Section 1

CSCI S-22

Will Begin Shortly

Section meetings

- **Goal: reinforce concepts from lecture and prepare you for problem sets**
- Look at concrete examples, write code together, prepare for midterm & final
- Attendance is optional but encouraged due to the intensity of this course
- One section meeting per lecture (two section meetings per week)
- Blank slides and solutions on the course website: https://cscie22.sites.fas.harvard.edu/sections/index.html
- Zoom recordings also available on Gather under *Published Recordings*: https://dcegather.canvas.harvard.edu/courses/157496

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?

An item in a bag doesn't have a position, so you can't request an item at a given index.

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`?

If the bag is not full, there are `null` elements at the end of the array. We don't need to consider these elements.

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

As soon as we see one occurrence of the item, we can stop looping.

If we were counting the number of occurrences, we would **not** be able to return inside the loop, only **after** the loop is done.

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?

This method returns *during the first iteration of the loop*. It only considers one element: `items[0]`.

What if the item appears later in the array?

The `ArrayBag` Methods

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

```
public ??????? remove(?????? ????) {









}
```

What should the return type and parameter type be?

The `ArrayBag` Methods

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

```
public boolean remove(?????? ????) {









}
```

What should the return type and parameter type be?

The `ArrayBag` Methods

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

```
public boolean remove(Object item) {









}
```

What should the return type and parameter type be?

We need to access every element of the `items` array to compare it to the parameter `item`. How?

The `ArrayBag` Methods

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

```
public boolean remove(Object item) {
    for (int i = 0; i < ????????; i++) {









    }



}
```

What should the return type and parameter type be?

We need to access every element of the `items` array to compare it to the parameter `item`. How?

What should the `for` loop bound be?

The `ArrayBag` Methods

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

```
public boolean remove(Object item) {
    for (int i = 0; i < numItems; i++) {









    }



}
```

What should the return type and parameter type be?

We need to access every element of the `items` array to compare it to the parameter `item`. How?

What should the `for` loop bound be?

How do we compare one element from the array to the parameter?

The `ArrayBag` Methods

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

```
public boolean remove(Object item) {
    for (int i = 0; i < numItems; i++) {

        if (items[i].equals(item)) {









        }

    }


}
```

What should the return type and parameter type be?

We need to access every element of the `items` array to compare it to the parameter `item`. How?

What should the `for` loop bound be?

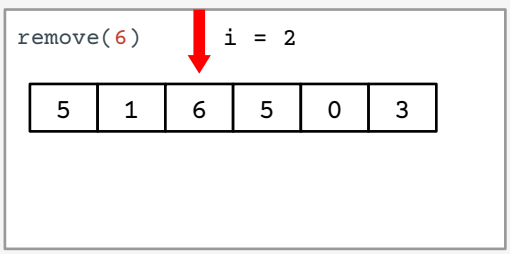
How do we compare one element from the array to the parameter?

What should we do if `items[i]` equals the parameter?

The `ArrayBag` Methods

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

```
public boolean remove(Object item) {
    for (int i = 0; i < numItems; i++) {

        if (items[i].equals(item)) {









        }

    }


}
```



What should the return type and parameter type be?

We need to access every element of the `items` array to compare it to the parameter `item`. How?

What should the `for` loop bound be?

How do we compare one element from the array to the parameter?

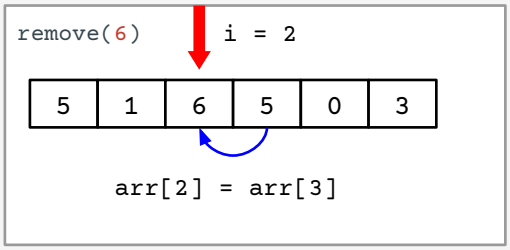
What should we do if `items[i]` equals the parameter?

How do we shift the rest of the elements left, to eliminate the element at index `i`?

The `ArrayBag` Methods

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

```
public boolean remove(Object item) {
    for (int i = 0; i < numItems; i++) {

        if (items[i].equals(item)) {









        }

    }

}
```



What should the return type and parameter type be?

We need to access every element of the `items` array to compare it to the parameter `item`. How?

What should the `for` loop bound be?

How do we compare one element from the array to the parameter?

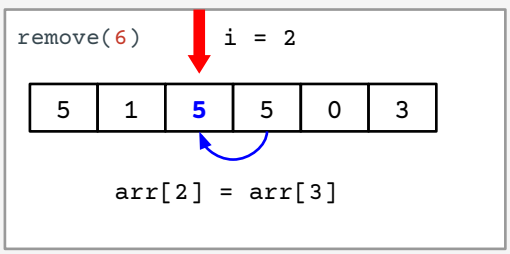
What should we do if `items[i]` equals the parameter?

How do we shift the rest of the elements left, to eliminate the element at index `i`?

The `ArrayBag` Methods

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

```
public boolean remove(Object item) {
    for (int i = 0; i < numItems; i++) {

        if (items[i].equals(item)) {









        }

    }

}
```

What should the return type and parameter type be?

We need to access every element of the `items` array to compare it to the parameter `item`. How?

What should the `for` loop bound be?

How do we compare one element from the array to the parameter?

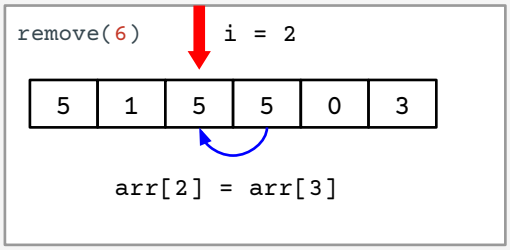
What should we do if `items[i]` equals the parameter?

How do we shift the rest of the elements left, to eliminate the element at index `i`?

The `ArrayBag` Methods

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

```
public boolean remove(Object item) {
    for (int i = 0; i < numItems; i++) {

        if (items[i].equals(item)) {









        }

    }

}
```



What should the return type and parameter type be?

We need to access every element of the `items` array to compare it to the parameter `item`. How?

What should the `for` loop bound be?

How do we compare one element from the array to the parameter?

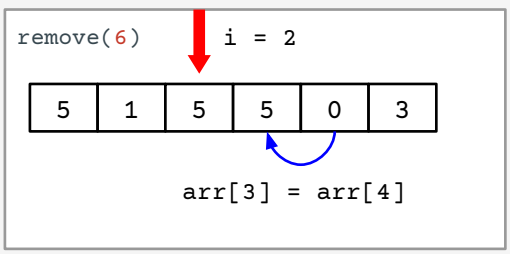
What should we do if `items[i]` equals the parameter?

How do we shift the rest of the elements left, to eliminate the element at index `i`?

The `ArrayBag` Methods

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

```
public boolean remove(Object item) {
    for (int i = 0; i < numItems; i++) {

        if (items[i].equals(item)) {









        }

    }

}
```

What should the return type and parameter type be?

We need to access every element of the `items` array to compare it to the parameter `item`. How?

What should the `for` loop bound be?

How do we compare one element from the array to the parameter?

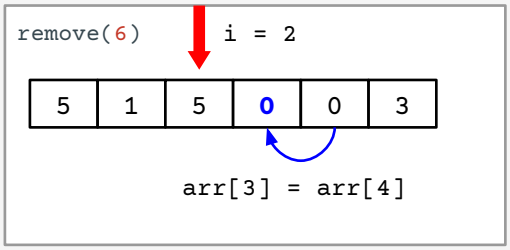
What should we do if `items[i]` equals the parameter?

How do we shift the rest of the elements left, to eliminate the element at index `i`?

The `ArrayBag` Methods

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

```
public boolean remove(Object item) {
    for (int i = 0; i < numItems; i++) {

        if (items[i].equals(item)) {









        }

    }

}
```



What should the return type and parameter type be?

We need to access every element of the `items` array to compare it to the parameter `item`. How?

What should the `for` loop bound be?

How do we compare one element from the array to the parameter?

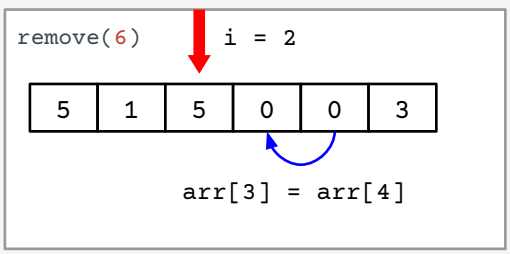
What should we do if `items[i]` equals the parameter?

How do we shift the rest of the elements left, to eliminate the element at index `i`?

The `ArrayBag` Methods

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

```
public boolean remove(Object item) {
    for (int i = 0; i < numItems; i++) {

        if (items[i].equals(item)) {









        }

    }

}
```



What should the return type and parameter type be?

We need to access every element of the `items` array to compare it to the parameter `item`. How?

What should the `for` loop bound be?

How do we compare one element from the array to the parameter?

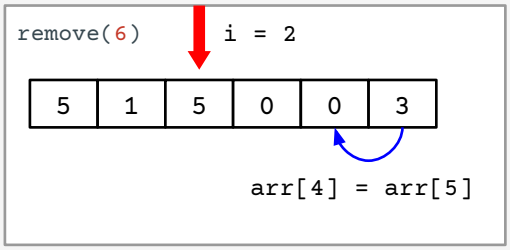
What should we do if `items[i]` equals the parameter?

How do we shift the rest of the elements left, to eliminate the element at index `i`?

The `ArrayBag` Methods

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

```
public boolean remove(Object item) {
    for (int i = 0; i < numItems; i++) {

        if (items[i].equals(item)) {









        }

    }

}
```



What should the return type and parameter type be?

We need to access every element of the `items` array to compare it to the parameter `item`. How?

What should the `for` loop bound be?

How do we compare one element from the array to the parameter?

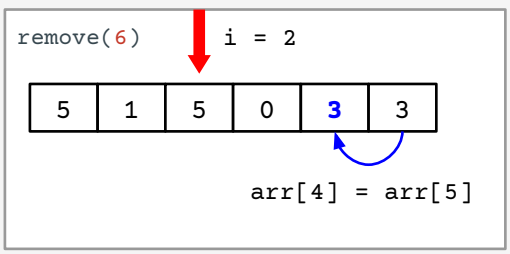
What should we do if `items[i]` equals the parameter?

How do we shift the rest of the elements left, to eliminate the element at index `i`?

The `ArrayBag` Methods

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

```
public boolean remove(Object item) {
    for (int i = 0; i < numItems; i++) {

        if (items[i].equals(item)) {









        }

    }

}
```

What should the return type and parameter type be?

We need to access every element of the `items` array to compare it to the parameter `item`. How?

What should the `for` loop bound be?

How do we compare one element from the array to the parameter?

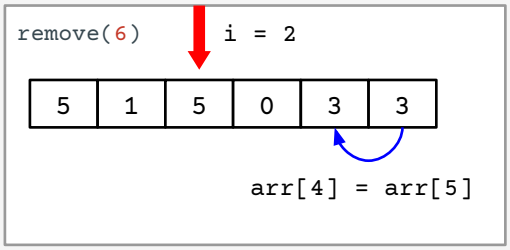
What should we do if `items[i]` equals the parameter?

How do we shift the rest of the elements left, to eliminate the element at index `i`?

The `ArrayBag` Methods

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

```
public boolean remove(Object item) {
    for (int i = 0; i < numItems; i++) {

        if (items[i].equals(item)) {









        }

    }

}
```



What should the return type and parameter type be?

We need to access every element of the `items` array to compare it to the parameter `item`. How?

What should the `for` loop bound be?

How do we compare one element from the array to the parameter?

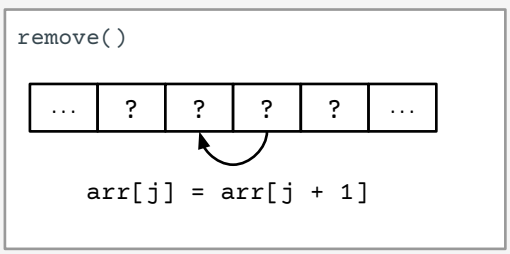
What should we do if `items[i]` equals the parameter?

How do we shift the rest of the elements left, to eliminate the element at index `i`?

The `ArrayBag` Methods

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

```
public boolean remove(Object item) {
    for (int i = 0; i < numItems; i++) {

        if (items[i].equals(item)) {
            for (int j = i; j < numItems - 1; j++) {
                items[j] = items[j + 1];
            }









        }

    }

}
```



A loop variable `j` starting out at `i`

Shift each element left until the index `numItems - 1` is reached

The `ArrayBag` Methods

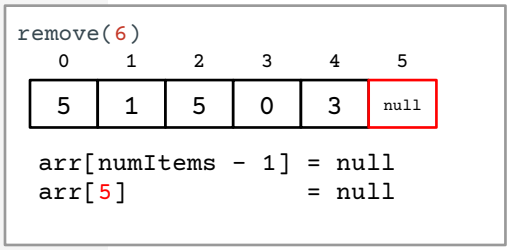
Let's now write the `ArrayBag` implementation of the `remove()` method:

```
public boolean remove(Object item) {
    for (int i = 0; i < numItems; i++) {

        if (items[i].equals(item)) {
            for (int j = i; j < numItems - 1; j++) {
                items[j] = items[j + 1];
            }

            items[numItems - 1] = null;


        }

    }


}
```

A loop variable `j` starting out at `i`

Shift each element left until the index `numItems - 1` is reached

Finally, set `numItems - 1` to `null` (it has a copy of the last element)



The `ArrayBag` Methods

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

```
public boolean remove(Object item) {
    for (int i = 0; i < numItems; i++) {

        if (items[i].equals(item)) {
            for (int j = i; j < numItems - 1; j++) {
                items[j] = items[j + 1];
            }

            items[numItems - 1] = null;
            numItems--;

        }

    }


}
```

A loop variable `j` starting out at `i`

Shift each element left until the index `numItems - 1` is reached

Finally, set `numItems - 1` to `null` (it has a copy of the last element)

Decrement `numItems` since the array has one more free element

The `ArrayBag` Methods

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

```
public boolean remove(Object item) {
    for (int i = 0; i < numItems; i++) {

        if (items[i].equals(item)) {
            for (int j = i; j < numItems - 1; j++) {
                items[j] = items[j + 1];
            }

            items[numItems - 1] = null;
            numItems--;
            return true;
        }

    }

    return false;
}
```

A loop variable `j` starting out at `i`

Shift each element left until the index `numItems - 1` is reached

Finally, set `numItems - 1` to `null` (it has a copy of the last element)

Decrement `numItems` since the array has one more free element

Finally, return `true` when an item was removed and `false` when nothing was removed

The `ArrayBag` Methods

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

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

    for (int i = 0; i < otherItems.numItems; i++) {
        if (!this.contains(otherBag.items[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?

`containsAll()` is a non-static method inside the `ArrayBag` class, so it can access the private fields of an `ArrayBag`.

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.

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?

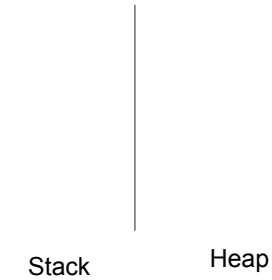
```
ArrayBag b1 = new ArrayBag(5);
b1.add("hello");
b1.add("world");
```

Copying Objects

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");
```

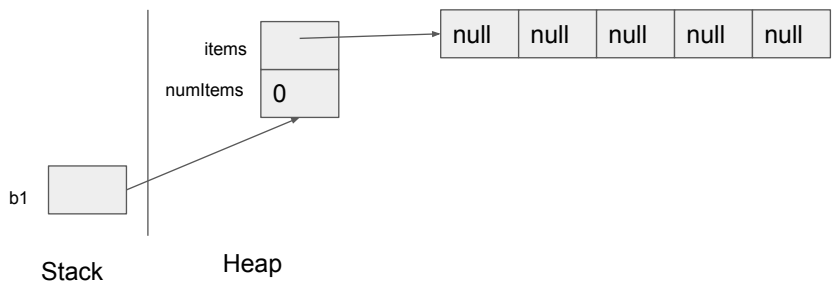


Copying Objects

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");
```

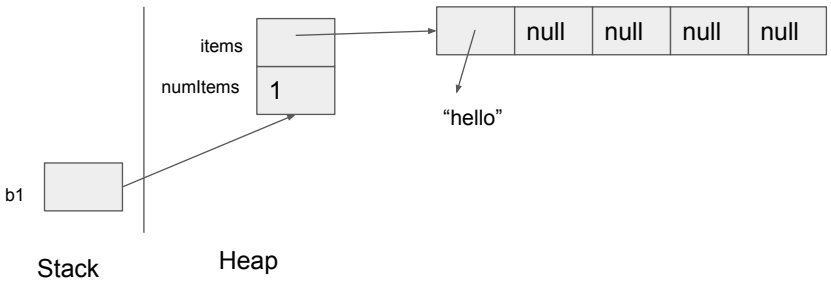


Copying Objects

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");
```

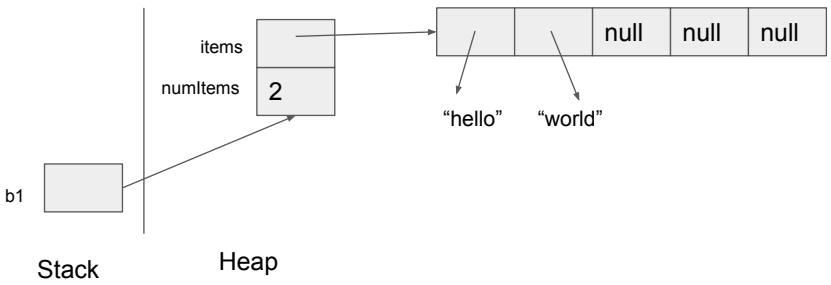


Copying Objects

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;
```

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;
```

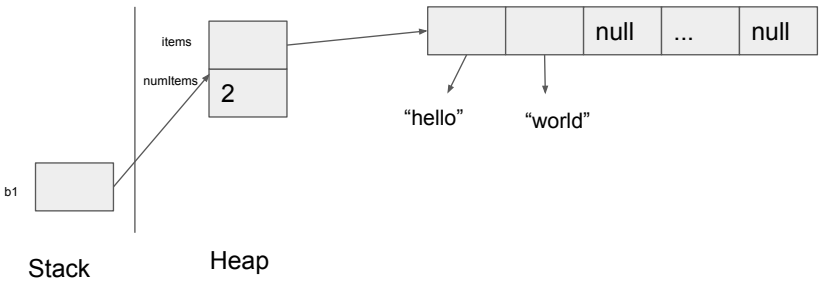
What would our memory diagram look like after this assignment?

Copying Objects

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?

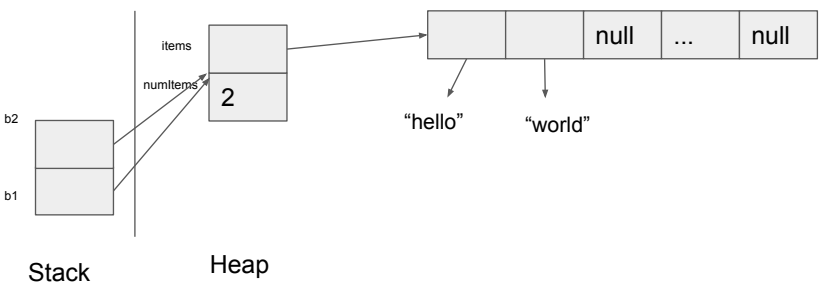


Copying Objects

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?



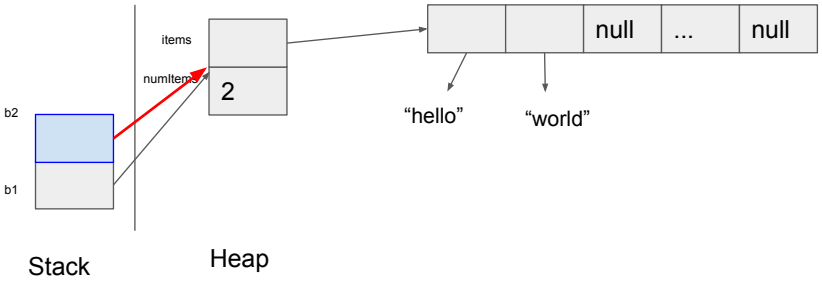
Copying Objects

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?

All we've done is copy over the reference inside of b1!



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.

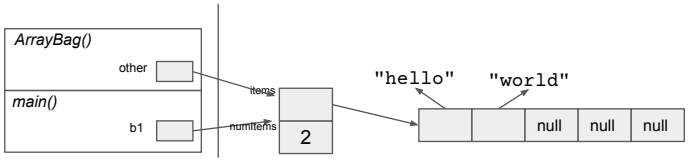
```
public class ArrayBag {
    private Object[] items;
    private int numItems;
    ...
    public ArrayBag(ArrayBag other) {
        // ???
    }
    ...
}
```

Objective: add all the items from the other bag to the `ArrayBag` we're constructing.

The client that called this constructor could look something like this:

```
ArrayBag b1 = new ArrayBag(5);
b1.add("hello");
b1.add("world");
ArrayBag b2 = new ArrayBag(b1);
```

Let's draw a diagram for this code.



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.

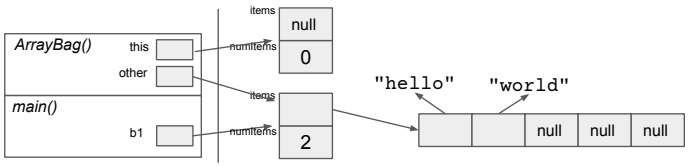
```
public class ArrayBag {
    private Object[] items;
    private int numItems;
    ...
    public ArrayBag(ArrayBag other) {
        // ???
    }
    ...
}
```

Objective: add all the items from the other bag to the `ArrayBag` we're constructing.

The client that called this constructor could look something like this:

```
ArrayBag b1 = new ArrayBag(5);
b1.add("hello");
b1.add("world");
ArrayBag b2 = new ArrayBag(b1);
```

Let's draw a diagram for this code.



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.

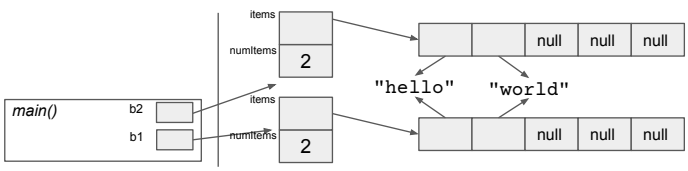
```
public class ArrayBag {
    private Object[] items;
    private int numItems;
    ...
    public ArrayBag(ArrayBag other) {
        // ???
    }
    ...
}
```

Objective: add all the items from the other bag to the `ArrayBag` we're constructing.

The client that called this constructor could look something like this:

```
ArrayBag b1 = new ArrayBag(5);
b1.add("hello");
b1.add("world");
ArrayBag b2 = new ArrayBag(b1);
```

Let's draw a diagram for this code.



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.

```
public class ArrayBag {
    ...
    public ArrayBag(ArrayBag other) {
        if (other == null) {
            throw new IllegalArgumentException(...);
        }









    }
    ...
}
```

If there is no other bag, throw an exception

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.


```
public class ArrayBag {
    ...
    public ArrayBag(ArrayBag other) {
        if (other == null) {
            throw new IllegalArgumentException(...);
        }

        this.items = new Object[???];









    }
    ...
}
```


If there is no other bag, throw an exception

Before we allocate memory for the array, we must specify its length, which cannot be changed

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.

```
public class ArrayBag {
    ...
    public ArrayBag(ArrayBag other) {
        if (other == null) {
            throw new IllegalArgumentException(...);
        }

        this.items = new Object[other.items.length];









    }
    ...
}
```

If there is no other bag, throw an exception

Before we allocate memory for the array, we must specify its length, which cannot be changed

One approach: use the length of the other bag's `items` array

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.

```
public class ArrayBag {
    ...
    public ArrayBag(ArrayBag other) {
        if (other == null) {
            throw new IllegalArgumentException(...);
        }

        this.items = new Object[other.items.length];

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

    }
    ...
}
```

If there is no other bag, throw an exception

Before we allocate memory for the array, we must specify its length, which cannot be changed

One approach: use the length of the other bag's `items` array

Our loop is bound by `other.numItems`

The default value of the `numItems` field is 0, and the `add()` method increments it for us