

# Section 1

CSCI E-22

Will Begin Shortly

## The ArrayBag Class

In lecture, we implemented `ArrayBag`, a simple collection class. A **bag** is a collection with no guaranteed ordering, no restriction on duplicate values, and no restriction on the types of values inside the bag. Here's an overview:

```
public class ArrayBag {
    private Object[] items;
    private int numItems;
    ...
    public boolean add(Object item)      { ... }
    public boolean remove(Object item)   { ... }
    public boolean contains(Object item) { ... }
    public int numItems()                 { ... }
    public Object grab()                  { ... }
    public Object[] toArray()             { ... }
}
```

Using `Object[]` allows the `items` array to contain references to values of any type (since all classes extend `Object`).

There is no `getItem(int index)` method for accessing a particular item in the bag. Why not?

## The toArray() Method

toArray() is another way to access the items in an ArrayBag. It creates **a copy** of the items array and returns it. The length of the copy is equal to numItems.

```
public ArrayBag {
    private Object[] items;
    private int numItems;
    ...
    public Object[] toArray() {
        Object[] copy = new Object[this.numItems];

        for (int i = 0; i < this.numItems; i++) {
            copy[i] = this.items[i];
        }

        return copy;
    }
    ...
}
```

## The add() Method

In lecture, we discussed the add() method. The method returns true if an item could be added or false if the items array is not large enough to store another item.

```
public ArrayBag {
    ...
    public boolean add(Object item) {
        if (item == null) {
            throw new IllegalArgumentException(...);
        } else if (this.numItems == this.items.length) {
            return false;
        } else {
            this.items[this.numItems] = item;
            this.numItems++;
            return true;
        }
    }
    ...
}
```

## The contains() Method

Now let's look at the `contains()` method, which returns `true` if the item can be found in the bag, and `false` otherwise.

```
public ArrayBag {  
    ...  
    public boolean contains(Object item) {  
        for (int i = 0; i < this.numItems; i++) {  
            if (this.items[i].equals(item)) {  
                return true;  
            }  
        }  
  
        return false;  
    }  
    ...  
}
```

We use the `equals()` method, rather than `==`, because we're comparing references

Why is the loop bound by `this.numItems` and not `this.items.length`?

Why is it safe to return `true` inside the loop?

## The contains() Method

What's wrong with this version of `contains()`?

```
public ArrayBag {  
    ...  
    public boolean contains(Object item) {  
        for (int i = 0; i < this.numItems; i++) {  
            if (this.items[i].equals(item)) {  
                return true;  
            } else {  
                return false;  
            }  
        }  
        ...  
    }  
    ...  
}
```

Why can't we return `false` inside the loop?

## The ArrayBag Methods

Let's now write the `ArrayBag` implementation of the `remove()` method:

## The ArrayBag Methods

Now let's look at the `containsAll()` method:

```
public boolean containsAll(Bag otherBag) {
    if (otherBag == null || otherBag.numItems() == 0) {
        return false;
    }

    Object[] otherItems = otherBag.toArray();
    for (int i = 0; i < otherItems.length; i++) {
        if (!this.contains(otherItems[i])) {
            return false;
        }
    }

    return true;
}
```

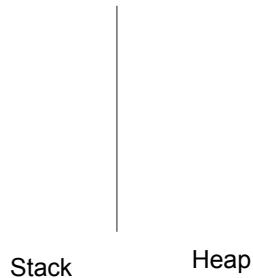
Why are we able to avoid using the `numItems()` and `toArray()` accessor methods, opting instead to use the private `numItems` and `items` fields?

## Copying Objects

Recall that variables that represent objects or arrays actually store a *reference* to the object or array—i.e., the memory address of the object or array on the heap.

What would things look like in memory after the following lines are executed?

```
ArrayBag b1 = new ArrayBag(5);  
b1.add("hello");  
b1.add("world");
```



## Copying Objects

If you want to create a copy of an object, you **can't** just do a simple assignment like the following:

```
ArrayBag b2 = b1;
```

What would our memory diagram look like after this assignment?

## Copying Objects

To create a true copy, we need to create a new object and make it equivalent to the original object.

Let's write an `ArrayBag` constructor that does this in VS Code!

```
public ArrayBag(ArrayBag other) {
```

Our goal: add everything from `ArrayBag other` to the `ArrayBag` that we're constructing

The client that called this constructor could look something like:

```
ArrayBag b1 = new ArrayBag(5);  
b1.add("hello");  
b1.add("world");
```

```
Bag b2 = new ArrayBag(b1);
```

```
}
```

