Preliminaries

- course account e-mail: cscie22@fas.harvard.edu. Use for general questions.
- office hours

ArrayBag Class

In lecture, we used an interface to specify the operations supported by the Bag abstract data type:

```java
public interface Bag {
    boolean add(Object item);
    boolean remove(Object item);
    boolean contains(Object item);
    int numItems();
    Object grab();
    Object[] toArray();
}
```

Note that there is no "getItem" method for accessing a particular item in the bag. Instead, there is a method called `grab` that accesses a random item in the bag. Why does this make sense, given the characteristics of the Bag ADT?

```
Because the items in a Bag are not ordered or given a position, so you can't request an item at a given location in the Bag.
```

If you want to process all of the items in the bag, you can use the `toArray` method, which creates and returns an array containing all of the items.

We then created one implementation of the Bag ADT. To do so, we created a class called `ArrayBag` that implements the `Bag` interface:

```
public class ArrayBag implements Bag {
}
```

Because we make this claim in the class header, the compiler checks to make sure that the `ArrayBag` class has method definitions for all of the methods whose headers are listed in the definition of the `Bag` interface. If it doesn’t, the compiler will produce an error.

Recall that an `ArrayBag` object has two fields:

```
public class ArrayBag implements Bag {
    private Object[] items;
    private int numItems;
```
ArrayBag Methods
In lecture, we looked at the ArrayBag implementation of the add method:

```java
public boolean add(Object item) {
    if (item == null)
        throw new IllegalArgumentException("item must be non-null");
    if (numItems == items.length)
        return false;                  // no more room!
    else {
        items[numItems] = item;
        numItems++;
        return true;
    }
}
```

It returns true if it is able to add the item, and false if it cannot add the item (because the bag is full).

Now let's look at the ArrayBag implementation of the contains method:

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

It returns true if the bag contains an occurrence of the specified item, and false if it does not contain any occurrences of that item.

Note that we have to use the equals method, rather than ==, because we're comparing objects.

Why can the method safely return true inside the loop?

Because all we need to know is that it contains at least one occurrence.

Why can't the method return false inside the loop, as shown below?

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

Because the item might still be found at a later position in the array.
Let's now write the `ArrayBag` implementation of the `remove()` method:

```java
/*
 * remove - removes the first occurrence of the specified
 * item (if any) from the Bag. Returns true on success
 * and false if an object equal to the specified item is
 * not in the Bag.
 */

public boolean remove(Object item) {
    for (int i = 0; i < numItems; i++) {
        if (items[i] != null && items[i].equals(item)) {
            // Shift the remaining items left by one.
            System.arraycopy(items, i+1, items, i, numItems-i-1);
            items[numItems-1] = null;
            numItems--;
            return true;
        }
    }
    return false;  // item not found
}
```

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

```java
/*
 * containsAll - does this ArrayBag contain all of the
 * items in otherBag? Returns true if it does, and
 * false if it does not, or if otherBag is null or empty.
 */

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 (!contains(otherItems[i])) {
            return false;
        }
    }
    return true;
}
```

Why is the type of the parameter `Bag` rather than `ArrayBag`?

Because we want it to work with any implementation of the Bag ADT.

Why does it use the `numItems()` and `toArray()` methods, rather than using the `numItems` and `items` fields?

Because those fields are not part of the interface, and thus the code would not compile. Other implementations may use different fields, so we need to rely instead on methods that we know are supported by all implementations of the interface.
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?

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

**Answer:**

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

```java
ArrayBag b2 = b1;
```

What would our memory diagram look like after this assignment?

**Answer:**
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.

```java
public ArrayBag(Bag other) {
    // We could also throw an exception in this case.
    if (other == null) {
        items = new Object[DEFAULT_MAX_SIZE];
        numItems = 0;
        return;
    }

    // Note: We create an array that allows room for
    // additional items. There's no way to determine the
    // max size of the other bag!
    int maxSize = other.numItems() * 2;
    if (maxSize < DEFAULT_MAX_SIZE)
        maxSize = DEFAULT_MAX_SIZE;
    items = new Object[maxSize];
    numItems = 0;

    // Add the items in the other bag to this bag.
    // Note: using add() allows us to avoid updating
    // numItems, since add() does it for us.
    Object[] otherItems = other.toArray();
    for (int i = 0; i < otherItems.length; i++)
        add(otherItems[i]);
}
```

**Extra: Interfaces in Java**

In addition to specifying a full ADT, an interface in Java can also be used to specify that one or more classes support some type of functionality.

For example, let's say that we're writing a video game, and that we want to write a method that draws all of the objects that are visible in the game. The method will look something like this:

```java
public void drawAll(Object[] gameObjects) {
    for (int i = 0; i < gameObjects.length; i++)
        gameObjects[i].draw();
}
```

The objects in the `gameObjects` array could belong to many different classes, not all of which are subclasses of each other. For the purposes of our `drawAll()` method, all that we care is that all of the objects have some type of `draw()` method.

Unfortunately the method as written above will produce an error at compilation. Why?

Because the array is declared to be of type `Object`, and the `Object` class does not have a `draw()` method.
How can we assure the compiler that the objects in the array all have a `draw()` method without restricting the array to hold only objects of a given type?

The answer is to use an interface:

```java
public interface Drawable {
    void draw();
}
```

Given this interface, we then do the following:

1. Specify that all components of the game are classes that implement this interface:

   ```java
   public class GameFigure implements Drawable {
       ...
   }
   public class Spaceship extends Vehicle implements Drawable {
       ...
   }
   etc.
   ```

2. Rewrite the method so that the argument uses the interface as the type:

   ```java
   public void drawAll(Drawable[] gameObjects) {
       for (int i = 0; i < gameObjects.length; i++)
           gameObjects[i].draw();
   }
   ```

The `gameObjects` array can now hold references to objects of any type that implements the `Drawable` interface. What principle of object-oriented programming is at work here?

```
Polymorphism, because the same code is operating on objects of different types.
```