



BOSTON
UNIVERSITY

Exercise: A Bag Data Structure

A Bag

- A bag is just a container for a group of data items.
 - analogy: a bag of candy
- The positions of the data items don't matter (unlike a sequence).
 - $\{3, 2, 10, 6\}$ is equivalent to $\{2, 3, 6, 10\}$
- The items do not need to be unique (unlike a set).
 - $\{7, 2, 10, 7, 5\}$ isn't a set, but it is a bag

Today's Goal:

- Create a blueprint class for bags.
- Each object of this class will represent an entire bag of items.
 - example: one object might represent {3, 2, 10, 6}
- This will be an example of a collection class.
 - each object will represent a collection of items

Implementation

- The operations we want each bag object to support:
 - `add(item)`: add `item` to the bag
 - `remove(item)`: remove one occurrence of `item` (if any) from the bag
 - `contains(item)`: check if `item` is in the bag
 - `numItems()`: get the number of items in the bag
 - `grab()`: get an item at random, without removing it
 - reflects the fact that the items don't have a position (and thus we can't say "get the 5th item in the bag")
 - `toArray()`: get an array containing the current contents of the bag
- We want the bag to be able to store objects of any type!

Using an Array (Top Hat!)

- One way to store the items in the bag is to use an array:

```
public class ArrayBag {  
    private _____[] items;  
    ...  
}
```

- What type should the array be?
- This allows us to store any type of object in the items array, thanks to the power of polymorphism:

```
ArrayBag bag = new ArrayBag();  
bag.add("hello");  
bag.add(new Rectangle(20, 30));
```

- How could we keep track of how many items are in a bag?

Method Overloading

- In Java, two methods in a given class can have the same name.
- To do so, the methods must have:
 - a different number of parameters and/or
 - parameters of different types
- We saw this earlier with the substring methods in String:

```
String substring(int beginIndex, int endIndex)
String substring(int beginIndex)
```
- This is known as method overloading.
 - When a method call is made, the compiler uses the values being passed in to figure out which version to call.

Two Constructors for the ArrayBag Class

```
public class ArrayBag {
    private Object[] items;
    private int numItems;
    public static final int DEFAULT_MAX_SIZE = 50;

    public ArrayBag() {
        this.items = new Object[DEFAULT_MAX_SIZE];
        this.numItems = 0;
    }

    public ArrayBag(int maxSize) {
        if (maxSize <= 0) {
            throw new IllegalArgumentException("maxSize must be > 0");
        }
        this.items = new Object[maxSize];
        this.numItems = 0;
    }
    ...
}
```

Keeping count of ArrayBag objects

```
public class ArrayBag {
    private Object[] items;
    private int numItems;
    public static final int DEFAULT_MAX_SIZE = 50;
    private static int numBagsCreated = 0;

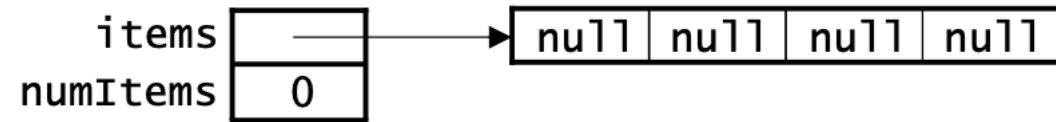
    public ArrayBag() {
        this.items = new Object[DEFAULT_MAX_SIZE];
        this.numItems = 0;
        numBagsCreated++;
    }

    public ArrayBag(int maxSize) {
        if (maxSize <= 0) {
            throw new IllegalArgumentException("maxSize must be > 0");
        }
        this.items = new Object[maxSize];
        this.numItems = 0;
        numBagsCreated++;
    }

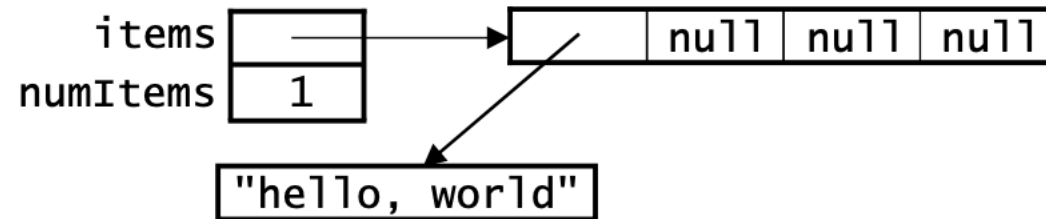
    ...
}
```

add(item)

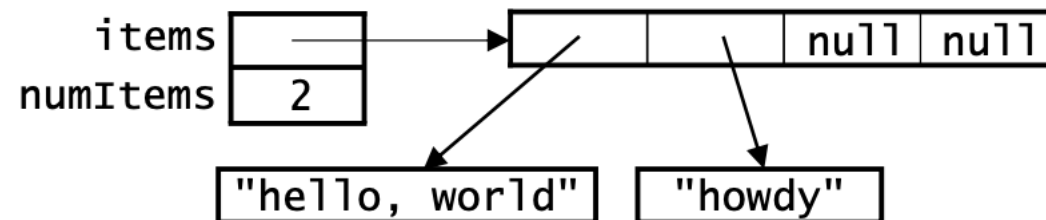
- We fill the array from left to right. Here's an empty bag:



- After adding the first item:

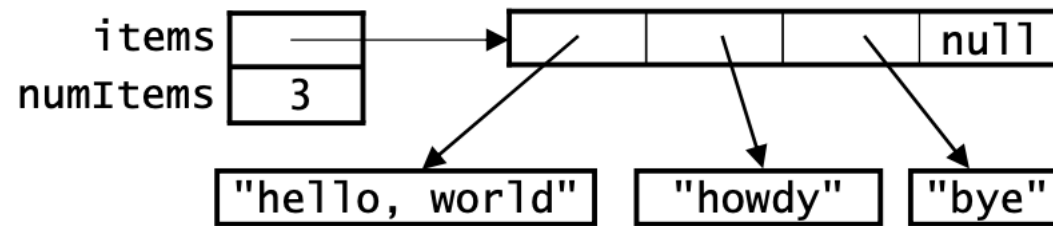


- After adding the second item:

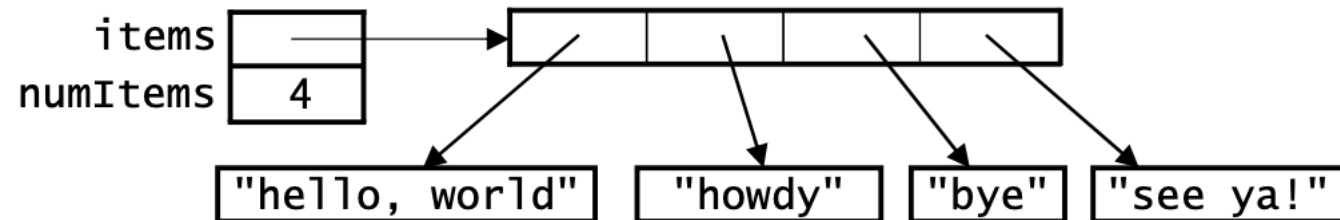


add(item)

- After adding the third item:



- After adding the fourth item:



- At this point, the ArrayBag is full!
 - it's non-trivial to "grow" an array, so we don't!
 - additional items cannot be added until one is removed

add(item)

```
public class ArrayBag {
    private Object[] items;
    private int numItems;
    ...

    public _____ add(Object item) {
        if (item == null) {
            throw new IllegalArgumentException("no nulls");
        } else if (this.numItems == this.items.length) {
            _____ // no more room!
        } else {
            this.items[this.numItems] = item;
            this.numItems++;
            _____ // success!
        }
    }
    ...
}
```

Replace `this.items.length` with `maxSize`?

```
public class ArrayBag {
    private Object[] items;
    private int numItems;
    ...
    public ArrayBag(int maxSize) {
        if (maxSize <= 0) {
            throw new IllegalArgumentException(...);
        }
        this.items = new Object[maxSize];
        this.numItems = 0;
    }

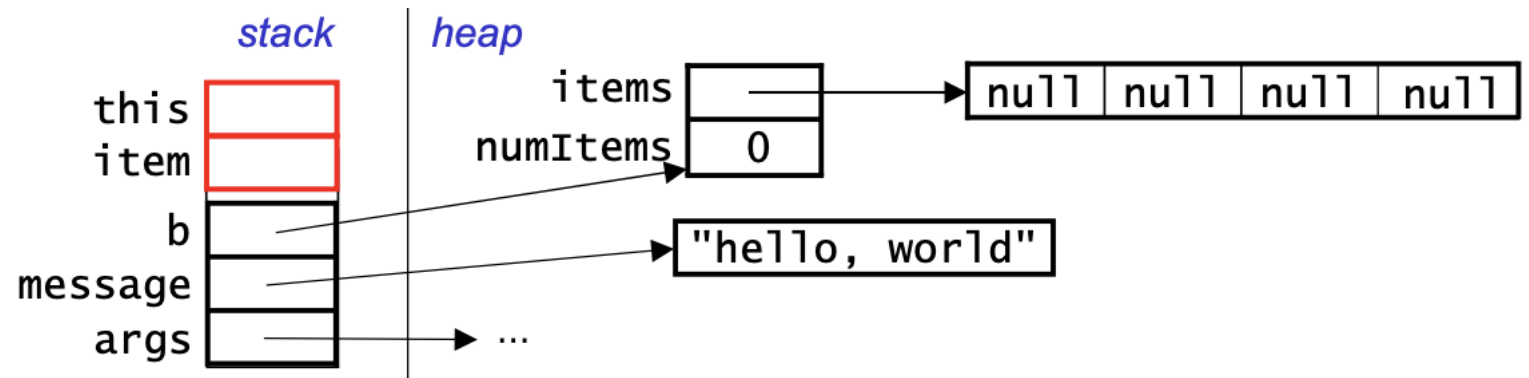
    public boolean add(Object item) {
        if (item == null) {
            throw new IllegalArgumentException("no nulls");
        } else if (this.numItems == this.items.length) {
            return false; // no more room!
        } else {
            this.items[this.numItems] = item;
            this.numItems++;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

Memory: Adding an item

```
public static void main(String[] args) {  
    String message = "hello, world";  
    ArrayBag b = new ArrayBag(4);  
    b.add(message);  
    ...  
}
```

```
public boolean add(Object item) {  
    ...  
    else {  
        this.items[this.numItems] = item;  
        this.numItems++;  
        return true;  
    } ...  
}
```

- add's stack frame includes:
 - item, which stores...
 - this, which stores...

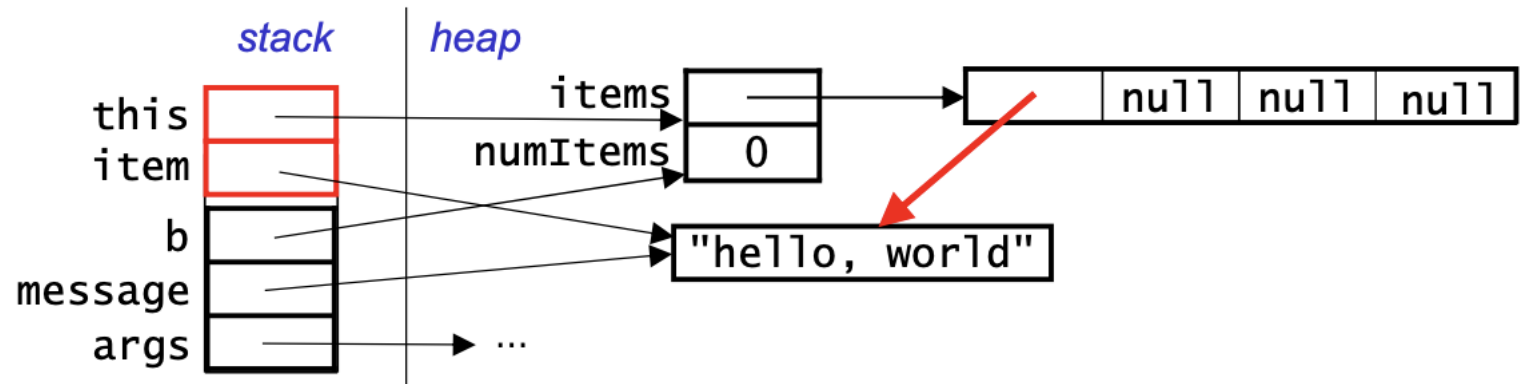


Memory: Adding an item

```
public static void main(String[] args) {  
    String message = "hello, world";  
    ArrayBag b = new ArrayBag(4);  
    b.add(message);  
    ...  
}
```

```
public boolean add(Object item) {  
    ...  
    else {  
        this.items[this.numItems] = item;  
        this.numItems++;  
        return true;  
    } ...  
}
```

- The method modifies the `items` array and `numItems`.
 - note that the array holds a copy of the reference to the item, not a copy of the item itself.

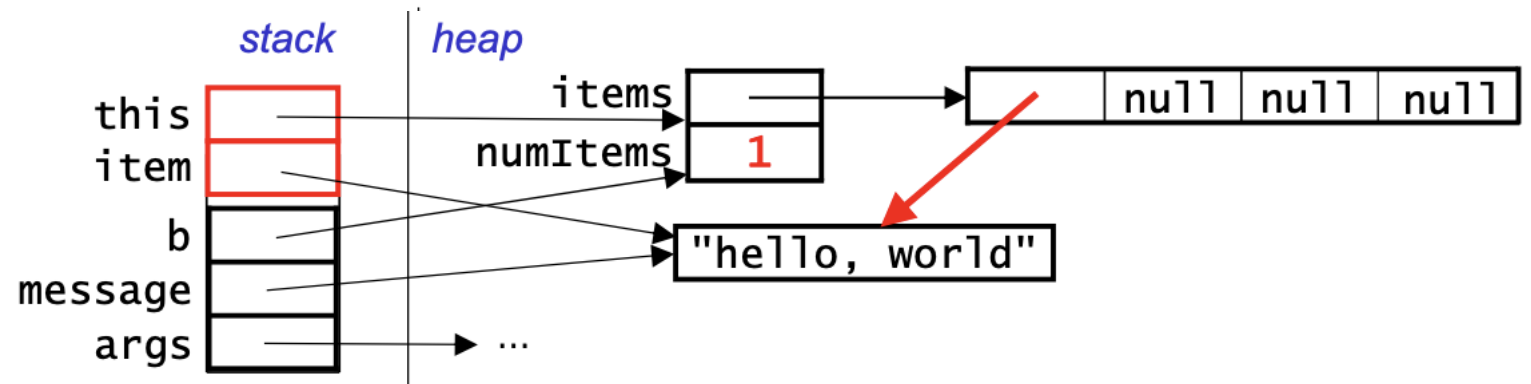


Memory: Adding an item

```
public static void main(String[] args) {  
    String message = "hello, world";  
    ArrayBag b = new ArrayBag(4);  
    b.add(message);  
    ...  
}
```

```
public boolean add(Object item) {  
    ...  
    else {  
        this.items[this.numItems] = item;  
        this.numItems++;  
        return true;  
    } ...  
}
```

- The method modifies the `items` array and `numItems`.
 - note that the array holds a copy of the reference to the item, not a copy of the item itself.

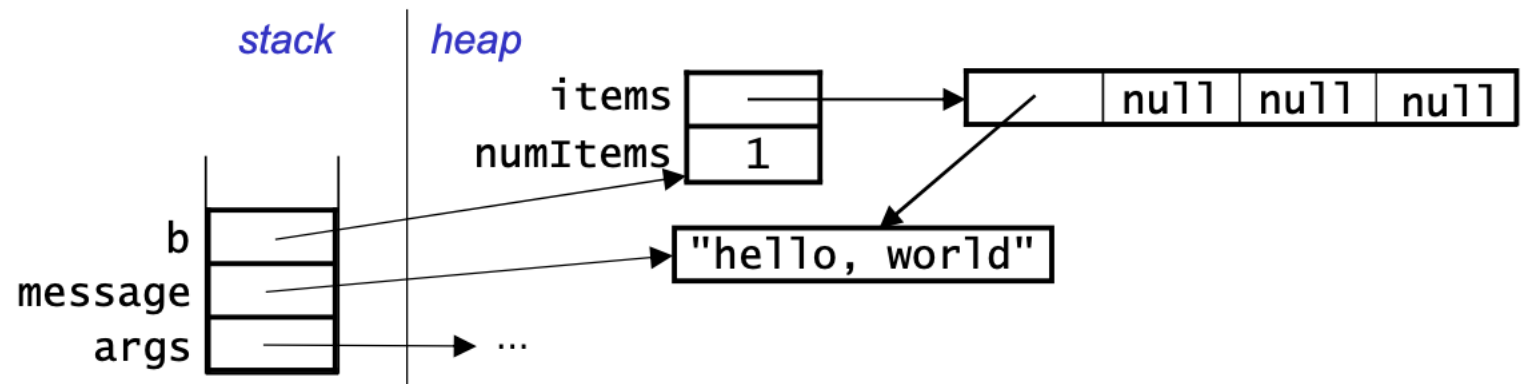


Memory: Adding an item

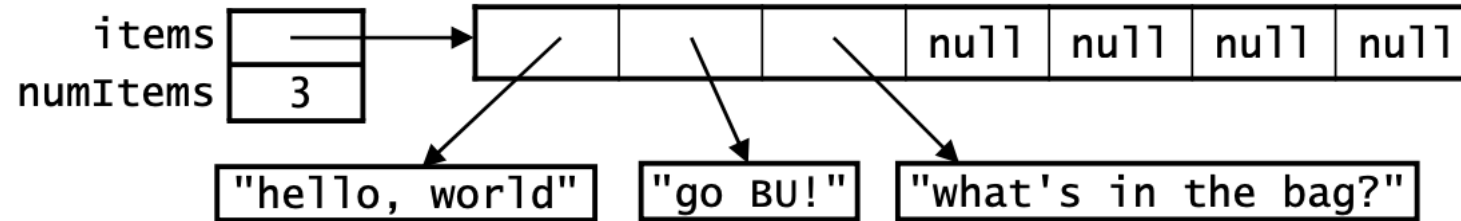
```
public static void main(String[] args) {  
    String message = "hello, world";  
    ArrayBag b = new ArrayBag(4);  
    b.add(message);  
    ...  
}
```

```
public boolean add(Object item) {  
    ...  
    else {  
        this.items[this.numItems] = item;  
        this.numItems++;  
        return true;  
    } ...  
}
```

- After the method call returns, add's stack frame is removed from the stack.



```
contains(item)
```



- Let's write the `ArrayBag contains()` method
 - should return `true` if an object equal to `item` is found, and `false` otherwise.

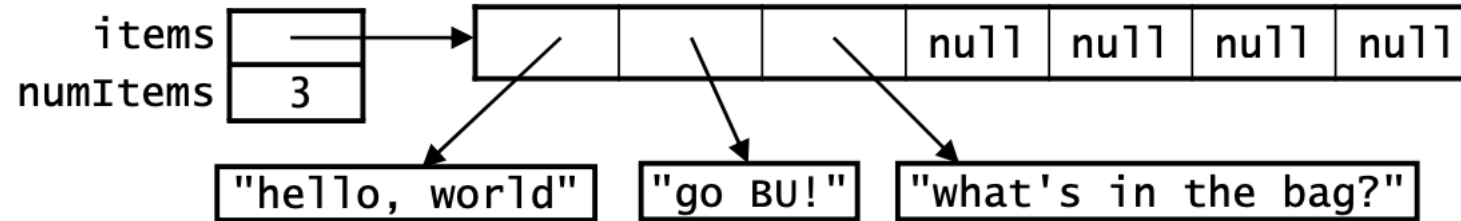
```

_____ contains (_____ item) {

```

}

contains (item)



- Let's write the `ArrayBag contains()` method
 - should return true if an object equal to item is found, and false otherwise.

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

An incorrect contains (item)

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

What's the problem with this?

containsAll (other)

- Now let's add a method `containsAll (ArrayBag other)`
 - returns true if all of the items in `other` are in the called `ArrayBag`, and false otherwise.

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

- Because this method is part of the `ArrayBag` class, it is able to access the private fields of the `other ArrayBag`

Type mismatch

- Here are the headers of two `ArrayBag` methods:

```
public boolean add(Object item)
public Object grab()
```

- Polymorphism allows us to pass `String` objects into `add()`:

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

- However, this will not work:

```
String str = stringBag.grab(); // compiler error
```

- the return type of `grab()` is `Object`
- `Object` isn't a subclass of `String`, so polymorphism doesn't help!

- Instead, we need to use a type cast:

```
String str = (String)stringBag.grab();
```

- this cast doesn't actually change the value being assigned
- it just reassures the compiler that the assignment is okay

Exercise: `remove(item)`

- remove one occurrence of `item` (if any) from the bag