Basic Data Structures

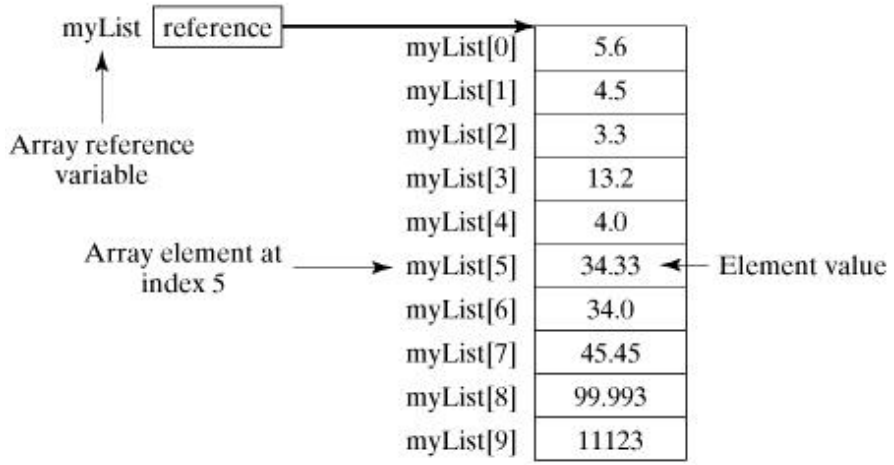
Java Programming for Absolute Beginners Phase I

What are Data Structures?

- In programming, often it is necessary to have a collection of data, rather than a bunch of distinct variables with their own names.
- For example, think of a deck of cards, there is no real reason to make a separate variable for each card, so a data structure can be used.
- Data structures have different behaviors and rules to them, as they are each made to serve different purposes.
- We will be looking at two data structures today, but first we need to talk about arrays.

Arrays

- An **array** is a collection of data of the *same type*, and has a *fixed length*.
  - Ex1: A grade book of 20 test scores
  - Ex2: A roster of 43 names
  - Ex3: A deck of 52 cards
- Each individual piece of data in the array is called an **element** and each element is accessed using an **index**, which is an integer from 0 – N-1 where N is the length of the array.
- Yes, we start counting the indices of the array at 0, so the first element has index 0, the second element has index 1, and so on.
- Arrays are objects, so an array variable is really a reference to the array object in memory.



Initializing an Array

- An array is an object, so the **new** keyword is going to be used in its initialization.
- Arrays are noted by square brackets ([]) after the type of the array.
  - Declaration: <data_type>[] <variable_name>;
- Arrays are always initialized to have a certain length.
- The following statements are equivalent:
  - int[] data = new int[35];
  - int data[] = new int[35];
  - int[] data; data = new int[35];
- Each of these creates an integer array of length 35, and assigns the reference to the data variable.
- Remember that once the array is created, it cannot be resized, so in order to have such an effect, the a new array must be created, and the reference needs to be reassigned.
- Arrays are filled with the default value of their type, i.e. 0 for numeric types, null for references, etc.

Initializing List

- Consider the following code:

```
1. int[] coins = new int[4];
2. coins[0] = 1;
3. coins[1] = 5;
4. coins[2] = 10;
5. coins[3] = 25;
```

- This array has a predetermined collection of values, so there is a shortcut. This line is equivalent to the 5 above:

```
1. int[] coins = {1, 5, 10, 25};
```

Length of Arrays

- Arrays are objects, so they can have more data.
- One piece of data all arrays have is a public final (constant) int called length.
- This instance variable is read-only, and represents the number of elements in the array.
- Accessing the length of an array:
  1. int n = myArr.length;
- **Note:**
  - Strings can be thought of as an array of characters, and in the backend, that *is* what they are.
  - However, the String class does not have an *is-a* relationship with an array, so it does not have the length variable.
  - It does have an accessor method for the length of the string:
    - int numCharacters = myString.length();

Traversing Arrays: For Loop

- There are two main ways to traverse an array, one is using the for loop we learned earlier, and the other is using an enhanced for loop (*for-each*).
- Here is the basic structure of traversing an array using a for loop:

```
1. <data_type>[] arr = new <data_type>[<length>];
2. <statements>
3. for (int i = 0; i < arr.length; i++) {
4.     <statements>;
5. }
```

- This works because the indices of the array are 0, 1, … , length – 1.
- Use this if you need to know the index of each individual element, i.e, if you need to change the value, or if you do not need to traverse the entire array.

Traversing Arrays: Enhanced For Loop

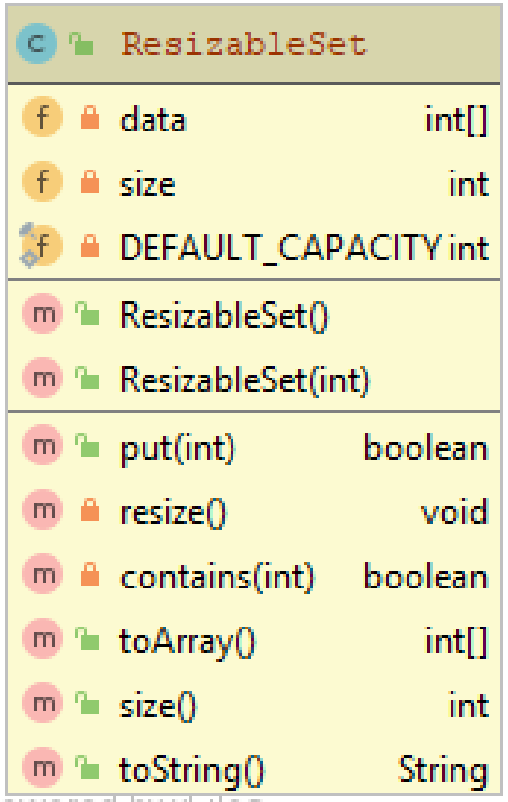
- If you only need to access the data in an array, and need to traverse the entire array, then an enhanced for loop can be used.
- An enhanced for loop is also called a **for each** loop.
- Here is the structure for a for each loop:

```
1. for (<array_type> <identifier> : <array_reference) {
2.         <statements>
3. }
```

- What this does, is traverses the array in order, assigning each individual value to the variable declared on the LHS of the ':' at each iteration.

Arrays as Parameters

- Note that array variables are memory references, so when passing an array to a method, no copy of the array is made, only a copy of reference.
- This means that the parameter also references the original array, which allows the array to be modified within the method body.
- This is *incredibly* useful in programming.



Example 7 UML

What About Data Structures?

- Today was supposed to be about data structures, why did we spend so much time looking at arrays?
- It's because all the ideas we have talked about transfer over when talking about different data structures. Especially the first one we will be talking about today.
- Data Structures are defined by their **Abstract Data Type** (ADT) which defines the behavior and features of a given data structure. An ADT has nothing to do with programming per se.
- In terms of programming, you can think of an ADT as an interface for a data structure.

List

- A list is a data structure with ordered elements. Duplicates are allowed, and elements are accessed using an integer index ranging from 0 to N – 1, where N is the number of elements in the list. (Sound Familiar?) The element of index 0 is the first element, and the element of index N – 1 is the last element.
- Items can be added to the end of the list, or inserted into a specific location in the list.
- Elements at given locations can be replaced with new values.
- Elements can be removed from the list.

A Quick Note on Generics

- Java allows for generic types, this is a can of worms, but basically what it means is that you can write a class that has variables of a type that will be determined when an instance is created.
- I know that sounds weird, let's look at a simple example.
- But first, the important rule about generics is that they must be Objects, so primitive data is not allowed.
- We can use wrapper classes instead.
- Read more: https://docs.oracle.com/javase/tutorial/java/generics/types.html

The List<E> Interface

- List<E> is an interface that represents the List ADT. Here are the useful methods in the List<E> Interface:
  - boolean add(E obj) : Adds the given element to the end of the list. Always returns true.
  - int size() : Returns the number of elements in the list.
  - E get(int index) : Returns the element at the given index in the list.
  - E set(int index, E element) : Replaces the element at the given index with the given element, returns the original element.
  - void add(int index, E element) : Inserts an element at the given index, the original elements with at index and higher are shifted over, so their indices increase by one, and size is increased by one.
  - E remove(int index) : Removes the element at the given index and returns it. All the elements after the given index are shifted over, so their indices decrease by one and size is decreased by one.
  - Iterator<E> iterator() : Returns an iterator over the list in order starting at the beginning.
- Full API: https://docs.oracle.com/javase/8/docs/api/java/util/List.html

The ArrayList<E> Class

- ArrayList<E> is an implementation of the List ADT using an array in the backend. It works almost exactly how our resizable set worked, only it does not check for duplicates.
- ArrayList<E> implements List<E> so an ArrayList object *is-a* List object.
- Keep in mind ArrayList is not the only implementation of the List ADT in Java, for example, there is also the LinkedList.
- The different implementations have different technical specifications, but since we defined the data structure in an abstract sense, the different implementations behave identically.
- If you want to learn more about the different types of implementations of different data structures, and when to use each kind, go to college!

Dictionary

- The Dictionary data structure (aka Map or Associative List) is a bit different. It works through key-value pairs. Where a key is an object that is used to access a value.
- Think of the key as the index of an element, but instead of using an integer, any object value is used.
- There may be duplicate values, but no duplicate keys.
- The implementation of this is *complex* but we will not be going over that.
- Elements are in no particular order.

The Map<K, V> Interface

- Map<K, V> is an interface that represents the Dictionary ADT in Java. K is the type of the keys, and V is the type of the values.
- Here are the methods:
  - V put(K key, V value) : Associates the given key with the given value. If an association for that key already exists, then the association will be overwritten and the original value is returned. Otherwise, the association is created and null is returned.
  - V get(K key) : Returns the value associated with the given key, or null if there is no mapping for the given key.
  - V remove(Object key) : Removes the association of the given key and returns the value originally associated. If the given key has no association, null is returned.

The HashMap<K, V> Class

- The HashMap<K, V> class is the implementation of Map<K, V> that we will be using.
- The internal working of a HashMap is *very* complicated, go to college to learn that.
- The order of the elements is defined by some computer science, but just keep in mind that you should not rely on the elements being in order when traversing the map.
- There are implementations where the keys are always sorted, HashMap is not such an implementation.

Any Questions?