01/04/23

Reviewing Arrays

Arrays

- Arrays are collections of values of the same type

```
type[] name;
```

- Examples

```
boolean[] answers;
String[] questions;
int[] scores;
Student[] students;
```

Arrays

- The size of an Array (i.e. the number of values it contains) is established during initialization and can not be changed (without re-initialization)
- Use the Array property `length` to determine the size of an Array

`boolean[] answers = {true, false, false, true};`	`answers.length == 4`
`int[] scores = {100, 84, 78};`	`scores.length == 3`
`double[] prices = new double[20];`	`prices.length == 20`
`String[] questions = new String[5];`	`questions.length == 5`

`int numStudents = 10;`
`Student[] students = new Student[numStudents];`
	`students.length == 10`

Re-Sizing Arrays

- Since the size of an Array is established during initialization - it can be challenging to use them for collections of data that are unknown in advance - or that are highly variable
- Examples
    - The students who attended a basketball game
    - The advertisements that appear while watching a video
    - The items in an online shopping cart

Re-Sizing Arrays

- So we end up writing code like this to resize Arrays (via re-initialization) as the size of the data collection needs to grow

```
int[] scores = new int[0];

void addNewScore(int newScore) {
  int[] newScoresArray = new int[scores.length + 1];
  for (int idx = 0; idx < scores.length; idx++ ) {
    newScoresArray[idx] = scores[idx];
  }
  newScoresArray[newScoresArray.length - 1] = newScore;
  scores = newScoresArray;
}
```

Re-Sizing Arrays

- ...or more concisely with the `Arrays.copyOf` helper

```
import java.util.Arrays;

int[] scores = new int[0];

void addNewScore(int newScore) {
  scores = Arrays.copyOf(scores, scores.length + 1);
  scores[scores.length - 1] = newScore;
}
```

Re-Sizing Arrays

- ...or more concisely with the `Arrays.copyOf` helper

```
import java.util.Arrays;

int[] scores = new int[0];

void addNewScore(int newScore) {
  scores = Arrays.copyOf(scores, sco
  scores[scores.length - 1] = newSco
}
```

But what if the size of Arrays could grow automatically as the collection increased in size?

7.1: `ArrayList`

ArrayList

- ArrayLists are collections of values of the same Object type; But have different declaration syntax than Arrays; **Primitive types (int, boolean, double, etc.) are not supported**

```
ArrayList<type> name;
```

- Examples

~~ArrayList<boolean> answers;~~ ** PRIMITIVE TYPES UNSUPPORTED **

```
ArrayList<Boolean> answers;
```

~~ArrayList<int> scores;~~ ** PRIMITIVE TYPES UNSUPPORTED **

```
ArrayList<Integer> scores;
ArrayList<String> questions;
ArrayList<Student> students;
```

- **Important:** You must import ArrayList prior to using it

```
import java.util.ArrayList;
```

Generics / Generic Types

- `ArrayList` is an example of function that uses a Generic Type

```
ArrayList<type> name;
```

- Generic Types are an option when the **same code** can be used across a variety of data types - and frees you from needing to create an overloaded function for every type
- `ArrayList` is able to use Generic Types because the internals assume everything is a `Object` type (and all `Object` types share the functionality required for `ArrayList` to work)
- You can read more about Generics in the online Java documentation
  - Oracle Java Documentation: Why Use Generics?

ArrayList

- Like Arrays, you must initialize `ArrayLists` prior to using them; The most common usage is with the no-parameter Constructor

```
ArrayList<Boolean> answers = new ArrayList<Boolean>();
ArrayList<Integer> scores = new ArrayList<Integer>();
ArrayList<String> questions = new ArrayList<String>();
ArrayList<Student> students = new ArrayList<Student>();
```

- **Note:** There are two other `ArrayList` Constructors that you can explore on your own

```
ArrayList<type> name = new ArrayList<type>(Collection<type> c);
ArrayList<type> name = new ArrayList<type>(init initialCapacity);
```

`ArrayList`

- Unlike Arrays, `ArrayLists` automatically manage their memory usage as you `ArrayList.add()` and `ArrayList.remove()` elements to/from the the `ArrayList`
- Unlike Arrays, `ArrayLists` do not have a `length` property that indicates the fixed-size of the Array; They have the `ArrayList.size()` method that indicates the current number of elements included in the `ArrayList`
- `ArrayLists` have an internal capacity - which you cannot access - that grows and shrinks as needed to ensure elements can be quickly added. **The default capacity is 10.**
- The capacity is adjusted to ensure that the there is enough free space to quickly accommodate new items via `ArrayList.add()`; But not so much excess free space that available memory is wasted

Abstraction & Encapsulation

- `ArrayLists` are a good example of both Abstraction and Encapsulation - two of the principal concepts in Object-Oriented Programming that we briefly covered in Section 5
- `ArrayLists` contain an Array that is inaccessible to code outside the class (Encapsulation) - and provides a set of functions that simplifies common operations used on Arrays (Abstraction)
- The core Java Language uses Array in its operations, but the Java authors providing pre-build classes like `ArrayList` that demonstrate how new classes can be created to create new (or simplified) functionality

Array vs ArrayList

Array

true	false	true	*false*	*false*	*false*	*false*	*false*	*false*	*false*
0	1	2	3	4	5	6	7	8	9

```
boolean[] answers = new boolean[10];
answers[0] = true; answers[1] = false; answers[2] = true;
answers.length == 10
answers[3-9] are set to default values
```

ArrayList

true	false	true							
0	1	2	3	4	5	6	7	8	9

```
ArrayList<Boolean> answers = new ArrayList<Boolean>();
answers.add(true); answers.add(false); answers.add(true);
answers.size() == 3
answers[3-9] are unused pre-allocated capacity
```

ArrayList

- When using the `ArrayList` no-parameter Constructor; the `ArrayList` has an internal capacity of 10; but no values are assigned; so `ArrayList.size()` returns `0`

```
ArrayList<Boolean> answers = new ArrayList<Boolean>();
> answers.size() == 0

ArrayList<String> questions = new ArrayList<String>();
> questions.size() == 0

ArrayList<Student> students;
> students.size() ** ERROR ** - students has not been initialized
```

`ArrayList` Methods

`ArrayList` Methods

- **`add()`**
- `clear()`
- `get()`
- `isEmpty()`
- `remove()`
- `removeRange()`
- `set()`
- `size()`

`ArrayList` `index` values are zero-based (just like Arrays)

Signatures

- `boolean add(E obj)`
- `void add(int index, E obj)`

Overview

- Add an item either to the end of the `ArrayList` (**always returns `true`**) or at the specified `index` (existing items will shift right; their index values will increase by 1)
  - The first version of **`add()`** always returns `true` because `ArrayList` implements the `Collection` interface - which **can** be implemented by other classes to restrict the creation of duplicate or `null` elements (`ArrayList` has no such restrictions)
- Automatically increases the `ArrayList` capacity as needed
- Will throw IndexOutOfBoundsException if `index` is out of range `(index < 0 || index > size())`

ArrayList Methods

- add()
- **clear()**
- get()
- isEmpty()
- remove()
- removeRange()
- set()
- size()

Signatures

- `void clear()`

Overview

- Removes all elements from the `ArrayList`
- After this call `ArrayList.size() == 0`
- Automatically decreases the `ArrayList` capacity as needed

`ArrayList` Methods

- `add()`
- `clear()`
- **`get()`**
- `isEmpty()`
- `remove()`
- `removeRange()`
- `set()`
- `size()`

`ArrayList` `index` values are zero-based (just like Arrays)

Signatures

- **`E`** **`get`**`(`**`int`** `index)`

Overview

- Returns the element at the specified position in the `ArrayList`
- You must use this method to access the items in an `ArrayList`; `ArrayList` does not support the `[]` syntax of Arrays
- Will throw IndexOutOfBoundsException if `index` is out of range `(index < 0 || index >= size())`

`ArrayList` Methods

- `add()`
- `clear()`
- `get()`
- **`isEmpty()`**
- `remove()`
- `removeRange()`
- `set()`
- `size()`

Signatures

- **`boolean isEmpty`**`()`

Overview

- Returns true if the `ArrayList` has no items

ArrayList Methods

- add()
- clear()
- get()
- isEmpty()
- **remove()**
- removeRange()
- set()
- size()

Signatures

- **boolean remove**(Object obj)
- **E remove**(**int** index)

Overview

- Removes the first item from the ArrayList that matches obj; or at the specified index (existing items will shift left; their index values will decrease by 1)
  - **remove(**Object obj**)** returns true/false if an element in the ArrayList returns true for obj.equals(element) (or obj == null == element) and was removed
    - Note: Does **not use** Object equality (obj == element)
  - **remove(int** index**)** returns the element that was removed from the ArrayList
- Automatically decreases the ArrayList capacity as needed
- Will throw IndexOutOfBoundsException if index is out of range (index < 0 || index >= size())

ArrayList index values are zero-based (just like Arrays)

ArrayList Methods

- add()
- clear()
- get()
- isEmpty()
- **remove()**
- removeRange()
- set()
- size()

**** CAUTION ****

If your ArrayList is collecting Integers be sure to pass an int if you want to remove by index and an Integer if you want to remove by value!

Signatures

- `boolean remove(Object obj)`
- `E remove(int index)`

```
ArrayList<Integer> values = new ArrayList<Integer>()

values.add(0); values.add(1);
values.add(2); values.add(3);
/* values == [0, 1, 2, 3] */

values.remove(1);
/* values == [0, 2, 3] */

Integer iValue = 2;
values.remove(iValue);
/* values == [0, 3] */
```

`ArrayList` Methods

- `add()`
- `clear()`
- `get()`
- `isEmpty()`
- `remove()`
- **`removeRange()`**
- `set()`
- `size()`

`ArrayList index` values are zero-based (just like Arrays)

Signatures

- `void removeRange(int fromIndex, int toIndex)`

Overview

- Removes all of the elements whose index is between `fromIndex` (inclusive) and `toIndex` (exclusive). Shifts any succeeding elements to the left (reduces their index).
- Automatically decreases the `ArrayList` capacity as needed
- Will throw IndexOutOfBoundsException if `fromIndex` or `toIndex` is out of range `(fromIndex < 0 || fromIndex >= size() || toIndex > size() || toIndex < fromIndex)`

`ArrayList` Methods

- `add()`
- `clear()`
- `get()`
- `isEmpty()`
- `remove()`
- `removeRange()`
- **`set()`**
- `size()`

`ArrayList` `index` values are zero-based (just like Arrays)

Signatures

- `E set(int index, E element)`

Overview

- Replaces the element at the specified position in this `ArrayList` with the specified element.
- Returns the element that was removed from the `ArrayList` at `index`
- You must use this method to access the items in an `ArrayList`; `ArrayList` does not support the `[]` syntax of Arrays
- Will throw IndexOutOfBoundsException if `index` is out of range `(index < 0 || index >= size())`

`ArrayList` Methods

- `add()`
- `clear()`
- `get()`
- `isEmpty()`
- `remove()`
- `removeRange()`
- `set()`
- **`size()`**

Signatures

- `int size()`

Overview

- Returns the number of elements in this `ArrayList`

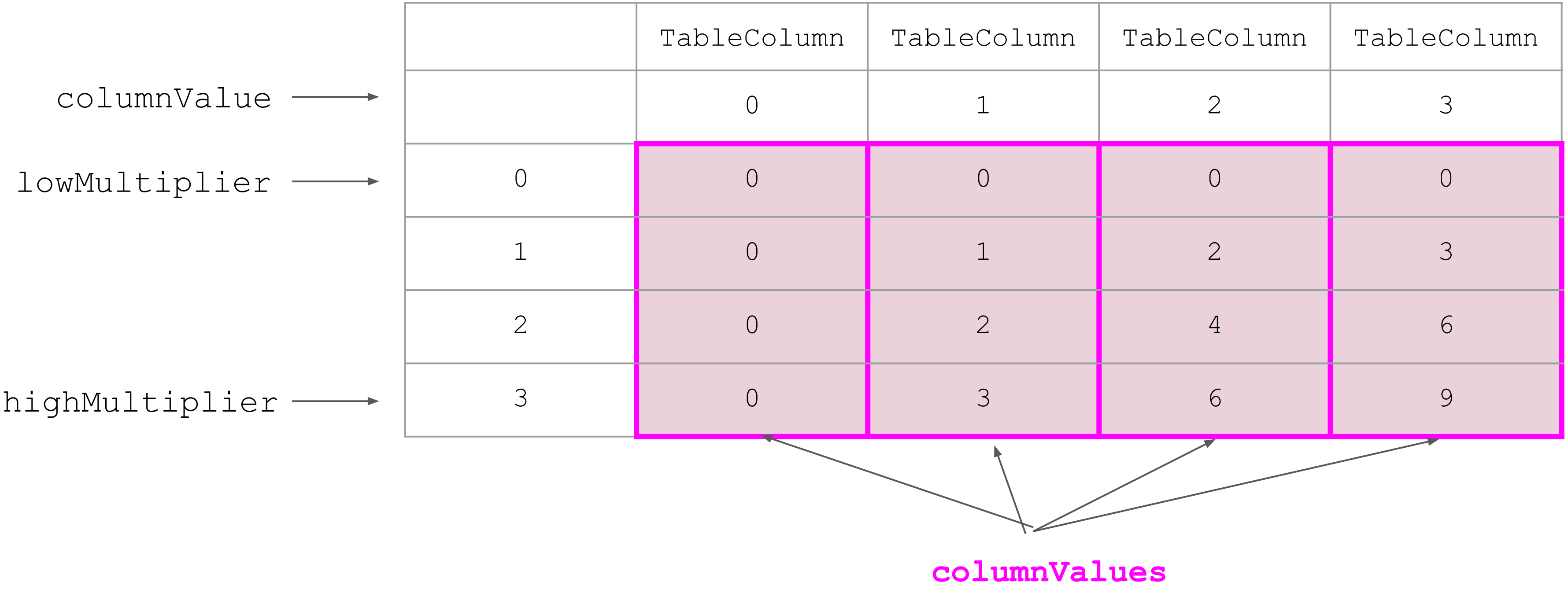
`ArrayList` Methods (Not Discussed)

- **Iteration (TBD Friday)**
  - `forEach()`
- **Operations**
  - `addAll()`
  - `clone()`
  - `removeAll()`
  - `removeIf()`
  - `replaceAll()`
  - `retainAll()`
  - `sort()`
  - `subList()`
  - `toArray()`
- **Memory**
  - `ensureCapacity()`
  - `trimToSize()`
- **Discovery**
  - `contains`
  - `indexOf`
  - `lastIndexOf`

Check out *Java Documentation: ArrayList Reference* **for the complete information about `ArrayList` methods and properties**

Practice on your own

- CSAwesome 7.1 - Intro to ArrayLists
- CSAwesome 7.2 - ArrayList Methods
- Replit - Multiplication Tables
    - We are going to use the same Replit today and ~~Friday~~ Monday
    - For today's exercise follow the instructions in Main.java and complete the code required to enable the `MainWed.run()` code path
    - On ~~Friday~~ Monday we will do a quick overview of traversing `ArrayLists` with loops and then spend the remaining time on the `MainFri.run()` code path



	TableColumn	TableColumn	TableColumn	TableColumn
	0	1	2	3
0	0	0	0	0
1	0	1	2	3
2	0	2	4	6
3	0	3	6	9