The `ArrayBag` class

Examine the fields and constructors of the `ArrayBag` class.

- `items` is of type `Object[]` which allows us to store references to any type of object
- the default constructor creates an underlying array of size 50
- the `ArrayBag(int maxSize)` constructor takes a value specifying the size

The `add()` method

The `add()` method takes an object and returns `true` if the item was added to the bag.

```
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;
    }
}
```

- When does it return `false`?
  - When there is no more room (`numItems == items.length`)

The `contains()` method

The `contains()` method returns `true` if the specified object exists in the array, and `false` otherwise.

```
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 `equals()` instead of == when comparing references
- The loop is bound by `numItems`, not `items.length`

Returning inside the loop

What if we modified `contains()` so it returns `false` inside the loop?

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

- Would this work?
  - **No!** The loop only runs once. The method will only find the item if it's at index 0.

The `remove()` method

Now write an implementation of `remove()`.

Avoid leaving `null`s in the middle of the array. To remove an item, shift all the items to the left by one position. Suppose we start with this array:

```
┌──────┬────┬─────┬───────┬──────┬──────┬─────┐
│ "hi" │ 32 │ '@' │ false │ null │ null │ ... │
└──────┴────┴─────┴───────┴──────┴──────┴─────┘
```

After removing 32, the array should look like this:

```
┌──────┬─────┬───────┬──────┬──────┬─────┐
│ "hi" │ '@' │ false │ null │ null │ ... │
└──────┴─────┴───────┴──────┴──────┴─────┘
```

Test-driven development

Starting from correct, compiling code:

1. Add a failing test case that represents a desired feature
2. Write the code that makes the test case pass
3. Refactor as needed (if you introduce a bug by accident, the test case will start failing)
4. Go to step 1 for another feature

Use this technique to help you implement `remove()`.

Before implementing, write some code that creates an `ArrayBag` with some items, calls `remove()`, and verifies that the item was correctly removed.

The containsAll() method

The containsAll() method takes another ArrayBag and returns true if all the items in the other bag are present in this bag.

```
public boolean containsAll(ArrayBag otherBag) {
    if (otherBag == null || otherBag.numItems == 0) {
        return false;
    }
    for (int i = 0; i < otherBag.numItems; i++) {
        if (!this.contains(otherBag.items[i])) {
            return false;
        }
    }
    return true;
}
```

- Note that this method can access the private fields of the other ArrayBag (this is only allowed inside the ArrayBag.java file)

Making a copy (first attempt)

What is happening in memory after these lines of code are executed?

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

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

Read the code carefully and come up with a guess. Then open this Java Tutor session and step through the code line by line.

- What are the contents of b1? How about b2?
  - There is only one bag; b1 and b2 contain the same reference (memory address)
  - The contents are {"hello", "world"}

Making a copy (correctly)

To get a new bag, we need to allocate memory on the heap. This requires the use of the `new` keyword, which calls a constructor.

Write a constructor for the `ArrayBag` class that accepts another `ArrayBag` and copies its references into a new bag's array.

- To allow for more items to be added, make the new array twice the size of the other bag's array

Bonus: to-do list app

Build a to-do list app that uses two bags to keep track of list items: one for unfinished items, and another for finished items.

The app must offer these features:

- Add a to-do list item
- Mark a to-do list item as done
- Delete a to-do list item

Think of how you could use two bags to implement these features. Try to use test-driven development (TDD) to build each feature without needing to write code in `main()` that interacts with a user.