BASIC DATA STRUCTURES

CS062

DATA STRUCTURES AND ADVANCED PROGRAMMING

7: ArrayLists

Alexandra Papoutsaki
she/her/hers

We are officially done with learning the basics of Java and we are starting a new chapter in the course, where we will talk about the most basic data structures.

Lecture 7: ArrayLists

- ArrayList
- Java Collections

Some slides adopted from Algorithms 4th Edition and Oracle tutorials

We will start with the first fundamental data structure, the array list.

Limitations of arrays

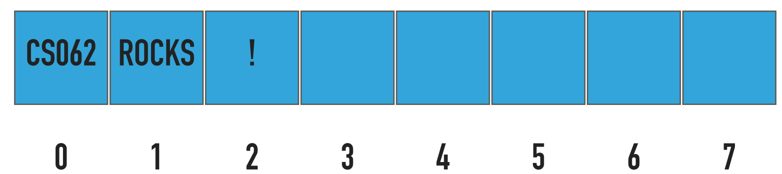
- Fixed-size.
- Do not work well with generics.
  - `E[] myArray = (E[]) new Object[capacity];`
- Limited functionality (Java requires the use of `Arrays` class for printing contents and manipulating arrays, such as sorting and searching).

- We want resizable arrays that support any type of object.

So far, we've talked about arrays and how limited they are. They are fixed in size which is not flexible. They don't work well with generics: we would need to create an array of objects and cast it into an array of the formal type parameter. Additionally, it's hard to do thinks like sort them, search through them, or event print their contents; for that we would need to pass our array to the Arrays class. Our aim is to work with resizable arrays that support any type of object.

ArrayList (or dynamic/growable/resizable/mutable array)

- Dynamic linear data structure that is zero-indexed.
  - We will use the `List` interface.
- Sequential data structure that requires consecutive memory cells.
- Implemented with an underlying array of a specific capacity.
  - But the user does not see that!



This is what array lists (also known as dynamic/growable/resizable/mutable arrays) provide. They are dynamic linear data structures that are zero-indexed which means they can implement the List interface we defined last time. In terms of implementation, their data is stored in consecutive memory cells and we achieve that using an underlying array of a specific capacity but we hide this behavior from the user.

Reminder: Interface List

```
public interface List <E> {
    void add(E element);
    void add(int index, E element);
    void clear();
    E get(int index);
    boolean isEmpty();
    E remove();
    E remove(int index);
    E set(int index, E element);
    int size();
}
```

Let’s refresh our memory about the List interface we defined in the last lecture. If we implement it, we promise to implement the methods:

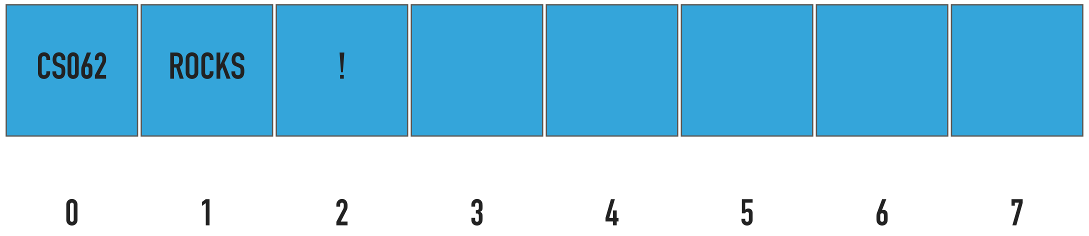
void add(E element);
void add(int index, E element);
void clear();
E get(int index);
boolean isEmpty();
E remove();
E remove(int index);
E set(int index, E element);

Standard Operations of `ArrayList<E>` class

- `ArrayList()`: Constructs an empty ArrayList with an initial capacity of 2 (can vary across implementations, another common initial capacity is 10).
- `ArrayList(int capacity)`: Constructs an empty ArrayList with the specified initial capacity.
- `isEmpty()`: Returns true if the ArrayList contains no elements.
- `size()`: Returns the number of elements in the ArrayList.
- `get(int index)`: Returns the element at the specified index.
- `add(E element)`: Appends the element to the end of the ArrayList.
- `add(int index, E element)`: Inserts the element at the specified index and shifts the element currently at that position (if any) and any subsequent elements to the right (adds one to their indices).
- `E remove()`: Removes and returns the element at the end of the ArrayList.
- `E remove(int index)`: Removes and returns the element at the specified index. Shifts any subsequent elements to the left (subtracts one from their indices).
- E `set(int index, E element)`: Replaces the element at the specified index with the specified element and returns the olde element.
- `clear()`: Removes all elements.

In fact we won't only do that, we will also support a few more things. We will also have two constructors, one that sets the initial capacity of the array list to 2 (other common implementations set it to 10) and one that the user chooses the initial capacity. When it comes to add, we will append an element to the end of the arratlist. Same thing from remove.

ArrayLists



Capacity = 8

Size = 3

This is what an array list could look like. This particular one has a capacity of 8 and there are three elements in it, from index 0 to 2.

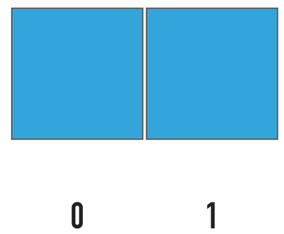
`ArrayList()`: Constructs an ArrayList

What should happen?

```
ArrayList<String> al = new ArrayList<String>();
```

Given what we’ve discussed so far, what would happen if we typed ArrayList<String> al = new ArrayList<String>();?

`ArrayList()`: Constructs an ArrayList




```
ArrayList<String> al = new ArrayList<String>();
```

Capacity = 2

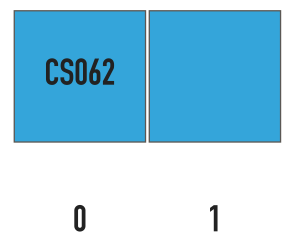
Size = 0

What should happen?

```
al.add("CS062");
```

An array list of capacity 2 with 0 elements would be created. What if we type al.add("CS062");?

`add(E element)`: Appends the element to the end of the ArrayList



`al.add("CS062");`

Capacity = 2

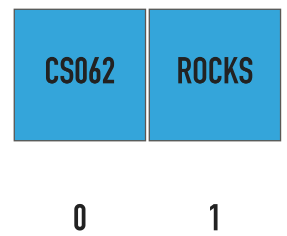
Size = 1

What should happen?

`al.add("ROCKS");`

"CS062" is appended to the end of the array list, that is at index 0 and the size grows by 1. What about al.add("ROCKS");?

`add(E element)`: Appends the element to the end of the ArrayList



`al.add("ROCKS");`

Capacity = 2

Size = 2

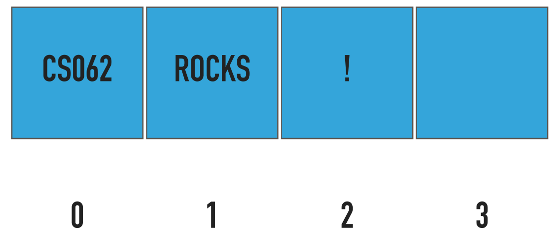
What should happen?

`al.add("!");`

"ROCKS" is appended to the end of the array list, that is at index 0 and the size grows by 1. What about al.add("ROCKS");?

`add(E element)`: Appends the element to the end of the ArrayList

- DOUBLE CAPACITY SINCE IT'S FULL AND THEN ADD NEW ELEMENT



`al.add("!");`

Capacity = 4

Size = 3

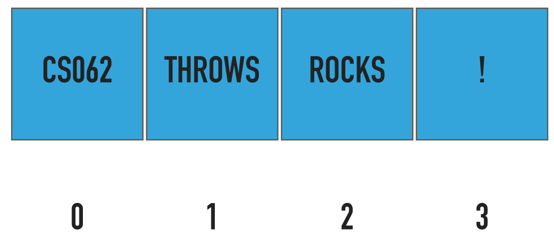
What should happen?

`al.add(1, "THROWS");`

Since the size is equal to the capacity, we will double the capacity, append ! To the 3rd cel and increase size by 1. What about al.add(1, "THROWS");?

`add(int index, E element)`: Adds element at the specified index

- SHIFT ELEMENTS TO THE RIGHT




```
al.add(1, "THROWS");
```

Capacity = 4

Size = 4

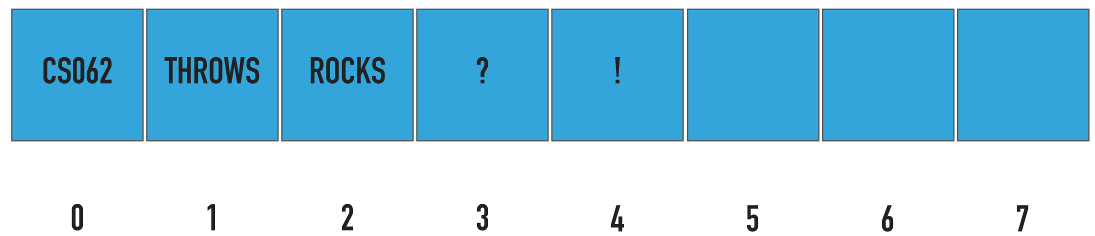
What should happen?

```
al.add(3, "?");
```

We are going to shift the elements at index 1 and to the right to make space for the new element, add it, and increase size by 1. What about al.add(3, “?”);?

`add(int index, E element)`: Adds element at the specified index

- DOUBLE CAPACITY SINCE IT'S FULL
- SHIFT ELEMENTS TO THE RIGHT



Capacity = 8

Size = 5

```
al.add(3, “?”);
```

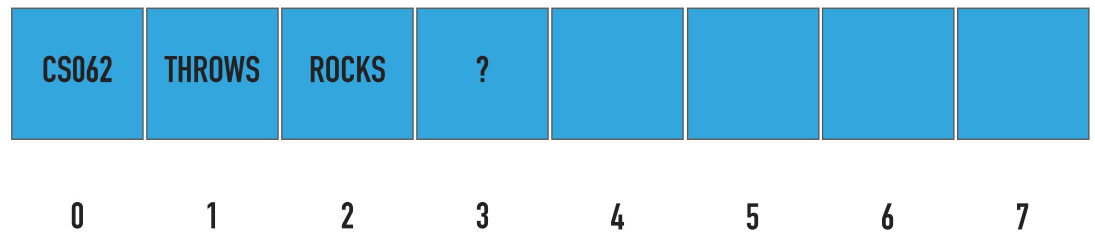
What should happen?

```
al.remove();
```

Since size and capacity are the same, we will double the capacity, shift elements from index 3 and to the right, add the new element, and increase size by 1. What about al.remove();?

`remove()`: Removes and returns element from the end of ArrayList

- REMOVE AND RETURN LAST ELEMENT



Capacity = 8

Size = 4

```
al.remove();
```

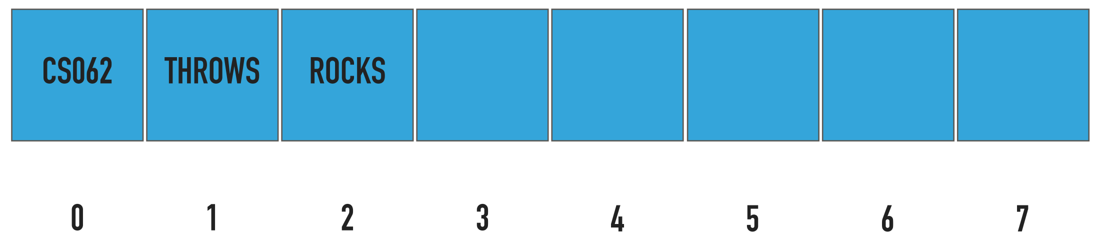
What should happen?

```
al.remove();
```

We will remove and return the element from the end of the array list and reduce the size by 1. What should happen if we type al.remove();?

`remove()`: Removes and returns element from the end of ArrayList

- REMOVE AND RETURN LAST ELEMENT



Capacity = 8

Size = 3

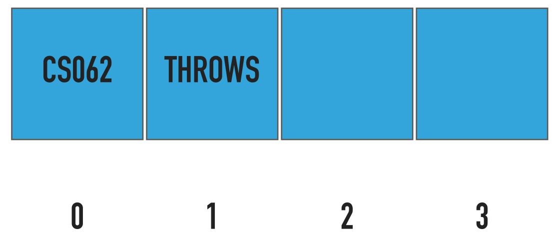
`al.remove();`

What should happen?

`al.remove();`

Same thing. We will remove and return the element from the end of the array list and reduce the size by 1. What should happen if we type again al.remove();?

`remove()`: Removes and returns element from the end of ArrayList




```
al.remove();
```

Capacity = 4

Size = 2

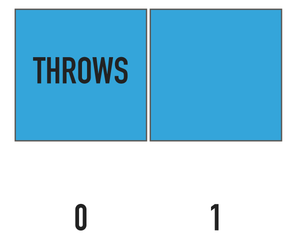
- **REMOVE AND RETURN ELEMENT FROM THE END**
- **HALVE CAPACITY WHEN 1/4 FULL**

What should happen?

```
al.remove(0);
```

Same thing. We will remove and return the element from the end of the array list and reduce the size by 1. But now the size is 1/4 of the capacity which means that we have a lot of unused space. To be conservative with our space usage, we will halve the capacity when the array is 1/4 full.What should happen if we type al.remove(0);?

`remove(int index)`: Removes and returns element from specified index




```
al.remove(0);
```

- REMOVE ELEMENT FROM INDEX
- SHIFT ELEMENTS TO THE LEFT
- HALVE CAPACITY WHEN 1/4 FULL

Capacity = 2

Size = 1

We will remove and return the element from index 0, shift elements to the left, and reduce the size by 1. But again the size is 1/4 of the capacity which means that we have a lot of unused space.

Our own implementation of ArrayLists

- We will follow the recommended textbook style.
  - It does not offer a class for this so we will build our own. We got to test Java's built-in implementation in lab!
- We will work with generics because we want arrayLists to hold objects of an type.
- We will implement the `List` interface we defined in the last lecture.
- We will use an array and we will keep track of how many elements we have in our ArrayList.

Now we understand the mechanics of how array lists should work let's see how we could implement an ArrayList class (you already have used the java.util.ArrayList class that is built-in Java. We will work with generics, we will implement the list interface, and we will use an underlying array to reserve consecutive memory and store our data and will keep track of how many elements we have in our data structure.

Instance variables and constructors

```
public class ArrayList<E> implements List<E> {
    private E[] data; // underlying array of elements
    private int size; // number of elements in ArrayList

    /**
     * Constructs an ArrayList with an initial capacity of 2.
     */
    @SuppressWarnings("unchecked")
    public ArrayList() {
        data = (E[]) new Object[2];
        size = 0;
    }

    /**
     * Constructs an ArrayList with the specified capacity.
     */
    @SuppressWarnings("unchecked")
    public ArrayList(int capacity) {
        data = (E[]) new Object[capacity];
        size = 0;
    }
```

This is captured in this class declaration. You see we have two instance variables, an array of type E and a variable that captures the number of elements in the array list. The two overloaded constructors show how we reserve an array of 2 or the specified capacity elements and set the size at 0.

PRACTICE TIME: Check if is empty and how many elements

```
/**
 * Returns true if the ArrayList contains no elements.
 *
 * @return true if the ArrayList does not contain any element
 */
public boolean isEmpty() {

}

/**
 * Returns the number of elements in the ArrayList.
 *
 * @return the number of elements in the ArrayList
 */
public int size() {

}
```

Think about how you would implement the isEmpty and size methods.

Check if is empty and how many elements

```java
/**
 * Returns true if the ArrayList contains no elements.
 *
 * @return true if the ArrayList does not contain any element
 */
public boolean isEmpty() {
    return size == 0;
}

/**
 * Returns the number of elements in the ArrayList.
 *
 * @return the number of elements in the ArrayList
 */
public int size() {
    return size;
}
```

Did you get something like this?

PRACTICE TIME: Resize underlying array's capacity

```
/**
 * Resizes the ArrayList's capacity to the specified capacity.
 */
@SuppressWarnings("unchecked")
private void resize(int capacity) {
    //reserve a new temporary array with the provided capacity

    //copy all the elements from the old array (data) into the temporary array


    //point data to the new temporary array




}
```

We will also have a helper method (this is why it's private) that resizes the underlying array. It does so by reserving a new temporary array at the new provided capacity, copying to it the old array (i.e. the contents of the array data), and then makes data point to the temporary array.

Resize underlying array's capacity

```java
/**
 * Resizes the ArrayList's capacity to the specified capacity.
 */
@SuppressWarnings("unchecked")
private void resize(int capacity) {
    //reserve a new temporary array of Es with the provided capacity
    E[] temp = (E[]) new Object[capacity];
    //copy all the elements from the old array (data) into the temporary array
    for (int i = 0; i < size; i++){
        temp[i] = data[i];
    }
    //point data to the new temporary array
    data = temp;
}
```

This is how that would look like

PRACTICE TIME: Append an element to the end of ArrayList

```
/**
 * Appends the element to the end of the ArrayList. Doubles its capacity if necessary.
 *
 * @param element the element to be inserted
 */
public void add(E element) {
    //check whether ArrayList is full

    //if yes, double in size


    //add the element at the end of the ArrayList and increase the counter by 1


}
```

Now let’s think about how we would implement the add method to append an element to the end. Remember, if our array list is full, we need to resize it.

Append an element to the end of ArrayList

```
/**
 * Appends the element to the end of the ArrayList. Doubles its capacity if necessary.
 *
 * @param element the element to be inserted
 */
public void add(E element) {
     //check whether ArrayList is full
    if (size == data.length){
     //if yes, double in size
        resize(2 * data.length);
    }
     //add the element at the end of the ArrayList and increase the counter by 1
    data[size] = element;
    size++;
}
```

Did you get something like that?

PRACTICE TIME: Add an element at a specified index

```java
/**
 * Inserts the element at the specified index. Shifts existing elements
 * to the right and doubles its capacity if necessary.
 *
 * @param index
 *            the index to insert the element
 * @param element
 *            the element to be inserted
 * @pre 0<=index<=size
 */
public void add(int index, E element) {
    //check whether index in range

    //if full double size

    //shift elements to the right


    //set element to position index

    //increase number of elements

}
```

What about adding an element to a specific index. Let's first make sure that the index is in range. Throw an IndexOutOfBoundsException otherwise.

Add an element at a specified index

```
/**
 * Inserts the element at the specified index. Shifts existing elements
 * to the right and doubles its capacity if necessary.
 *
 * @param index
 *            the index to insert the element
 * @param element
 *            the element to be inserted
 * @pre 0<=index<=size
 */
public void add(int index, E element) {
    //check whether index in range
    if (index > size || index < 0){
           throw new IndexOutOfBoundsException("Index " + index + " out of bounds");
     }

    //if full double size
    if (size == data.length){
           resize(2 * data.length);
    }

    //shift elements to the right
    for (int i = size; i > index; i--){
           data[i] = data[i - 1];
    }
    size++;

    //set element to position index
    data[index] = element;
}
```

Did your implementation look like this?

PRACTICE TIME: Replace an element at a specified index

```
/**
 * Replaces the element at the specified index with the specified element.
 * @param index
 *            the index of the element to replace
 * @param element
 *            element to be stored at specified index
 * @return the old element that was changed.
 * @pre 0<=index<size
*/
public E set(int index, E element) {
    //check whether index in range

    //retrieve old element at index

    //update index with new element

    //return old element

}
```

The set method should replace an element at the specified index and return the old element at that position.

Replace an element at a specified index

```
/**
 * Replaces the element at the specified index with the specified element.
 * @param index
 *            the index of the element to replace
 * @param element
 *            element to be stored at specified index
 * @return the old element that was changed.
 * @pre 0<=index<size
 */
public E set(int index, E element) {
    //check whether index in range
     if (index >= size || index < 0){
        throw new IndexOutOfBoundsException("Index " + index + " out of bounds");
      }
    //retrieve old element at index
    E old = data[index];
    //update index with new element
    data[index] = element;
    //return old element
    return old;
}
```

Did your implementation look like this?

PRACTICE TIME: Retrieve and remove element from the end of ArrayList

```
/**
 * Removes and returns the element from the end of the ArrayList.
 * @return the removed element
 * @throws NoSuchElementException if ArrayList is empty
 * @pre size>0
 */
public E remove() {
    //if ArrayList is empty throw NoSuchElementException


    //retrieve last element after you reduce number of elements by 1

    //set the position where the removed element is to null


    //shrink in half to save space if number of elements in ArrayList is 1/4 of its capacity


    //return the removed element

}
```

The remove method should remove and return the element from the end of the array list. Don’t forget we want to shrink it to half its capacity if it’s 1/4 occupied.

Retrieve and remove element from the end of ArrayList

```java
/**
 * Removes and returns the element from the end of the ArrayList.
 * @return the removed element
 * @throws NoSuchElementException if ArrayList is empty
 * @pre size>0
 */
public E remove() {
    //if ArrayList is empty throw NoSuchElementException
    if (isEmpty()){
        throw new NoSuchElementException("The list is empty");
    }
    //retrieve last element after you reduce number of elements by 1
    size--;
    E element = data[size];
    //set the position where the removed element is to null
    data[size] = null; // Avoid loitering (see text).

    //shrink in half to save space if number of elements in ArrayList is 1/4 of its capacity
    if (size > 0 && size == data.length / 4)
        resize(data.length / 2);
    //return the removed element
    return element;
}
```

You will notice that the last element is set to null. Java's garbage collection policy is to reclaim the memory associated with any objects that can no longer be accessed. In our remove() implementations, the reference to the removed element remains in the array. When the client relinquishes its last reference to the removed element, the element is effectively an orphan - it cannot be accessed but the Java garbage collector has no way to know this until the array entry is overwritten. This condition (holding a reference to an element that is no longer needed) is known as loitering. It is easy to avoid here by setting the array entry corresponding to the removed element to null.

PRACTICE TIME: Retrieve and remove element from a specific index

```java
/**
 * Removes and returns the element at the specified index.
 *
 * @param index
 *            the index of the element to be removed
 * @return the removed element
 * @pre 0<=index<size
 */
public E remove(int index) {
    //check whether index in range

    //retrieve element at index

    //reduce number of elements by 1

    //shift all elements from index till the end one position to the left


    //set the last element (since they have been shifted to the left), to null

    //shrink in half to save space if number of elements in ArrayList is 1/4 of its capacity

    //return removed element

}
```

The idea behind the remove an element at a specified index is similar.

Retrieve and remove element from a specific index

```
/**
 * Removes and returns the element at the specified index.
 *
 * @param index
 *            the index of the element to be removed
 * @return the removed element
 * @pre 0<=index<size
 */
public E remove(int index) {
    //check whether index in range
    if (index >= size || index < 0){
           throw new IndexOutOfBoundsException("Index " + index + " out of bounds");
    }
    //retrieve element at index
      E element = data[index];
    //reduce number of elements by 1
      size--;
     //shift all elements from index till the end one position to the left
      for (int i = index; i < size; i++){
           data[i] = data[i + 1];
     }
    //set the last element (since they have been shifted to the left), to null
      data[size] = null; // Avoid loitering (see text).

    //shrink in half to save space if number of elements in ArrayList is 1/4 of its capacity
      if (size > 0 && size == data.length / 4){
           resize(data.length / 2);
     }
    //return removed element
      return element;
}
```

Did you get something similar?

PRACTICE TIME: Clear all elements

```
/**
 * Clears the ArrayList of all elements.
 */
public void clear() {

    // Go through all elements of the array and set them to null


    // Set number of elements to 0

}
```

How about clearing an arraylist? Note that there is no need to call remove(), we can just set all elements to null and avoid unnecessary computation.

Clear all elements

```java
/**
 * Clears the ArrayList of all elements.
 */
public void clear() {

    // Go through all elements of the array and set them to null
    for (int i = 0; i < size; i++){
        data[i] = null;
    }
    // Set number of elements to 0
    size = 0;
}
```

Straightforward right?

Lecture 7: ArrayLists

- ArrayList
- Java Collections

What we saw so far is our own implementation of the abstract data type array list. Java as we've already seen supports array lists in the util package. In fact, Java supports a lot of common data structures through the Java Collections framework.

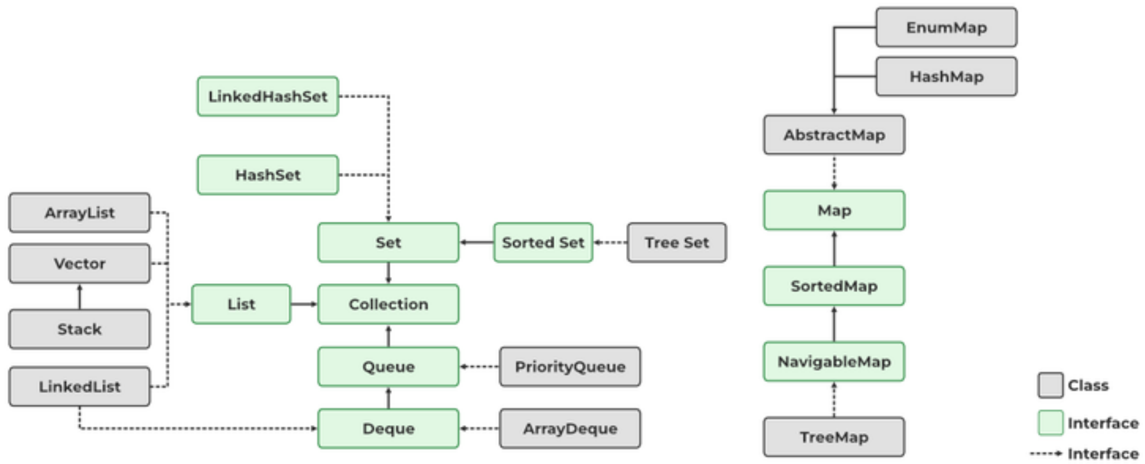
The Java Collections Framework

- Collection: an object that groups multiple elements into a single unit, allowing us to store, retrieve, manipulate data.
- Collections Framework:
  - Interfaces: ADTs (abstract data types) that represent collections.
  - Implementations: The actual data structures.
  - Algorithms: methods that perform useful operations, such as searching and sorting.

https://docs.oracle.com/javase/tutorial/collections/intro/index.html

A collection is an object that groups multiple elements into a single unit, allowing us to store, retrieve, manipulate data. The Collections framework provides interfaces which represent abstract data types (ADTs), their implementations which are the actual data structures, and algorithms that is methods that perform useful operations, such as searching and sorting.

The Java Collections Framework



https://www.geeksforgeeks.org/collections-in-java-2/

This is what the Collections framework looks like in terms of dependencies. You might wonder what an abstract class is. You can see that the ArrayList class implements a List interface as well!

List Interface

- A collection storing elements in an ordered fashion.
- Elements are accessed in a zero-based fashion.
- Typically allow duplicate elements and null values but always check the specifications of implementation.

https://en.wikipedia.org/wiki/Java_collections_framework

The List interface dictates that any data structure that implements it will store its elements in an ordered fashion and that elements are accessed in a zero-based fashion. Typically Java allows duplicate elements and null values but always check the specifications of implementation.

`ArrayList` in Java Collections

- Resizable list that increases by 50% when full and does NOT shrink.
- Not thread-safe (more in CS105).

```
java.util.ArrayList;

public class ArrayList<E> extends AbstractList<E>
implements List<E>
```

https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html

The java.util.ArrayList class is a resizable list that increases by 50% when full and in contrast to our implementation does NOT shrink. It is not thread-safe which means multiple threads can work on ArrayList at the same time. For e.g. if one thread is performing an add operation on ArrayList, there can be an another thread performing remove operation on ArrayList at the same time in a multithreaded environment and that can get generous.

Vector in Java Collections

- Java has one more class for resizable arrays.
- Doubles when full.
- Is synchronized (more in CS105).

```
java.util.Vector;

public class Vector<E> extends AbstractList<E>
implements List<E>
```

https://docs.oracle.com/javase/8/docs/api/java/util/Vector.html

If you work with multiple threads, the class Vector is preferable. Vectors are resizable arrays that double when full. They are synchronized. This means if one thread is working on Vector, no other thread can get a hold of it. Unlike ArrayList, only one thread can perform an operation on vector at a time. Since it is slower, unless you work with multiple threads, ArrayList is recommended.

Lecture 7: ArrayLists

- ArrayList
- Java Collections

And that's all for today, next time we will talk about how fast the different array list operations are.

Readings:

- Oracle's guides:
  - Collections: https://docs.oracle.com/javase/tutorial/collections/intro/index.html
  - ArrayLists: https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html
- Recommended Textbook:
  - Chapter 1.3 (Page 136-137)
- Recommended Textbook Website:
  - Resizable arrays: https://algs4.cs.princeton.edu/13stacks/

Code

- Lecture 7 code