studentAges = {16, 17, 18};*
* **Accessing Elements** 
  + Elements in an array are referred by the name of their array and their position in the array
    - Position of an element is called an index
    - Array index must always be a non-negative integer value
  + To access all elements in an array you can use a repetition structure
    - Mainly a for loop
* Programmers often use more than 1 array at a time, copying values from one array to another which often leads to errors if not done properly
  + Common mistakes:
    - array1 = array2;
      * This will cause 2 variable names pointing to the same array object
      * The original array1 is now lost
      * Editing the contents of either array will edit the contents of both arrays
    - **To copy data from 1 array to another**
      * Individual elements must be copied from 1 array to another using a repetition structure

**Lesson 9 – Array Sorting**

* Variety of ways to compare algorithms
  + **Algorithm Complexity** = How confusing the code is to write
  + **Algorithm Structure** = Basic method we will use to sort the array
  + **Computational Complexity** = How hard it is for the computer to sort the data
  + **Memory Usage** = How much extra memory is required for the algorithm
  + **Array Stability** = How likely is the array to stay sorted
* **Algorithm Complexity** 
  + Comparison of the overall complexity of the code required to write the algorithm
    - SelectionSort, BubbleSort, InsertionSort are simple
* **Algorithm Structure**
  + Comparison of the general style or type of algorithm
    - Can be:
      * Swap (Switching paired items)
      * Merge (divides array into sorted and unsorted parts)
      * Tree (places data in a binary tree)
      * Other (generally more complex)
    - Bubble Sort
      * Compare the first two elements in the array and swap them if they are out of order and continue this process until you reach the last entry
        + Last entry is now the largest value in the array
      * Makes n-1 comparisons in the first pass
        + n-2 comparisons in the second pass and so on
        + Makes a total of n(n-1)/2 comparisons

Makes n^2/2 -n/2 comparisons

For large values we say the algorithm is O(n^2)

* + - * Swap Based (Swapping individual elements)
    - Selection Sort
      * Given an array of entries, search for the smallest element and swap it into the first entry then search the remaining elements for the second smallest element and put it in the next position. Continue this process until the array is fully sorted.
      * Swap Based (Swapping individual elements)
    - Insert Sort
      * Takes advantage of pre-sorting
      * It requires fewer comparisons than bubble sort unless the list is backwards
      * If the array is already sorted it only requires one comparison per element (n comparisons)
      * Method can be used to insert new elements to a sorted array at any time
        + This is most often how insertion sort is used in the real world
      * Merge Based (merging unsorted to sorted side)
* **Computational Complexity** 
  + Comparison is based on the processing time of the algorithm
    - Based on the number of comparisons made since it can be difficult to compute the number of swaps made
    - Processing time can change because of the complexity/amount of disarray of the array to be sorted
      * Results in analyses of best, worst, and average case scenarios
  + Generally, programmers want to know the efficiency of their programs when there is a large amount of data (n)
    - Evaluated using O Notation
      * A mathematical language for evaluating the running-time of algorithms
      * Gives an approximate rate of growth for an algorithm (e.g. linear vs quadratic)
      * We typically evaluate the worst-case scenario
      * Examples:
        + Bubble Sort with an array of 10 elements

45 total comparisons

In general, there are N-1 comparisons each time

Formula: N \* (N-1)/2

Which has a growth rate of O(N^2) comparisons

* + - * + Selection Sort with an array of 10 elements has the same computational complexity as Bubble Sort but it sorts much quicker (10 swaps for 10 elements)
        + Insert Sort

This algorithm is twice as fast as Bubble Sort since on average only half the number of comparisons is performed for each pass before the term is found

* **Memory Usage**
  + Used to compare the amount of memory required to sort the array (not including the size of the actual array)
    - All 3 algorithms require very little extra memory
      * Only 1 variable is used for temporary storage during data swapping
* **Stability**
  + Used to see if data reminds in the same order when sorted for multiple attributes
    - Example:
      * If you sort a list of people by first name that have already been arranged alphabetically by last name, do the names with the same first name remain sorted alphabetically by last name?
        + If yes, it is stable if not it is not stable
    - All 3 algorithms are stable

**Lesson 11 – 2D Arrays**

* Sometimes data comes in a format that is two dimensional
  + Examples:
    - Tables
    - Maps
    - Images
  + This data needs to be stored in a 2D array
    - Rows is 1 array and Columns would be another array

**Lesson 12 – Collection Classes**

* Collection Classes are other ways to manipulate sets of data
  + **Lists**
    - Example: ArrayList, LinkedList, etc.
  + **Sets**
    - Example: HashSet, etc.
  + **Maps**
    - Example: TreeMap, HashMap, etc.
* **List** 
  + An **ordered** collection of elements which may contain duplicated values
    - **List is Dynamic but Array is Static**
* **Set**
  + An **unordered** collection of unique elements
* **Map**
  + An **ordered** collection of **keys** that are linked to an associated **value**
* **List Class**
  + **ArrayList**
    - Keeps elements in order
      * Any element can be accessed by its index
    - Dynamic
      * Size of the ArrayList can be modified
    - Can only be objects (no primitives)
    - Can only be 1D
* **Set Class**
  + **HashSet**
    - Elements are not in any order
      * Elements do not have an index
    - Dynamic
      * Size of HashSet can be modified
    - All elements must be unique
      * Duplicate elements are ignored when added
    - Can only be 1D
* **Map Class**
  + **TreeMap** 
    - Elements are sorted by Key
      * Key Objects must have a compareTo Method
    - Dynamic
      * The size of the TreeMap can be modified
    - Both Keys and Values must be Objects
      * Can be different types of Objects
    - All Keys must be unique
    - Associated Values can be duplicated
      * Multiple Keys can be linked to the same Values
        + Example: 2 Students with the same address
* Processing All Elements
  + Because some of these Classes do not use indices (Maps, Sets) we need another way to loop through all the elements
    - We use a **For Each Loop**
      * Cycles through all elements in the collection by assigning them to a temporary variable
      * Set Example
        + for (elementType name: collectionName) {use name here}
      * Map Example
        + for (String key: mapName.keySet()) {}

Get the key then use the get(key) method to get the value

* Use a Wrapper Class to convert the primitive data type to use it in the Collection Classes
  + This converts the primitive value into an Object