



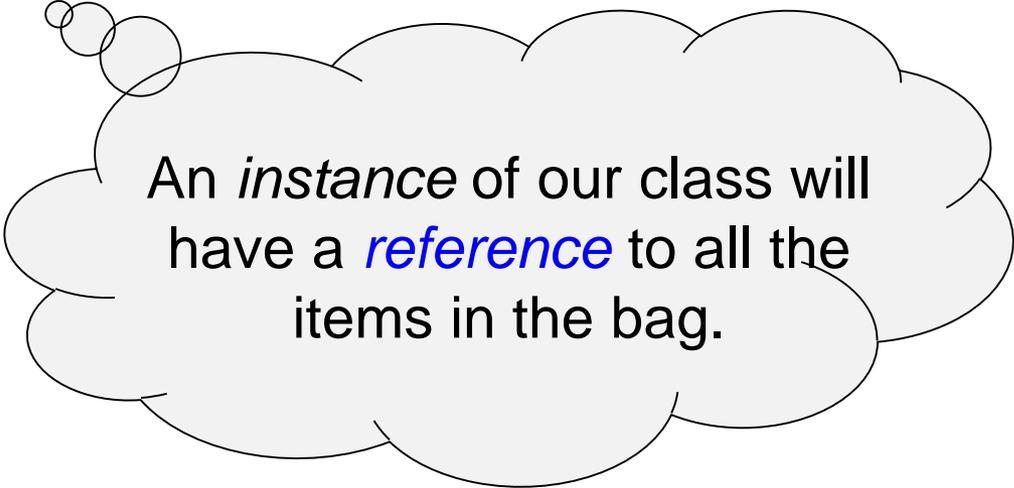
A Bag Data Structure

Computer Science 112
Boston University

Christine Papadakis-Kanaris

A Bag Data Structure

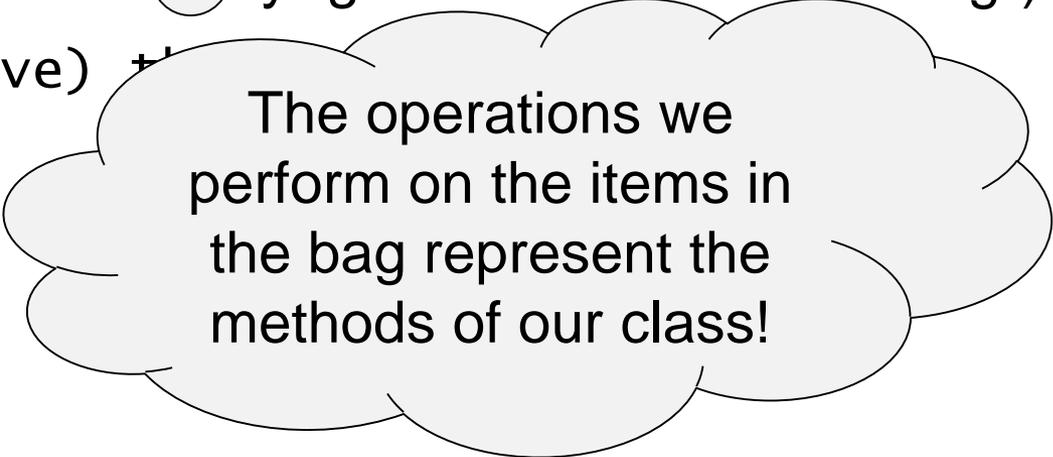
- A bag is just a container for a group of data items.
 - analogy: a bag of candy
- The positions of the data items don't matter (unlike a sequence).
 - {3, 2, 10, 6} is equivalent to {2, 3, 6, 10}
- The items do *not* need to be unique (unlike a set).
 - {7, 2, 10, 7, 5} isn't a set, but it is a bag



An *instance* of our class will have a *reference* to all the items in the bag.

A Bag Data Structure

- The operations we want our Bag to support:
 - **add** an item to the Bag
 - **remove** one occurrence of an item (if any) from the Bag
 - **check** if a specific item is in the Bag
 - **count** the the number of items in the Bag
 - **select** 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")
 - **carry** (or move) +



The operations we perform on the items in the bag represent the methods of our class!

Storing Items in a Bag Data Structure

an array of Objects

- We store the items in

```
public class ArrayBag {
```

```
}
```

Storing Items in a Bag Data Structure

an array of Objects

- We store the items in

```
public class ArrayBag {
```

```
}
```

How could
we store
one item of
any type?



Storing Items in a Bag Data Structure

an array of Objects

- We store the items in

```
public class ArrayBag {  
    Object item;  
  
}
```

Storing Items in a Bag Data Structure

an array of Objects

- We store the items in

```
public class ArrayBag {
```

```
}
```

How could
we store
multiple
items of
any type?



Storing Items in a Bag Data Structure

an array of Objects

- We store the items in an array of type Object.

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

A large orange thought bubble with a black outline, containing the text "Why an array of Objects?". Three smaller orange circles of increasing size lead from the top left of the bubble towards the main bubble.

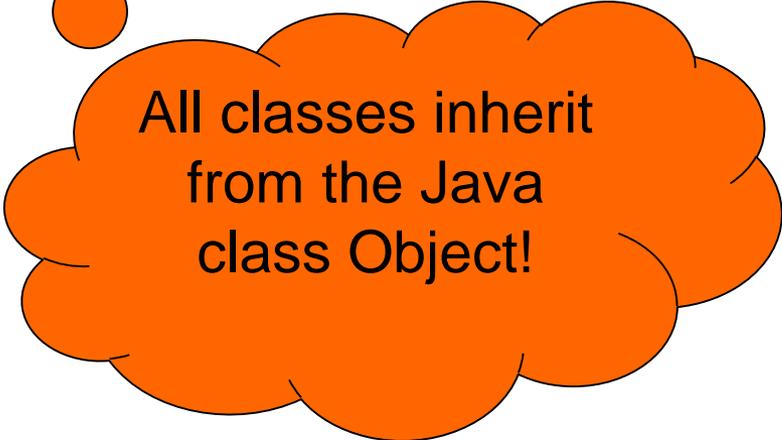
Why an
array of Objects?

Storing Items in a Bag Data Structure

an array of Objects

- We store the items in an array of type Object.

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



All classes inherit
from the Java
class Object!

Storing Items in a Bag Data Structure

an array of any object type

- We store the items in an array of type `Object`.

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

- 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));
```

Storing Items in a Bag Data Structure

an array of any object type

- We store the items in an array of type `Object`.

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

- 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));
```

Storing Items in a Bag Data Structure

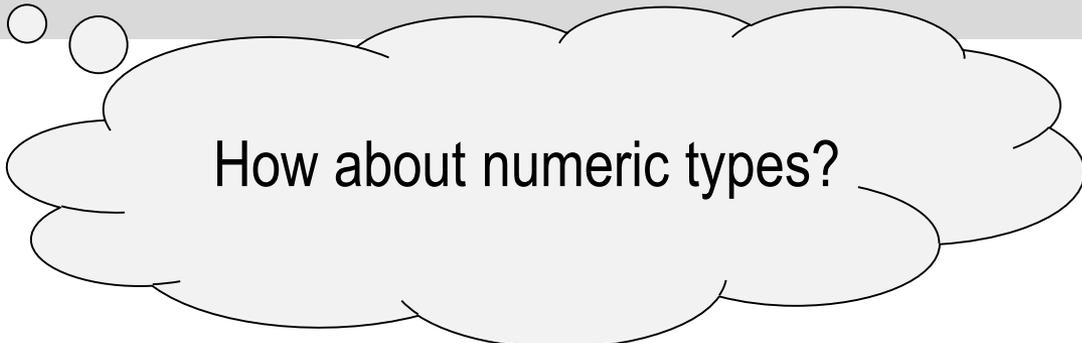
an array of any object type

- We store the items in an array of type Object.

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

- 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 about numeric types?

Storing Items in a Bag Data Structure

an array of any object type

- We store the items in an array of type Object.

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

Need to create numeric objects using the wrapper classes (e.g. Integer).

- This allows us to store *anything*, thanks to the power of **polymorphism**.

```
ArrayBag bag = new ArrayBag();  
bag.add("hello");  
bag.add(new Rectangle(20, 30));  
bag.add(new Integer(5));
```

Storing Items in a Bag Data Structure

an array of any object type

- We store the items in an array of type Object.

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

Java has a feature called *autoboxing* that will automatically create an instance of the appropriate type.

- This allows us to store any object in an array, thanks to the power of *polymorphism*.

```
ArrayBag bag = new ArrayBag();  
bag.add("hello");  
bag.add(new Rectangle(20, 30));  
bag.add(5);
```

Storing Items in a Bag Data Structure

an array of any object type

- We store the items in an array of type Object.

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

- 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 many items can
our bag hold?

Storing Items in a Bag Data Structure

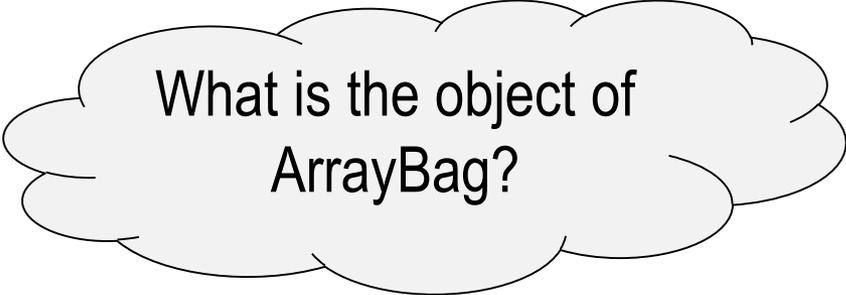
an array of any object type

- We store the items in an array of type Object.

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

- 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));
```



What is the object of
ArrayBag?

Storing Items in a Bag Data Structure

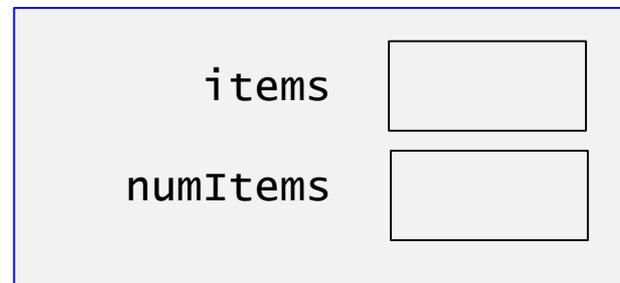
an array of any object type

- We store the items in an array of type Object.

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

- 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));
```



an ArrayBag object

Storing Items in a Bag Data Structure

an array of any object type

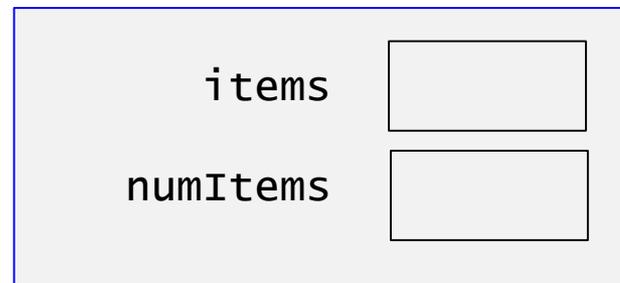
- We store the items in an array of

```
public class ArrayBag  
    private Object[]  
    private int num  
    ...  
}
```

Calls the constructor
to initialize the data
members of our object!

- This allows us to store *any type* object in the items array, thanks to the power of polymorphism:

```
ArrayBag bag = new ArrayBag(); // no arg  
bag.add("hello");  
bag.add(new Rectangle(20, 30));
```



an ArrayBag object

Storing Items in a Bag Data Structure

an array of any object type

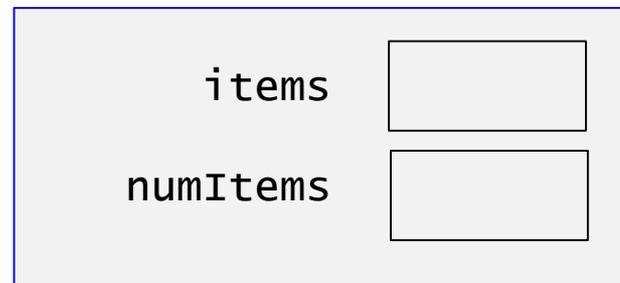
- We store the items in an array of

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

Calls the constructor
to initialize the data
members of our object!

- This allows us to store *any type* object in the items array, thanks to the power of polymorphism:

```
ArrayBag bag = new ArrayBag(5); // custom
bag.add("hello");
bag.add(new Rectangle(20, 30));
```



an ArrayBag object

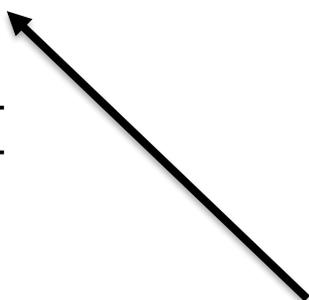
Two Constructors for the ArrayBag Class

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

- We can have two different constructors!
 - the parameters must differ in some way

Two Constructors for the ArrayBag Class

```
public class ArrayBag {  
    private Object[] items;  
    private int numItems;  
  
    public ArrayBag() {  
        this.items = new Object[50];  
        this.numItems = 0;  
    }  
    public ArrayBag(int maxSize) {  
        ...  
    }  
}
```

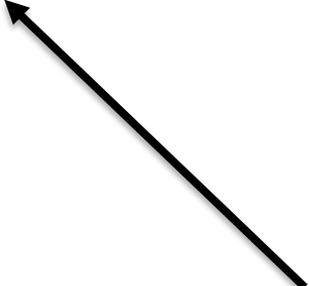


- We can have two different constructors!
 - the parameters must differ in some way
- The first constructor is useful for small bags.
 - creates an array with room for 50 items.

Two Constructors for the ArrayBag Class

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

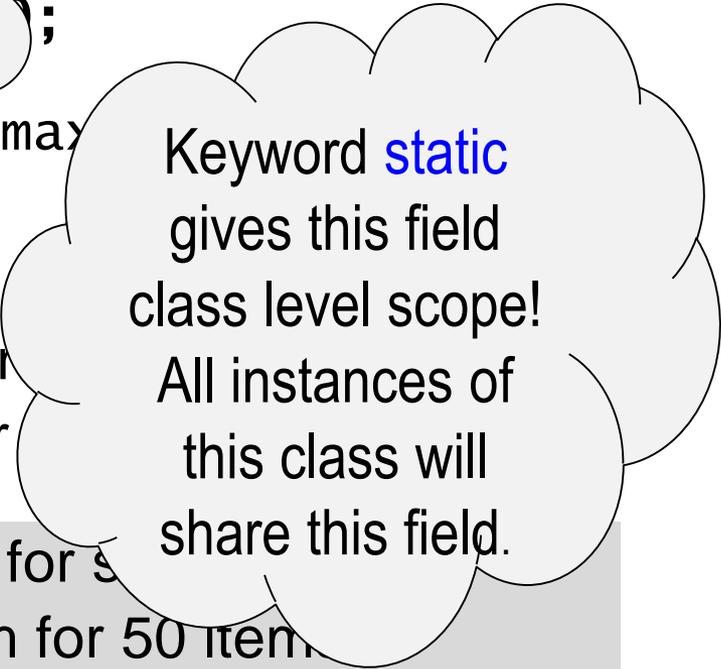
    public ArrayBag() {
        this.items = new Object[DEFAULT_MAX_SIZE];
        this.numItems = 0;
    }
    public ArrayBag(int maxSize) {
        ...
    }
}
```



- We can have two different constructors!
 - the parameters must differ in some way
- The first constructor is useful for small bags.
 - creates an array with room for 50 items.

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 max  
    ...  
}
```



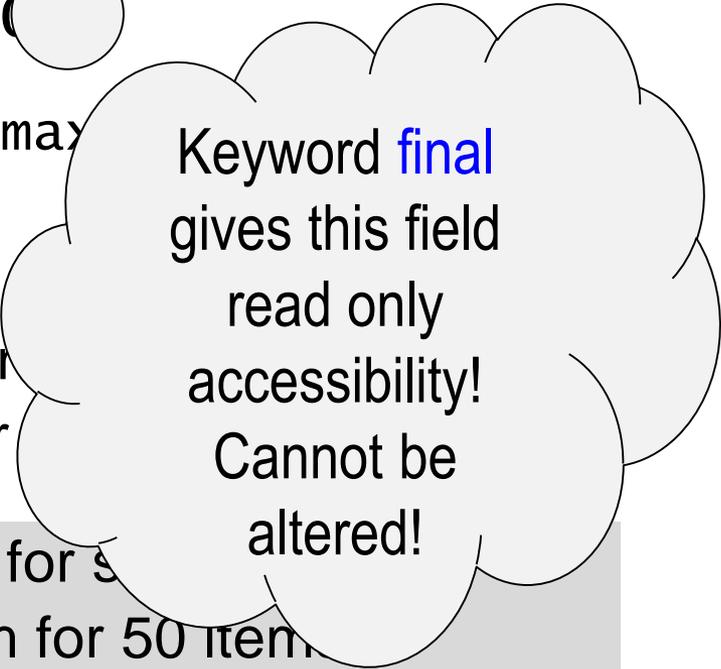
Keyword **static**
gives this field
class level scope!
All instances of
this class will
share this field.

- We can have two different constructors
 - the parameters must differ
- The first constructor is useful for simple use cases
 - creates an array with room for 50 items

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 maxCapacity) {  
        ...  
    }  
}
```

- We can have two different constructors
 - the parameters must differ
- The first constructor is useful for situations where:
 - creates an array with room for 50 items



Keyword **final**
gives this field
read only
accessibility!
Cannot be
altered!

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) {
        ...
    }
}
```

- We can have two different constructors!
 - the parameters must differ in some way
- The first one is useful for small bags.
 - creates an array with room for 50 items.
- The second one allows the client to specify the max # of items.

Two Constructors for the ArrayBag Class

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

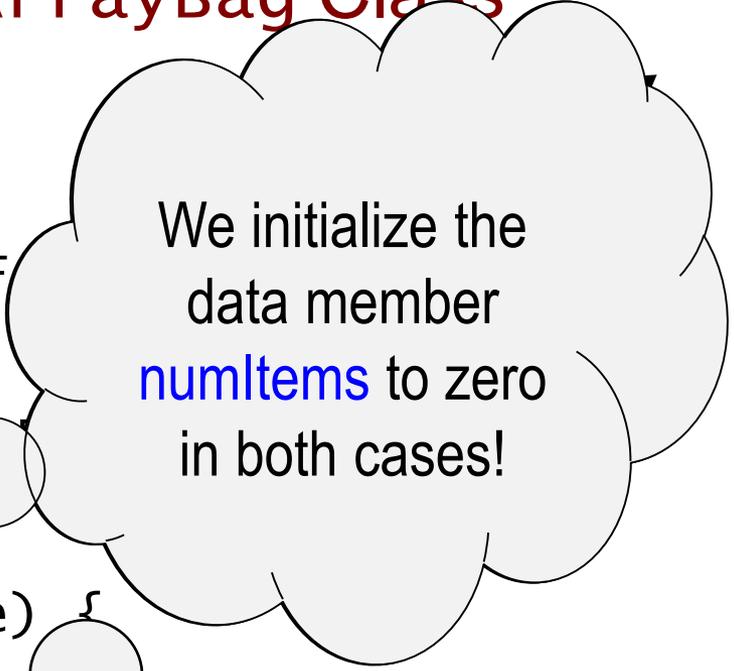
    public ArrayBag() {
        this.items = new Object[DEFAULT_MAX_SIZE];
        this.numItems = 0;
    }
    public ArrayBag(int maxSize) {
        if (maxSize <= 0) {
            throw new IllegalArgumentException(
                "maxSize must be > 0");
        }
        this.items = new Object[maxSize];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

- If the user inputs an invalid maxSize, we throw an exception.

Two Constructors for the ArrayBag Class

```
public class ArrayBag {
    private Object[] items;
    private int numItems;
    public static final int DEF

    public ArrayBag() {
        this.items = new Object[DEF];
        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;
    }
    ...
}
```



We initialize the data member `numItems` to zero in both cases!

- If the user inputs an invalid `maxSize`, we throw an exception.

Two Constructors for the ArrayBag Class

```
public class ArrayBag {
    private Object[] items;
    private int numItems;
    public static final int DEF

    public ArrayBag() {
        this.items = new Object[
        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;
    }
    ...
}
```



What if we wanted to keep track of *how many* bags we created?

- If the user inputs an invalid maxSize, we throw an exception.

Two Constructors for the ArrayBag Class

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

- If the user inputs an invalid maxSize, we throw an exception.

Two Constructors for the ArrayBag Class

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

Note that this
static member
is not declared
final because
.....

- If the user inputs an invalid maxSize, we throw an exception.

Two Constructors for the ArrayBag Class

```
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++;
    }
    ...
}
```

Increment the static variable every time that we create an ArrayBag.

Two Constructors for the ArrayBag Class

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

Increment the static variable

`this.items = new Object[DEFAULT_MAX_SIZE];`
Assume I have four different colors: **green**, **blue**, **red**, and **black**.

} Let's say I want to enhance the ArrayBag class to:

1. create a bag with a specific color, and
2. keep track of how many bags of each color I created.

What additional fields (data members) do I need to add to my Arraybag class, and what type of fields should they be, static or non-static (i.e. instance)?

}

...

}

Two Constructors

```
public class ArrayBag {
    private Object[] items;
    private int numItems;
    public static final int CAPACITY = 100;
    private static final int RED = 1;
    private static final int GREEN = 2;
    private static final int BLACK = 3;
    public ArrayBag() {
        this.items = new Object[CAPACITY];
    }
}
```

1. Add an *instance* variable to represent color (i.e. non static).
2. Change the constructor(s) to initialize this instance member, either using a *default* color in the no-arg constructor or by adding a parameter in the custom constructor.

Assume I have four different colors: **green**, **red**, and **black**.

} Let's say I want to enhance the ArrayBag class to:

1. create a bag with a specific color, and
2. keep track of how many bags of each color I created.

What additional fields (data members) do I need to add to my Arraybag class, and what type of fields should they be, static or non-static (i.e. instance)?

Two Constructors

```
public class ArrayBag {  
    private Object[] items;  
    private int numItems;  
    public static final int MAX_CAPACITY = 100;  
    private static int[] colorCounts;  
    public ArrayBag() {  
        this.items = new Object[MAX_CAPACITY];  
        numItems = 0;  
        colorCounts = new int[Color.values().length];  
    }  
}
```

1. Add an int **static** variable, one for each possible color **or**
2. Add an int **static** array where each element of the array represents a color.
3. Change the constructor(s) to update the appropriate *counter* each time a bag is created.

Assume I have four different colors: red, yellow, blue, and black.

} Let's say I want to enhance the ArrayBag class to:

1. create a bag with a specific color, and
2. keep track of how many bags of each color I created.

What additional fields (data members) do I need to add to my Arraybag class, and what type of fields should they be, static or non-static (i.e. instance)?

Two Constructors for the ArrayBag Class

```
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];  
        numBagsCreated++;  
    }  
}
```

Increment the static variable

Assume I have four different colors: **green**, **blue**, **red**, and **black**.

} Let's say I want to enhance the ArrayBag class to:

p

1. create a bag with a specific color, and
2. keep track of how many bags of each color I created.

What additional fields (data members) do I need to add to my Arraybag class, and what type of fields should they be, static or non-static (i.e. instance)?

} Now let's say I wanted to add a method that would allow me to change the color of a specific bag. What type of method would that be and what operations would it perform?
}

- The method should be an instance method because it will be called on a specific bag. **Example:**

```
ArrayBag bag = new ArrayBag(15, "green");
bag.changeColor( "blue" );
```

- The method should update the instance variable that represents the color of the bag, **and** update the class level counters. *In this example, the number of green bags should decrease by one and the number of blue bags should increase by one.*

static variable
and **black**.

} Let's say I want to create a Bag class to:

1. create a bag with a specific color, and
2. keep track of how many bags of each color I created.

What additional fields (data members) do I need to add to my Arraybag class and what type of fields should they be, static or non-static (i.e. instance)?

} Now let's say I wanted to add a method that would allow me to change the color of a specific bag. What type of method would that be and what operations would it perform?

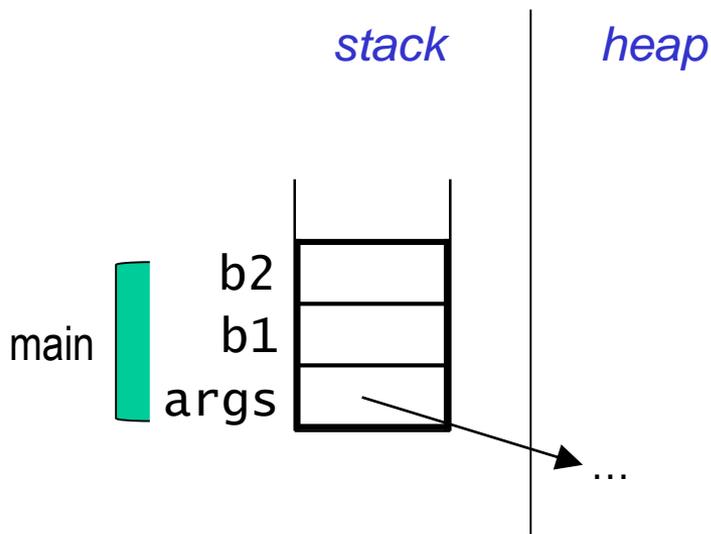
Two Constructors for the ArrayBag Class

```
public class ArrayBag {
    private Object[] items;
    private int numItems;
    public static final int DEFAULT_MAX_SIZE = 50;
    private static int numBagsCreated = 0;
    public ArrayBag() {
        // take advantage of constructor chaining
        // to consolidate the code
        this(DEFAULT_MAX_SIZE);
    }
    public ArrayBag(int maxSize) {
        if (maxSize <= 0) {
            throw new IllegalArgumentException(
                "maxSize must be > 0");
        }
        this.items = new Object[maxSize];
        this.numItems = 0;
        numBagsCreated++;
    }
    ...
}
```

Example: Creating Two ArrayBag Objects

```
// client
public static void main(String[] args) {
    ArrayBag b1 = new ArrayBag(2);
    ArrayBag b2 = new ArrayBag(4);
    ...
}
```

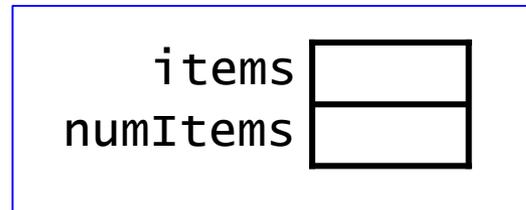
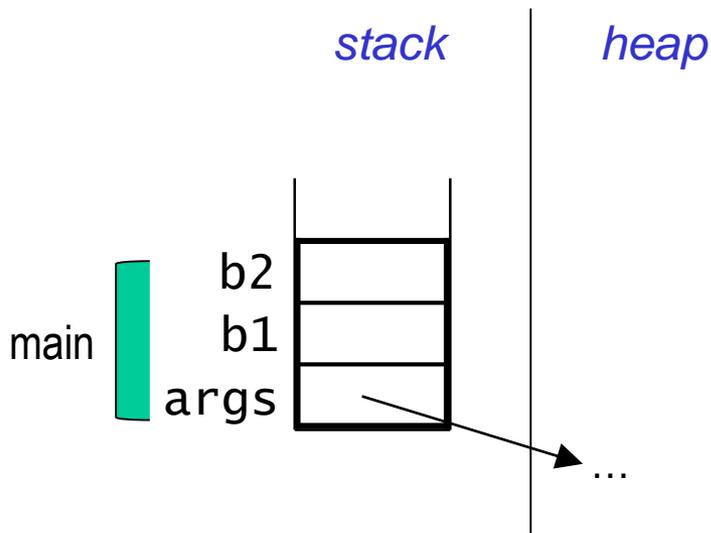
```
// constructor
public ArrayBag(int maxSize) {
    ... // error-checking
    this.items = new Object[maxSize];
    this.numItems = 0;
}
```



Example: Creating Two ArrayBag Objects

```
// client
public static void main(String[] args) {
    ArrayBag b1 = new ArrayBag(2);
    ArrayBag b2 = new ArrayBag(4);
    ...
}
```

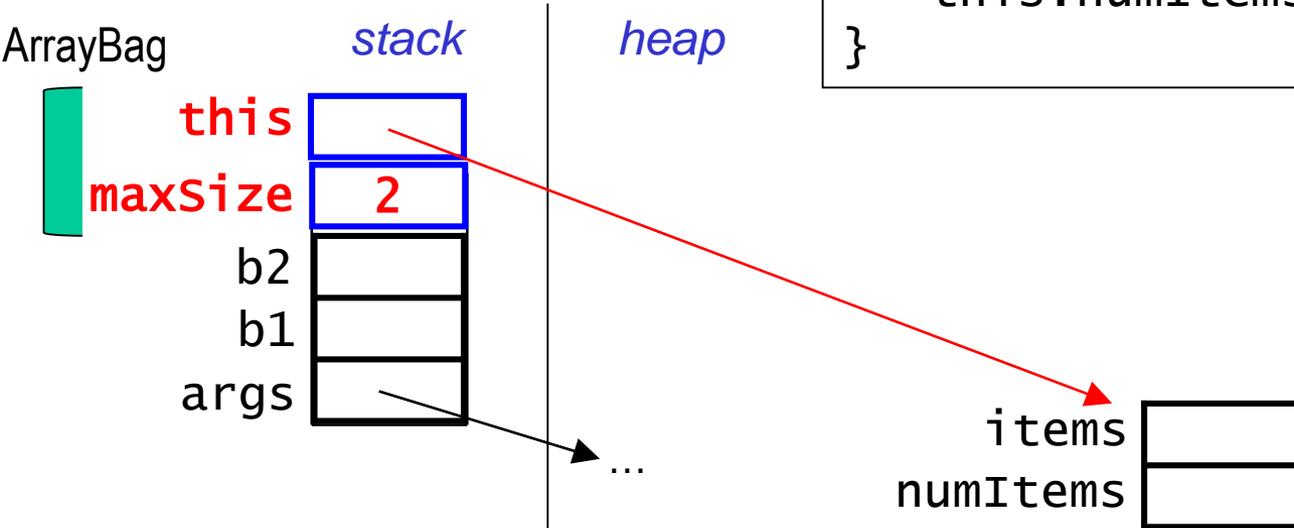
```
// constructor
public ArrayBag(int maxSize) {
    ... // error-checking
    this.items = new Object[maxSize];
    this.numItems = 0;
}
```



Example: Creating Two ArrayBag Objects

```
// client
public static void main(String[] args) {
    ArrayBag b1 = new ArrayBag(2);
    ArrayBag b2 = new ArrayBag(4);
    ...
}
```

```
// constructor
public ArrayBag(int maxSize) {
    ... // error-checking
    this.items = new Object[maxSize];
    this.numItems = 0;
}
```

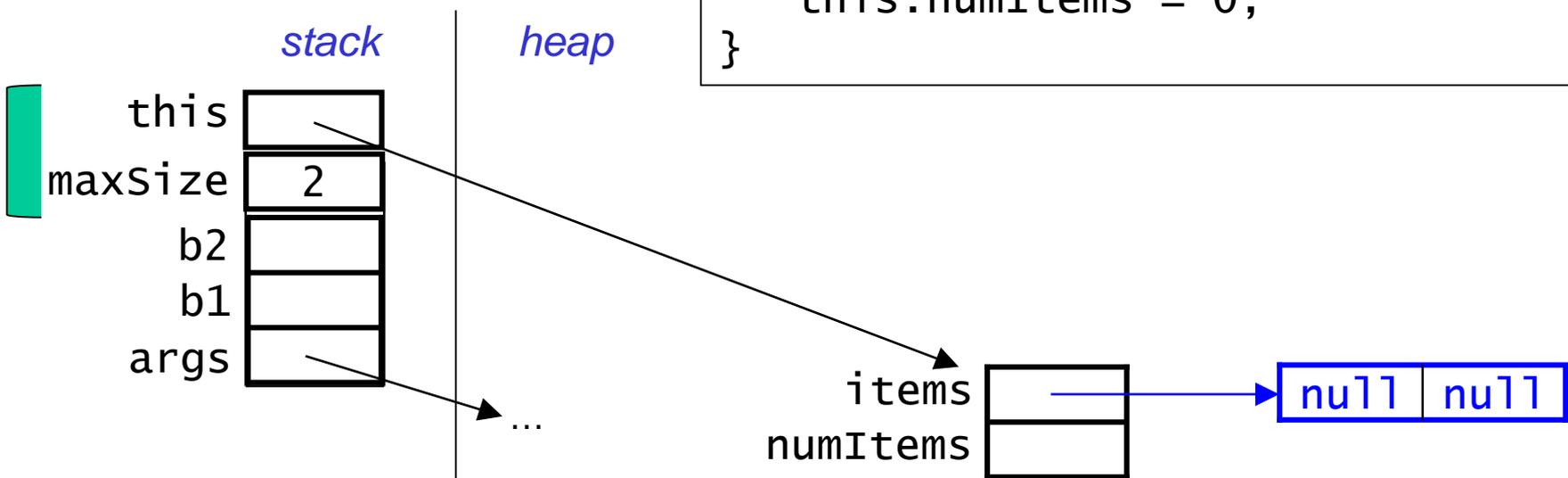


Note: We are not showing the `return address` that is also in this stack frame.

Example: Creating Two ArrayBag Objects

```
// client
public static void main(String[] args) {
    ArrayBag b1 = new ArrayBag(2);
    ArrayBag b2 = new ArrayBag(4);
    ...
}
```

```
// constructor
public ArrayBag(int maxSize) {
    ... // error-checking
    this.items = new Object[maxSize];
    this.numItems = 0;
}
```

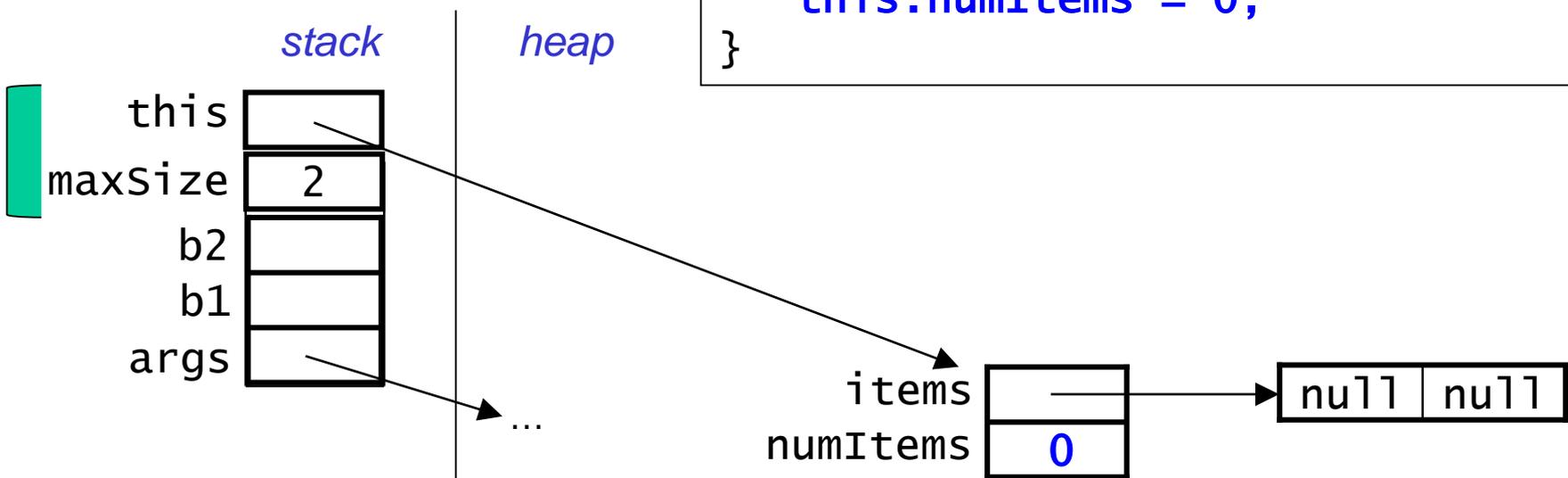


Note: We are not showing the `return address` that is also in this stack frame.

Example: Creating Two ArrayBag Objects

```
// client
public static void main(String[] args) {
    ArrayBag b1 = new ArrayBag(2);
    ArrayBag b2 = new ArrayBag(4);
    ...
}
```

```
// constructor
public ArrayBag(int maxSize) {
    ... // error-checking
    this.items = new Object[maxSize];
    this.numItems = 0;
}
```

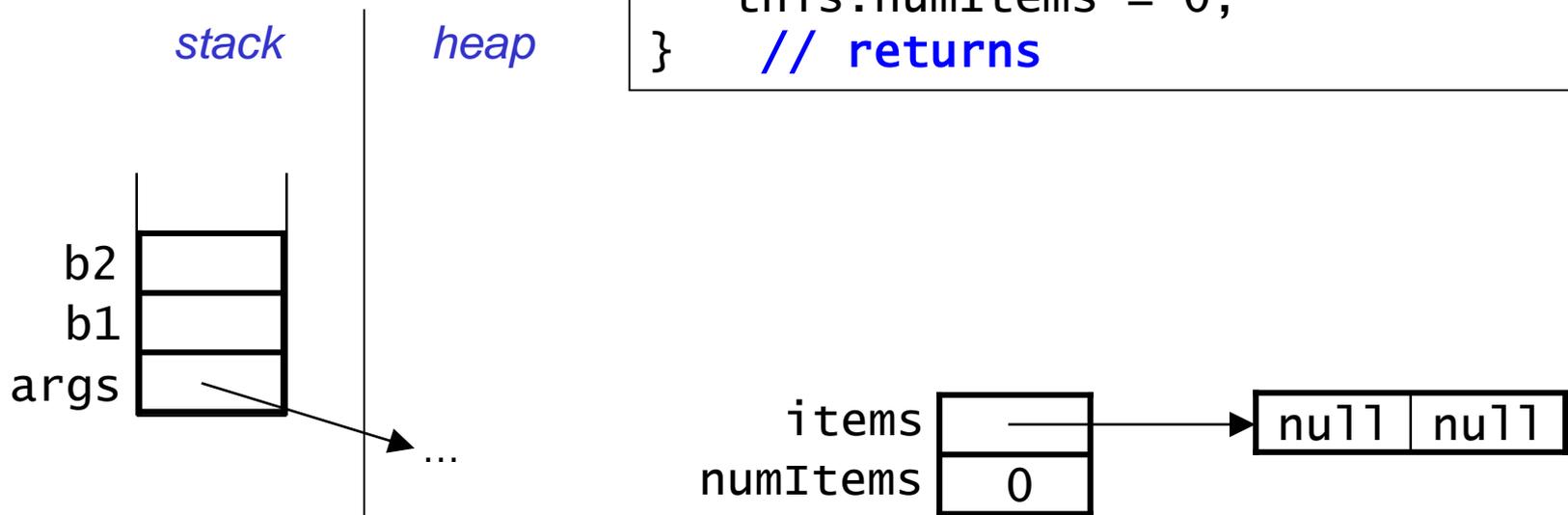


Note: We are not showing the `return address` that is also in this stack frame.

Example: Creating Two ArrayBag Objects

```
// client
public static void main(String[] args) {
    ArrayBag b1 = new ArrayBag(2);
    ArrayBag b2 = new ArrayBag(4);
    ...
}
```

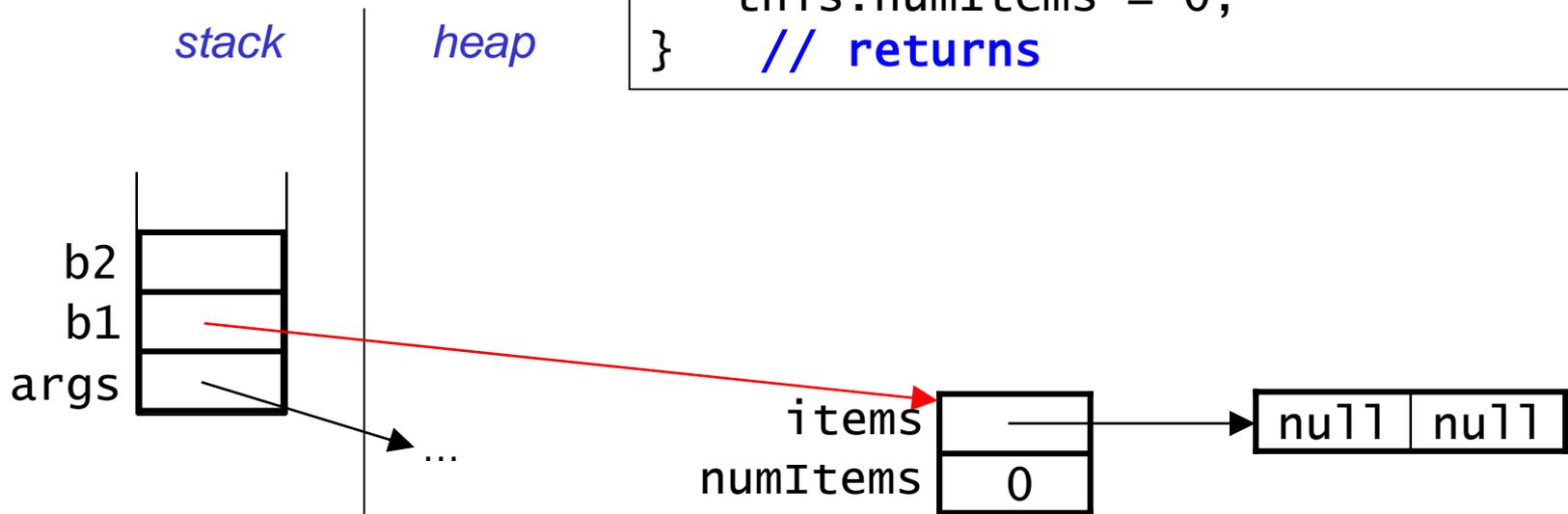
```
// constructor
public ArrayBag(int maxSize) {
    ... // error-checking
    this.items = new Object[maxSize];
    this.numItems = 0;
} // returns
```



Example: Creating Two ArrayBag Objects

```
// client
public static void main(String[] args) {
    ArrayBag b1 = new ArrayBag(2);
    ArrayBag b2 = new ArrayBag(4);
    ...
}
```

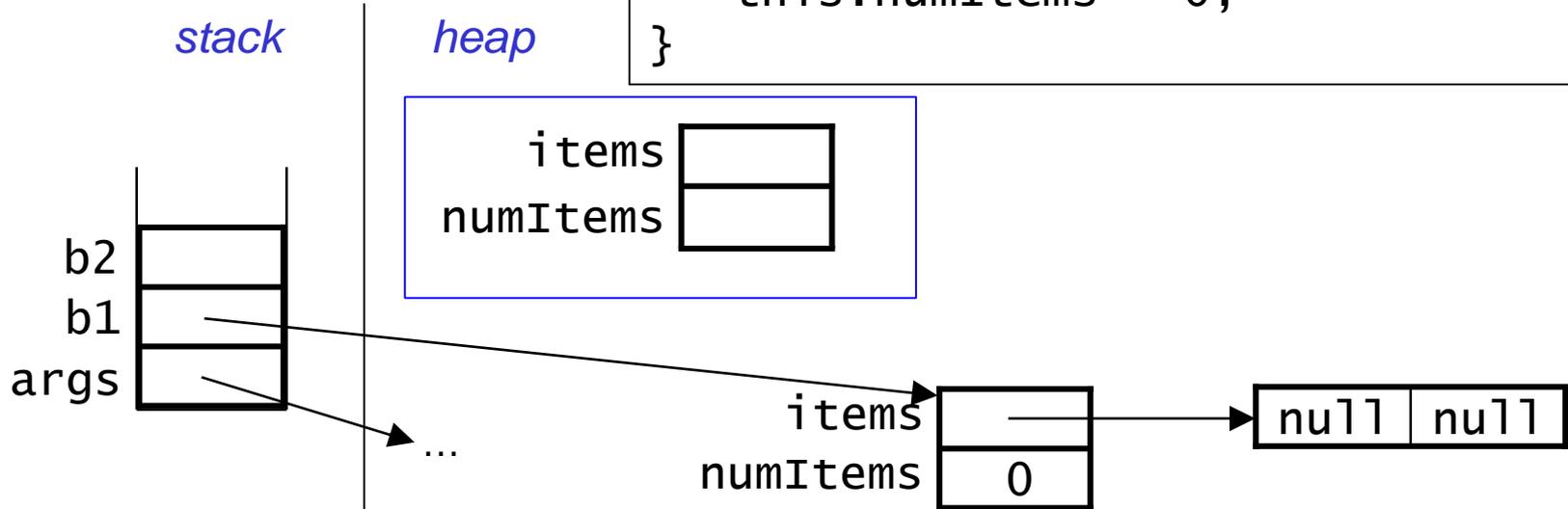
```
// constructor
public ArrayBag(int maxSize) {
    ... // error-checking
    this.items = new Object[maxSize];
    this.numItems = 0;
} // returns
```



Example: Creating Two ArrayBag Objects

```
// client
public static void main(String[] args) {
    ArrayBag b1 = new ArrayBag(2);
    ArrayBag b2 = new ArrayBag(4);
    ...
}
```

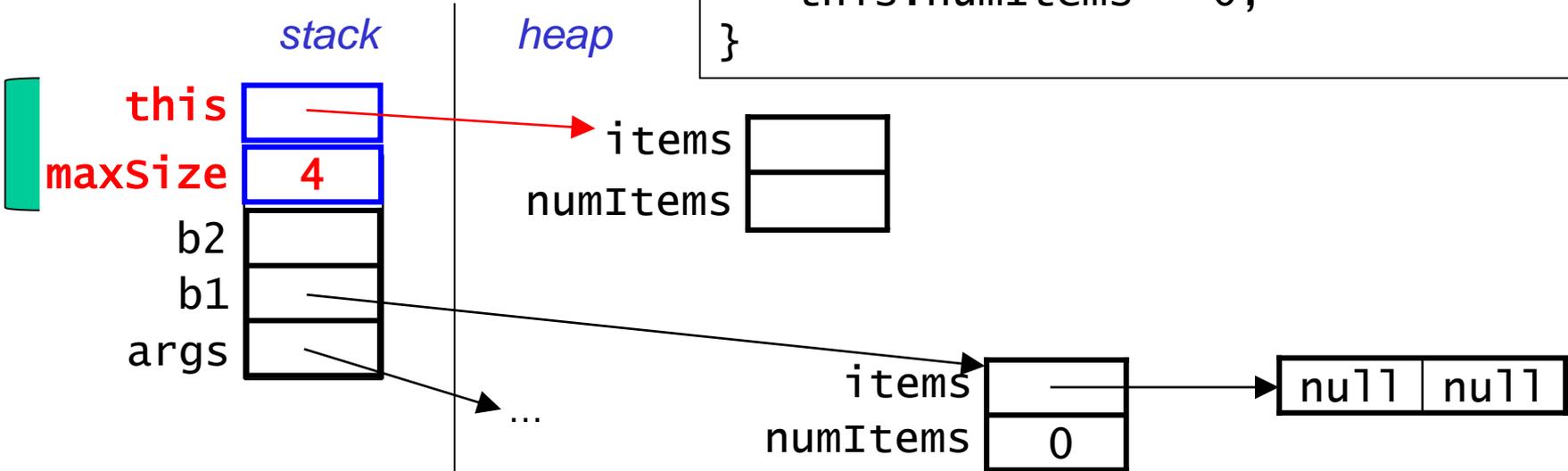
```
// constructor
public ArrayBag(int maxSize) {
    ... // error-checking
    this.items = new Object[maxSize];
    this.numItems = 0;
}
```



Example: Creating Two ArrayBag Objects

```
// client
public static void main(String[] args) {
    ArrayBag b1 = new ArrayBag(2);
    ArrayBag b2 = new ArrayBag(4);
    ...
}
```

```
// constructor
public ArrayBag(int maxSize) {
    ... // error-checking
    this.items = new Object[maxSize];
    this.numItems = 0;
}
```

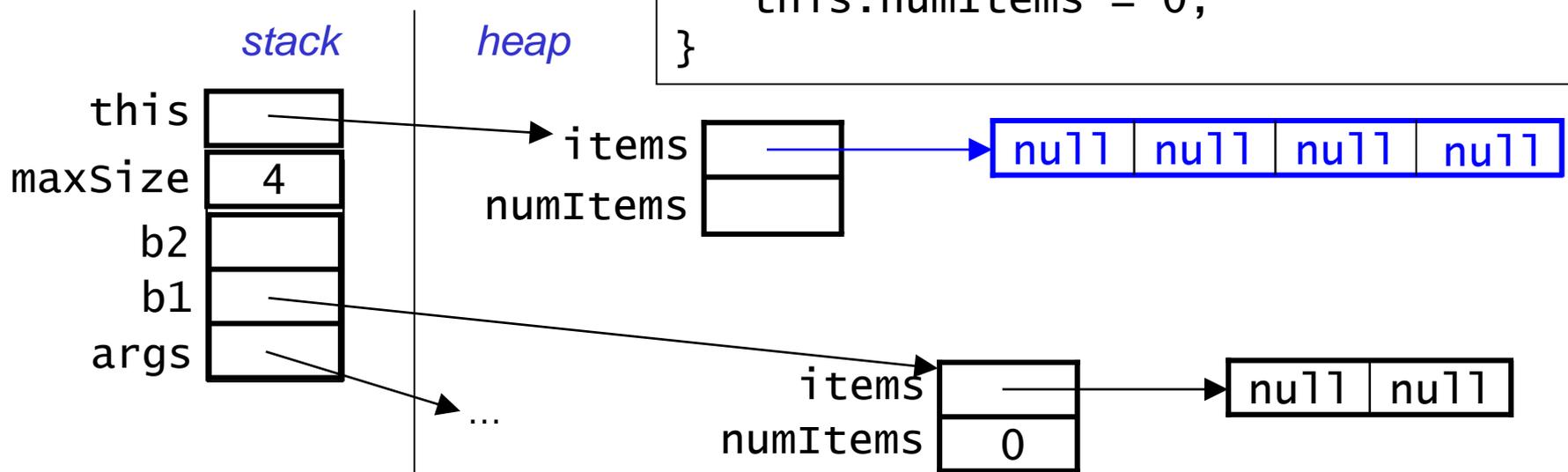


Note: We are not showing the `return address` that is also in this stack frame.

Example: Creating Two ArrayBag Objects

```
// client
public static void main(String[] args) {
    ArrayBag b1 = new ArrayBag(2);
    ArrayBag b2 = new ArrayBag(4);
    ...
}
```

```
// constructor
public ArrayBag(int maxSize) {
    ... // error-checking
    this.items = new Object[maxSize];
    this.numItems = 0;
}
```

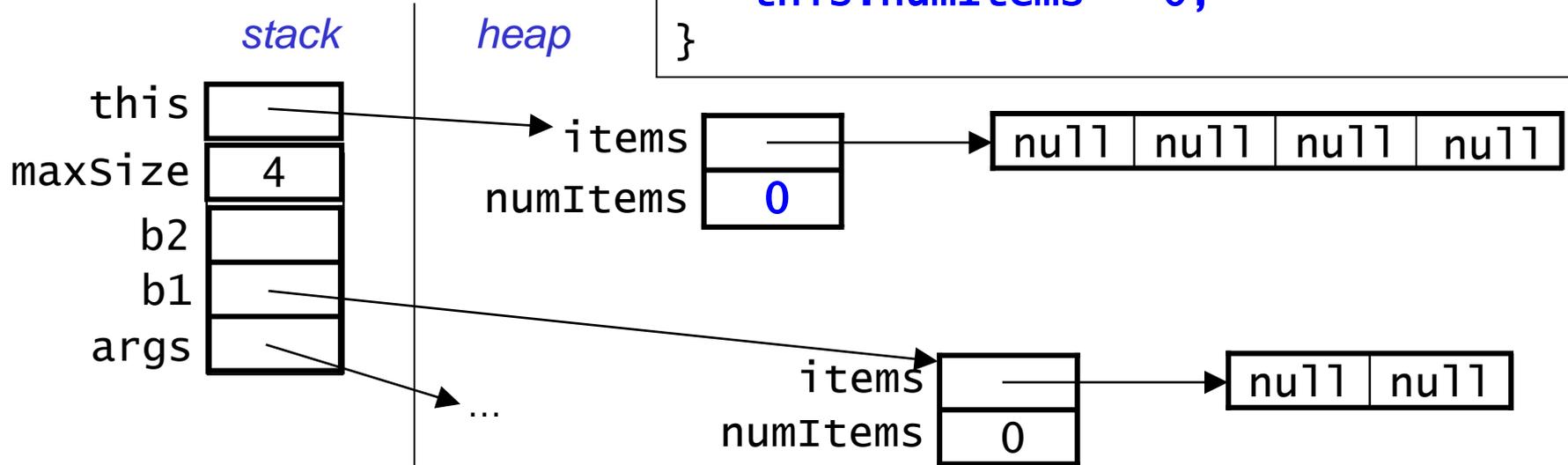


Note: We are not showing the `return address` that is also in this stack frame.

Example: Creating Two ArrayBag Objects

```
// client
public static void main(String[] args) {
    ArrayBag b1 = new ArrayBag(2);
    ArrayBag b2 = new ArrayBag(4);
    ...
}
```

```
// constructor
public ArrayBag(int maxSize) {
    ... // error-checking
    this.items = new Object[maxSize];
    this.numItems = 0;
}
```

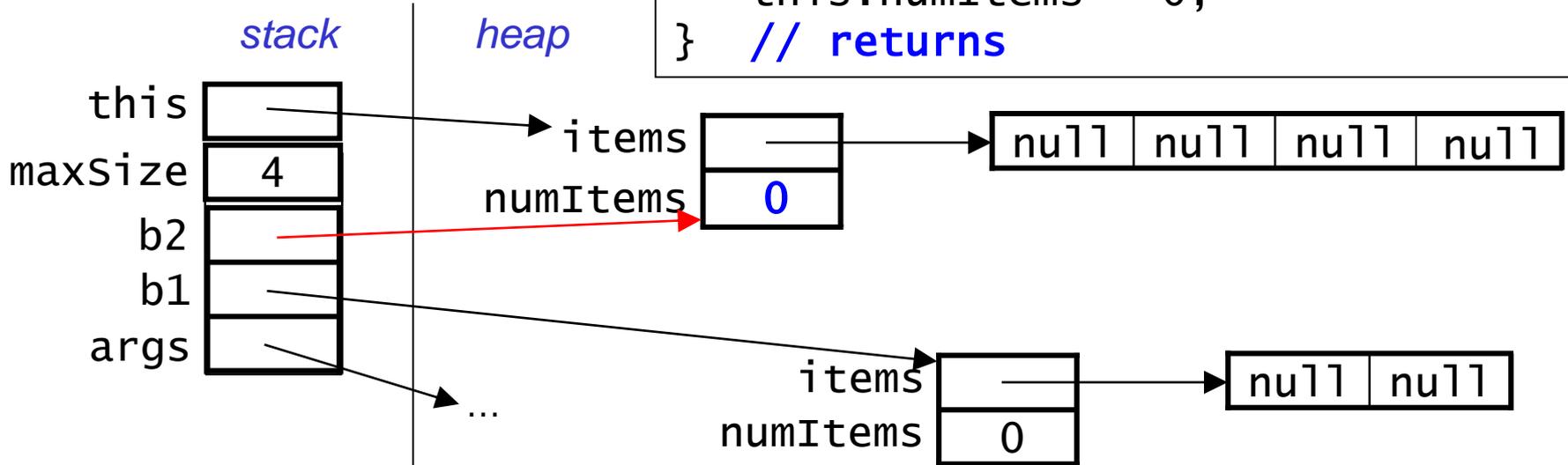


Note: We are not showing the `return address` that is also in this stack frame.

Example: Creating Two ArrayBag Objects

```
// client
public static void main(String[] args) {
    ArrayBag b1 = new ArrayBag(2);
    ArrayBag b2 = new ArrayBag(4);
    ...
}
```

```
// constructor
public ArrayBag(int maxSize) {
    ... // error-checking
    this.items = new Object[maxSize];
    this.numItems = 0;
} // returns
```

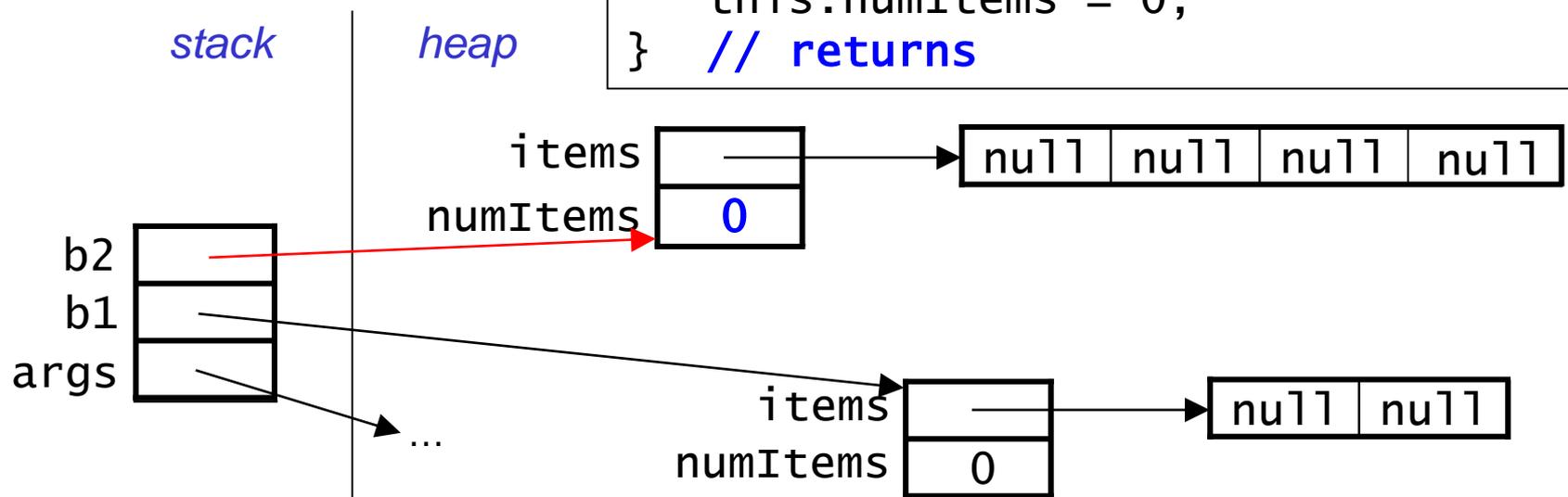


Note: We are not showing the `return` address that is also in this stack frame.

Example: Creating Two ArrayBag Objects

```
// client
public static void main(String[] args) {
    ArrayBag b1 = new ArrayBag(2);
    ArrayBag b2 = new ArrayBag(4);
    ...
}
```

```
// constructor
public ArrayBag(int maxSize) {
    ... // error-checking
    this.items = new Object[maxSize];
    this.numItems = 0;
} // returns
```

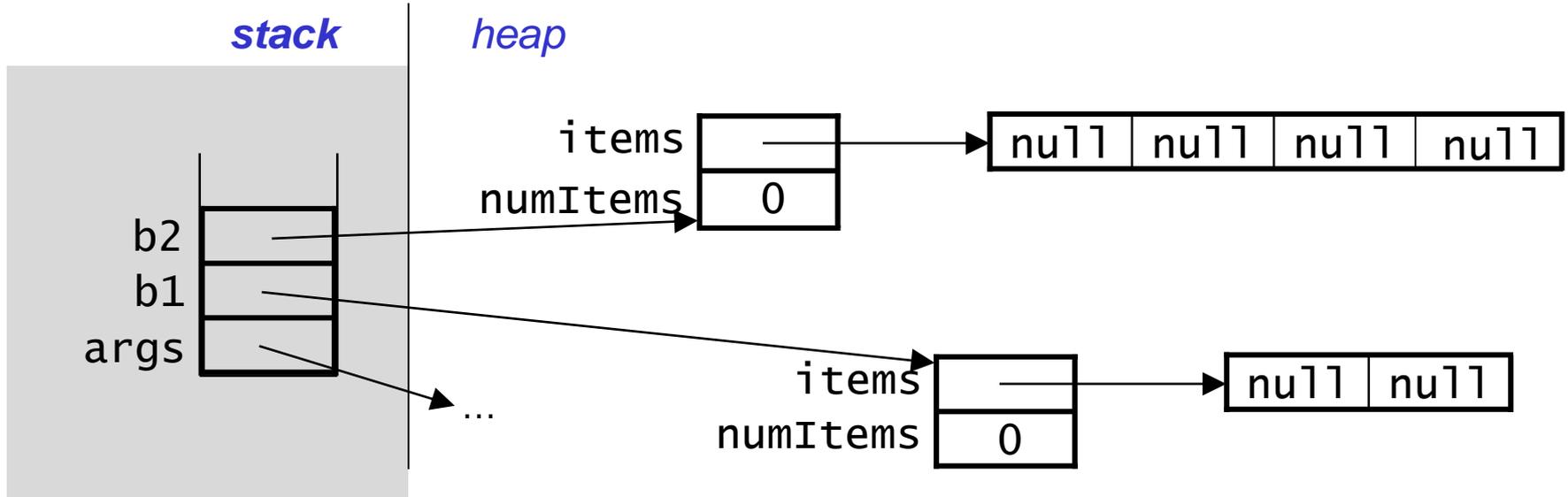


Note: We are not showing the `return` address that is also in this stack frame.

Example: Creating Two ArrayBag Objects

```
// client
public static void main(String[] args) {
    ArrayBag b1 = new ArrayBag(2);
    ArrayBag b2 = new ArrayBag(4);
    ...
}
```

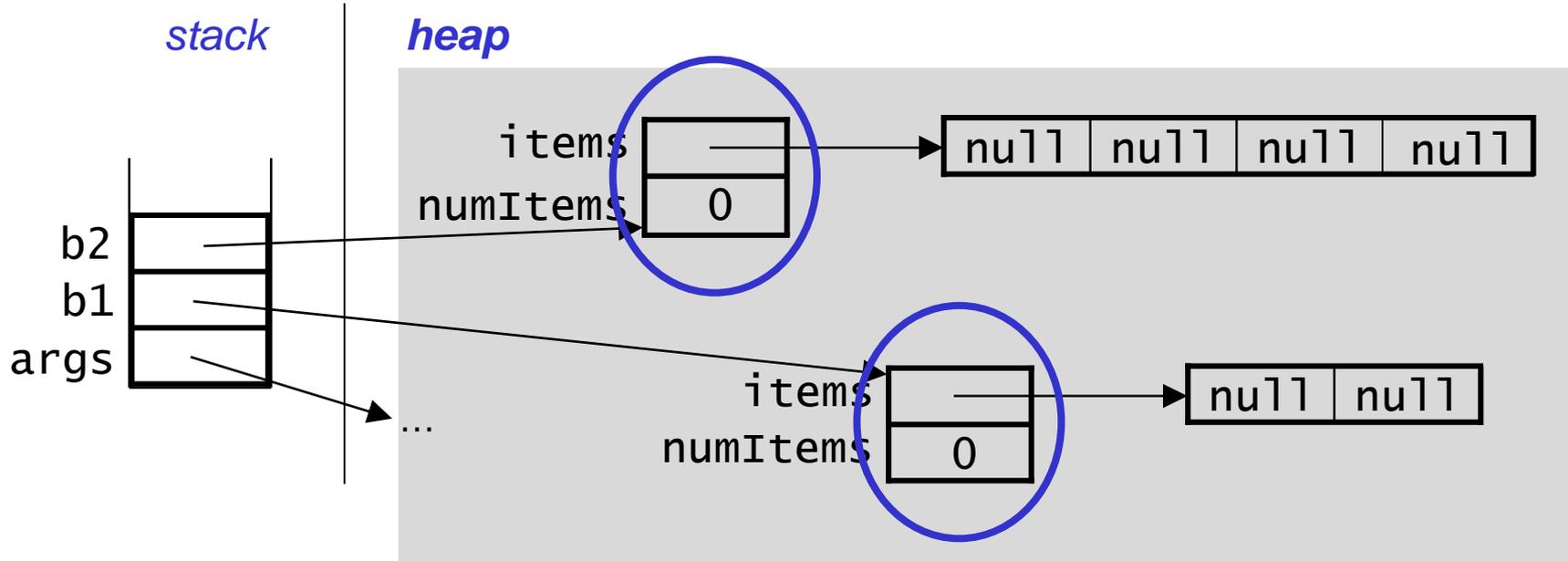
- After the objects have been created, here's what we have:



Example: Creating Two ArrayBag Objects

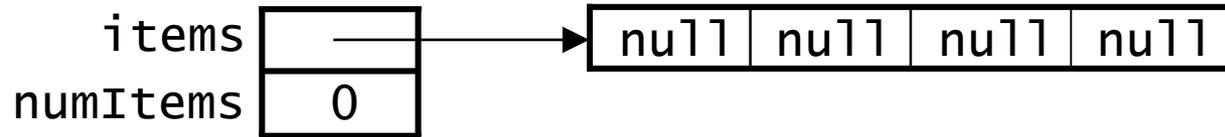
```
// client
public static void main(String[] args) {
    ArrayBag b1 = new ArrayBag(2);
    ArrayBag b2 = new ArrayBag(4);
    ...
}
```

- After the objects have been created, here's what we have:

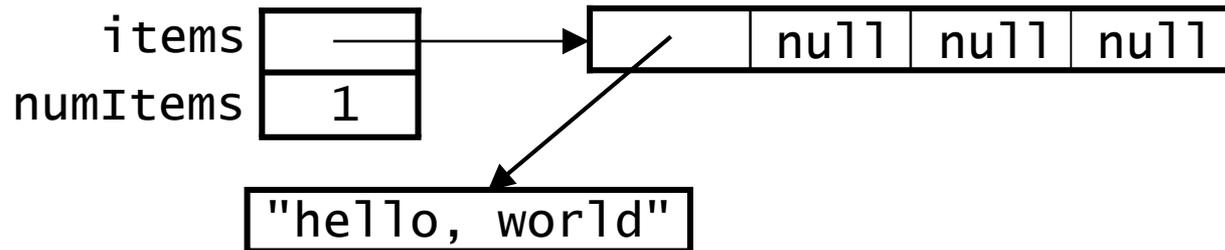


Adding Items

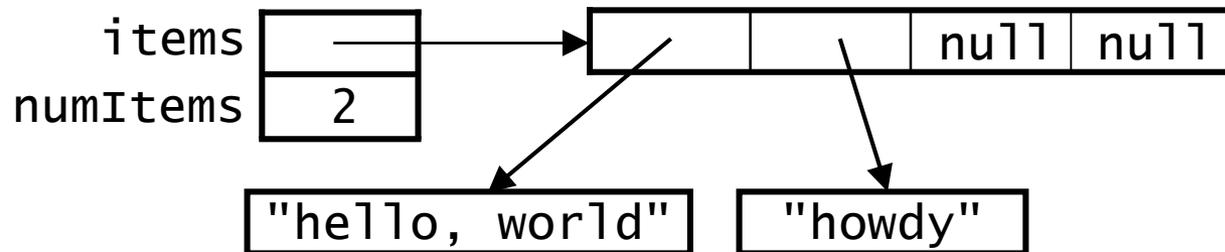
- We fill the array from left to right. Here's an empty bag:



- After adding the first item:

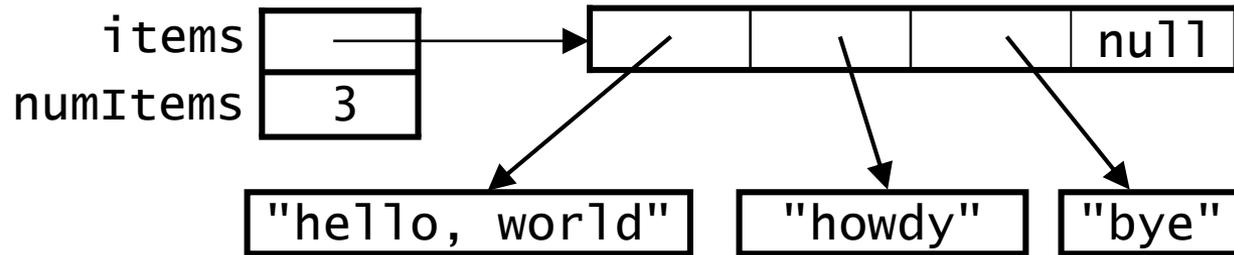


- After adding the second item:



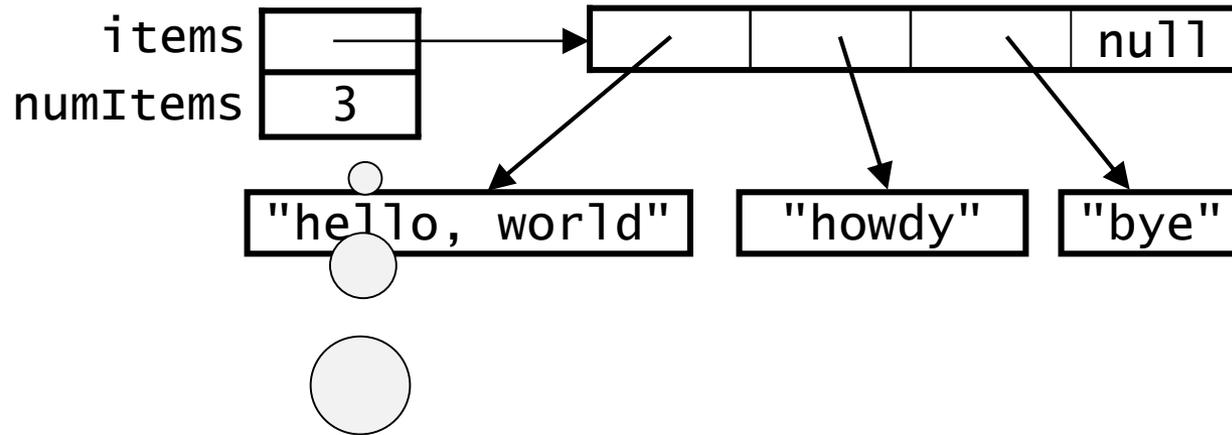
Adding Items (cont.)

- After adding the third item:



Adding Items (cont.)

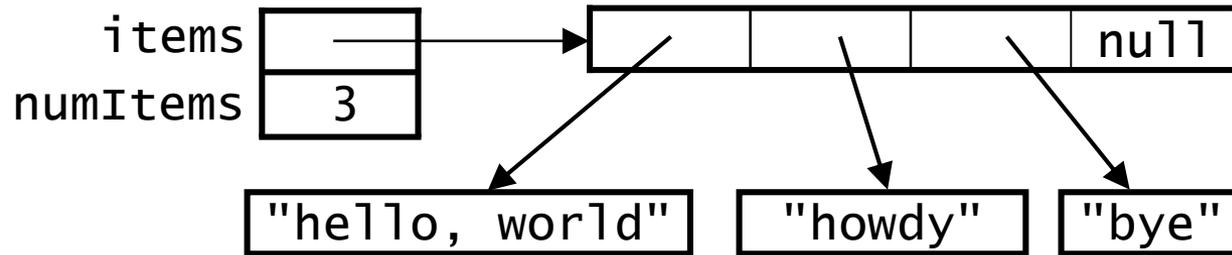
- After adding the third item:



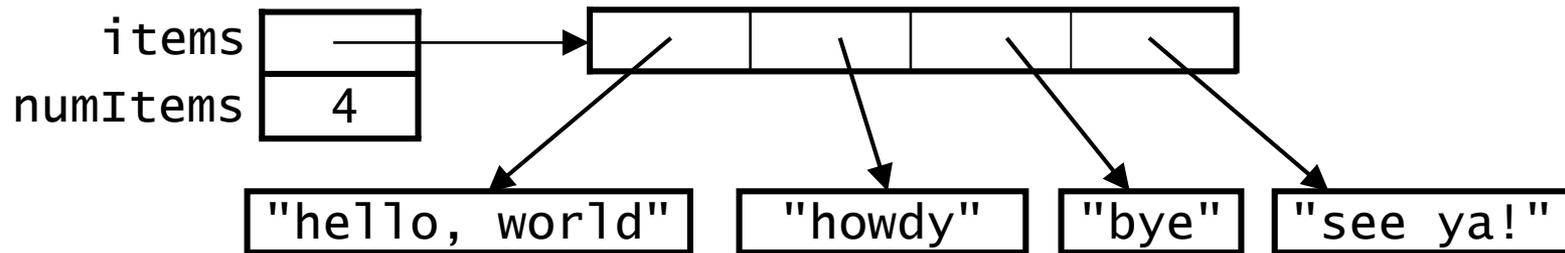
Note the correlation between the *number* of items currently in the bag and the *index* (or offset) we will use to add the next item!

Adding Items (cont.)

- After adding the third item:



- After adding the fourth item:



- At this point, the ArrayBag is full!
 - We would have to "grow" the array, but for our purposes
 - additional items cannot be added until one is removed

Adding Items (cont.)

- After Assume an array of primitive type `int` is referenced by a variable `items`, the basic steps to grow the array are:

1. Create a new array of a larger size. Example:

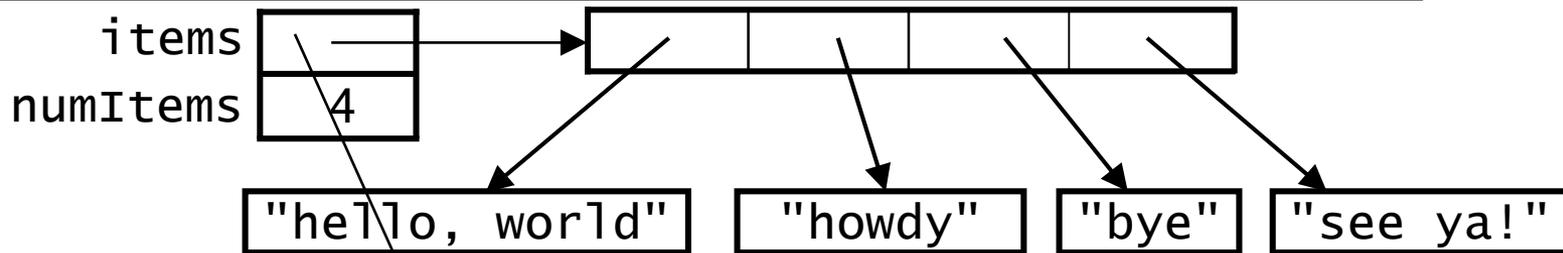
```
int[] tmp = new int[items.length * 2];
```

2. Loop through the array referenced by `items` and assign each element to the array referenced by `tmp`.

3. Reassign the reference (i.e. `items`) to the new array (i.e. `tmp`).

Example: `items = tmp;`

- After



- At this point, the `ArrayBag` is full!
 - We would have to "grow" the array, but for our purposes additional items cannot be added until one is removed

A Method for Adding an Item to a Bag

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

```
    ...
```

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

- takes an object of any type!
- returns a boolean to indicate whether the operation succeeded

```
    }
```

```
    ...
```

```
}
```

A Method for Adding an Item to a Bag

```
public class ArrayBag {
    private Object[] items;
    private int numItems;
    ...
    public boolean add(Object item) {
        boolean item_added = false;    // init return variable

        if (item == null) {
            throw new IllegalArgumentException("no nulls");
        } else if (this.numItems < this.items.length) {
            this.items[this.numItems] = item;
            this.numItems++;
            item_added = true;    // successfully added an item
        }
        return(item_added);
    }
    ...
}
```

- takes an object of any type!
- returns a boolean to indicate whether the operation succeeded

A Method for Adding an Item to a Bag

```
public class ArrayBag {
    private Object[] items;
    private int numItems;
    ...
    public boolean add(Object item) {
        boolean item_added = false;    // init return variable

        if (item == null) {
            throw new IllegalArgumentException("no nulls");
        } else if (this.numItems < this.items.length) {
            this.items[this.numItems] = item;
            this.numItems++;
            item_added = true;    // successfully added an item
        }
        return(item_added);
    }
    ...
}
```

- takes an object of any type!
- returns a boolean to indicate whether the operation succeeded

A Method for Adding an Item to a Bag

```
public class ArrayBag {
    private Object[] items;
    private int numItems;
    ...
    public boolean add(Object item) {
        boolean item_added = false;    // init return variable

        if (item == null) {
            throw new IllegalArgumentException("no nulls");
        } else if (this.numItems < this.items.length) {
            this.items[this.numItems] = item;
            this.numItems++;
            item_added = true;
        }
        return(item_added);
    }
    ...
}
```

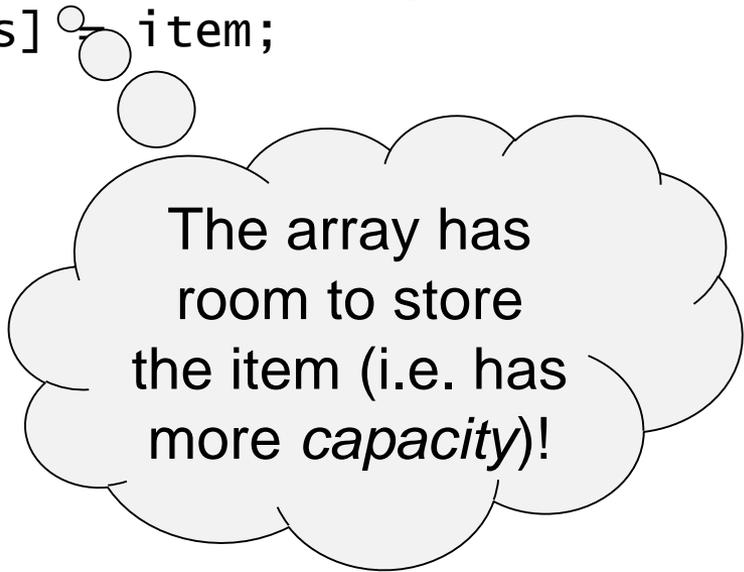
- takes an object of any type!
- returns a boolean to indicate whether the operation succeeded

A Method for Adding an Item to a Bag

```
public class ArrayBag {
    private Object[] items;
    private int numItems;
    ...
    public boolean add(Object item) {
        boolean item_added = false;    // init return variable

        if (item == null) {
            throw new IllegalArgumentException("no nulls");
        } else if (this.numItems < this.items.length) {
            this.items[this.numItems] = item;
            this.numItems++;
            item_added = true;
        }
        return(item_added);
    }
    ...
}
```

- takes an object of any type!
- returns a boolean to indicate whether the operation succeeded



The array has room to store the item (i.e. has more *capacity*)!

A Method for Adding an Item to a Bag

```
public class ArrayBag {
    private Object[] items;
    private int numItems;
    ...
    public boolean add(Object item) {
        boolean item_added = false;    // init return variable

        if (item == null) {
            throw new IllegalArgumentException("no nulls");
        } else if (this.numItems < this.items.length) {
            this.items[this.numItems] = item;
            this.numItems++;
            item_added = true;
        }
        return(item_added);
    }
    ...
}
```

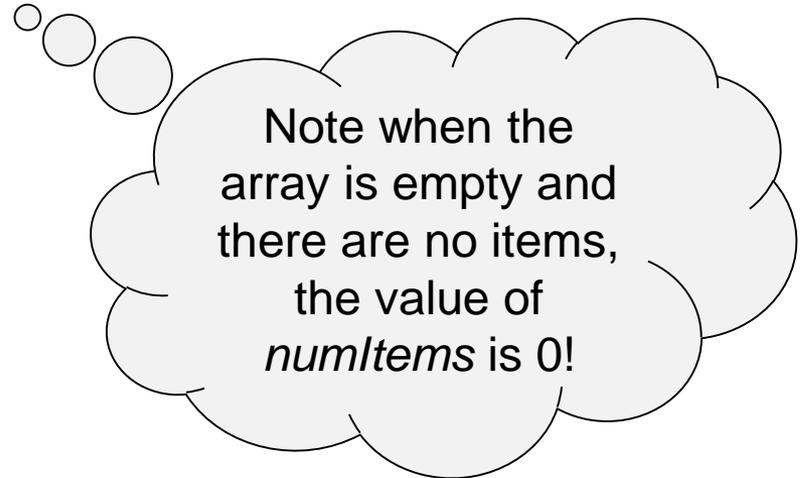
- takes an object of any type!
- returns a boolean to indicate whether the operation succeeded

A Method for Adding an Item to a Bag

```
public class ArrayBag {
    private Object[] items;
    private int numItems;
    ...
    public boolean add(Object item) {
        boolean item_added = false;    // init return variable

        if (item == null) {
            throw new IllegalArgumentException("no nulls");
        } else if (this.numItems < this.items.length) {
            this.items[this.numItems] = item;
            this.numItems++;
            item_added = true;
        }
        return(item_added);
    }
    ...
}
```

- takes an object of any type!
- returns a boolean to indicate whether the operation succeeded



A Method for Adding an Item to a Bag

```
public class ArrayBag {
    private Object[] items;
    private int numItems;
    ...
    public boolean add(Object item) {
        boolean item_added = false;    // init return variable

        if (item == null) {
            throw new IllegalArgumentException("no nulls");
        } else if (this.numItems < this.items.length) {
            this.items[this.numItems] = item;
            this.numItems++;
            item_added = true;
        }
        return(item_added);
    }
    ...
}
```

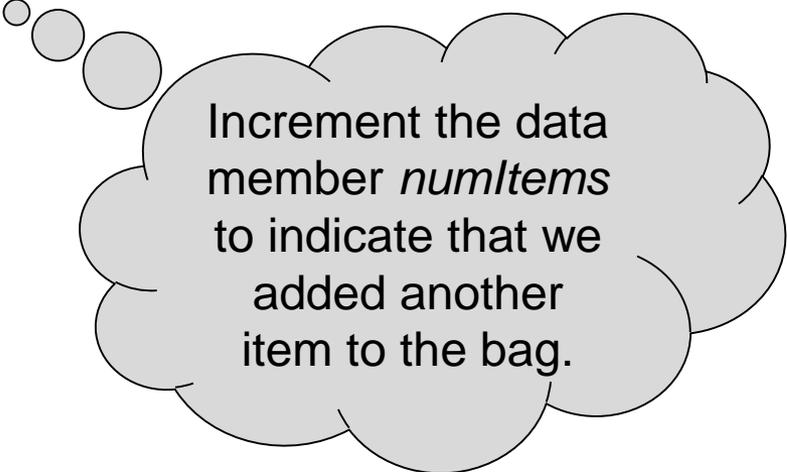
- takes an object of any type!
- returns a boolean to indicate whether the operation succeeded

A Method for Adding an Item to a Bag

```
public class ArrayBag {
    private Object[] items;
    private int numItems;
    ...
    public boolean add(Object item) {
        boolean item_added = false;    // init return variable

        if (item == null) {
            throw new IllegalArgumentException("no nulls");
        } else if (this.numItems < this.items.length) {
            this.items[this.numItems] = item;
            this.numItems++;
            item_added = true;
        }
        return(item_added);
    }
    ...
}
```

- takes an object of any type!
- returns a boolean to indicate whether the operation succeeded



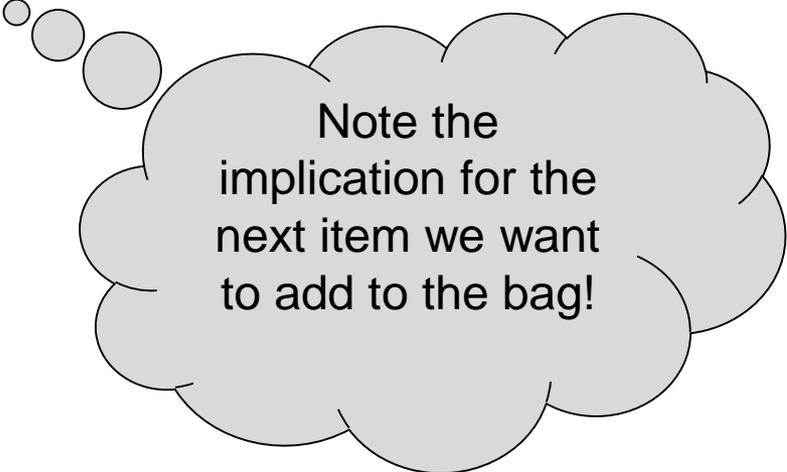
Increment the data member *numItems* to indicate that we added another item to the bag.

A Method for Adding an Item to a Bag

```
public class ArrayBag {
    private Object[] items;
    private int numItems;
    ...
    public boolean add(Object item) {
        boolean item_added = false;    // init return variable

        if (item == null) {
            throw new IllegalArgumentException("no nulls");
        } else if (this.numItems < this.items.length) {
            this.items[this.numItems] = item;
            this.numItems++;
            item_added = true;
        }
        return(item_added);
    }
    ...
}
```

- takes an object of any type!
- returns a boolean to indicate whether the operation succeeded



Note the implication for the next item we want to add to the bag!

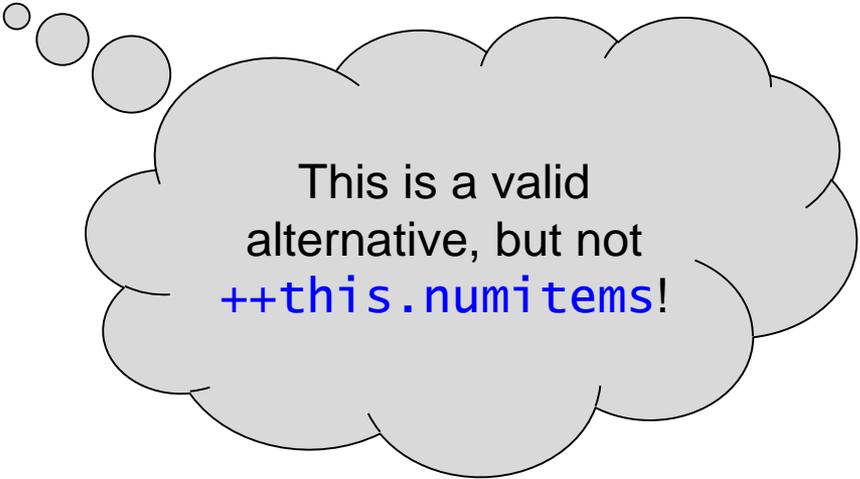
A Method for Adding an Item to a Bag

```
public class ArrayBag {
    private Object[] items;
    private int numItems;
    ...
    public boolean add(Object item) {
        boolean item_added = false;    // init return variable

        if (item == null) {
            throw new IllegalArgumentException("no nulls");
        } else if (this.numItems < this.items.length) {
            this.items[this.numItems++] = item;

            item_added = true;
        }
        return(item_added);
    }
    ...
}
```

- takes an object of any type!
- returns a boolean to indicate whether the operation succeeded



This is a valid alternative, but not `++this.numItems!`

A Method for Adding an Item to a Bag

```
public class ArrayBag {
    private Object[] items;
    private int numItems;
    ...
    public boolean add(Object item) {
        boolean item_added = false;    // init return variable

        if (item == null) {
            throw new IllegalArgumentException("no nulls");
        } else if (this.numItems < this.items.length) {
            this.items[this.numItems] = item;
            this.numItems++;
            item_added = true;          // indicate success
        }
        return(item_added);
    }
    ...
}
```

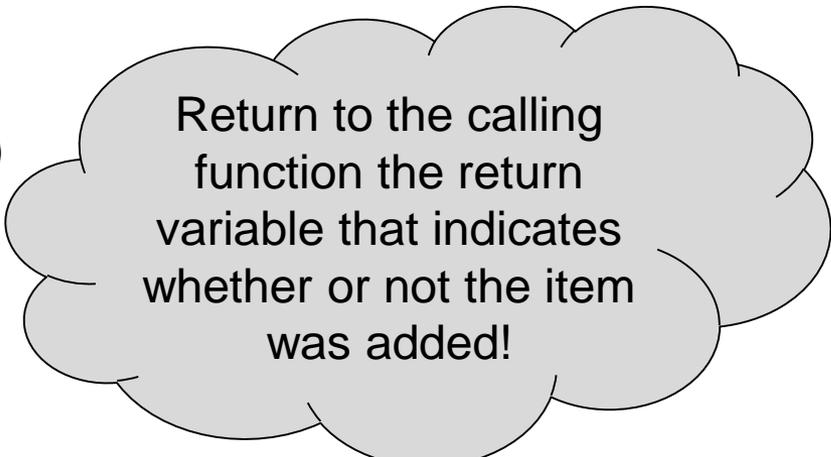
- takes an object of any type!
- returns a boolean to indicate whether the operation succeeded

A Method for Adding an Item to a Bag

```
public class ArrayBag {
    private Object[] items;
    private int numItems;
    ...
    public boolean add(Object item) {
        boolean item_added = false;    // init return variable

        if (item == null) {
            throw new IllegalArgumentException("no nulls");
        } else if (this.numItems < this.items.length) {
            this.items[this.numItems] = item;
            this.numItems++;
            item_added = true;          // indicate success
        }
        return(item_added);
    }
    ...
}
```

- takes an object of any type!
- returns a boolean to indicate whether the operation succeeded

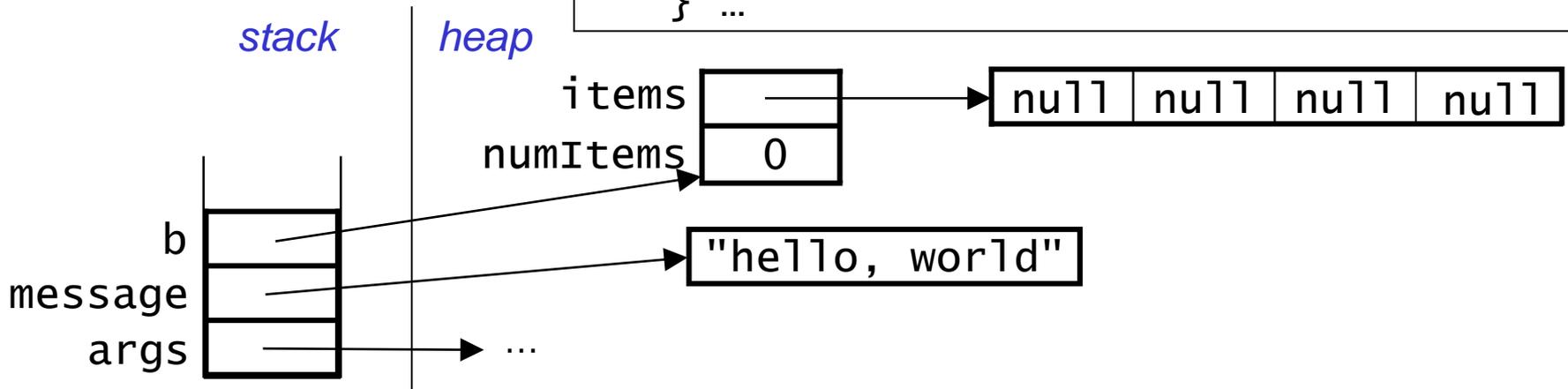


Return to the calling function the return variable that indicates whether or not the item was added!

Example: Adding an Item

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

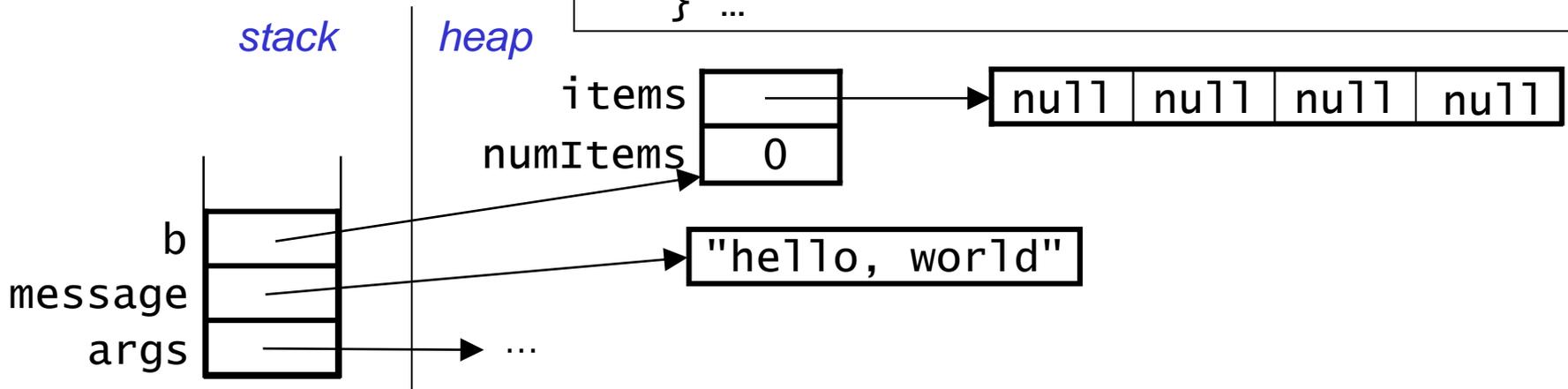
```
public boolean add(Object item) {  
    ...  
    else {  
        this.items[this.numItems] = item;  
        this.numItems++;  
        item_added = true;  
    } ...  
}
```



Example: Adding an Item

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

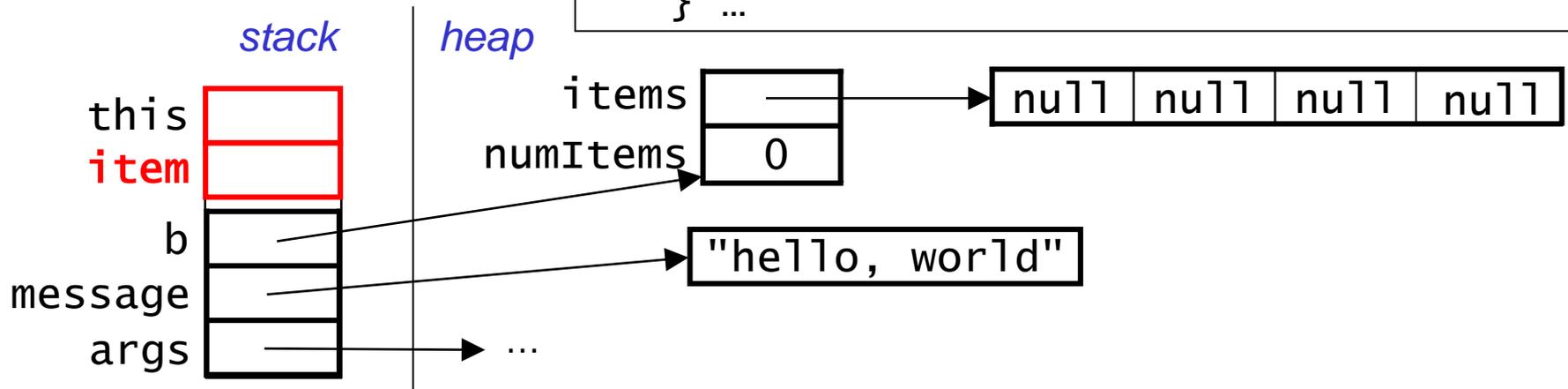
```
public boolean add(Object item) {  
    ...  
    else {  
        this.items[this.numItems] = item;  
        this.numItems++;  
        item_added = true;  
    } ...  
}
```



Example: Adding an Item (cont.)

```
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++;  
        item_added = true;  
    } ...  
}
```



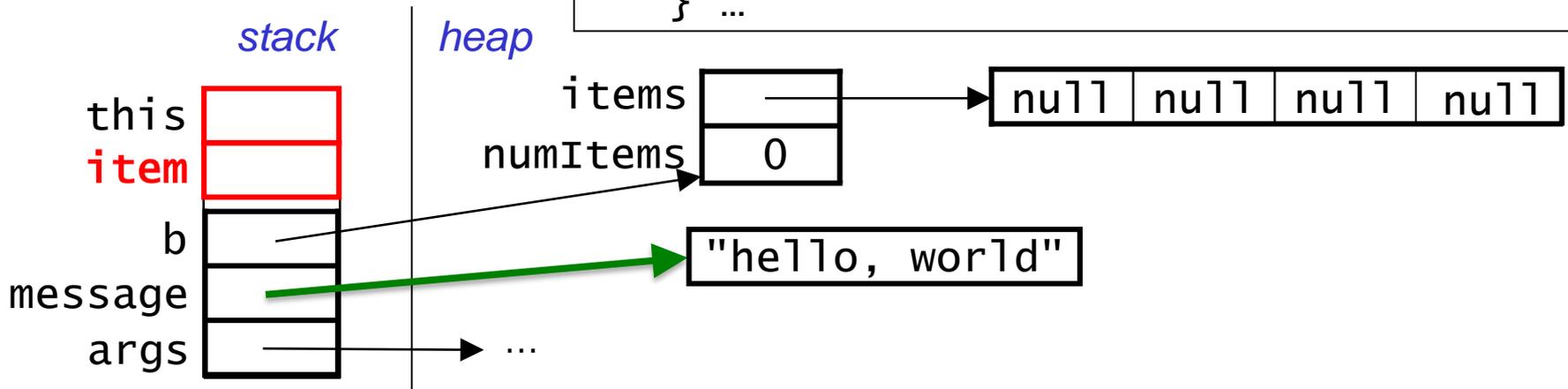
- `add`'s stack frame

Note: We are not showing the `return address` or local variable `item_added`.

Example: Adding an Item (cont.)

```
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++;  
        item_added = true;  
    } ...  
}
```

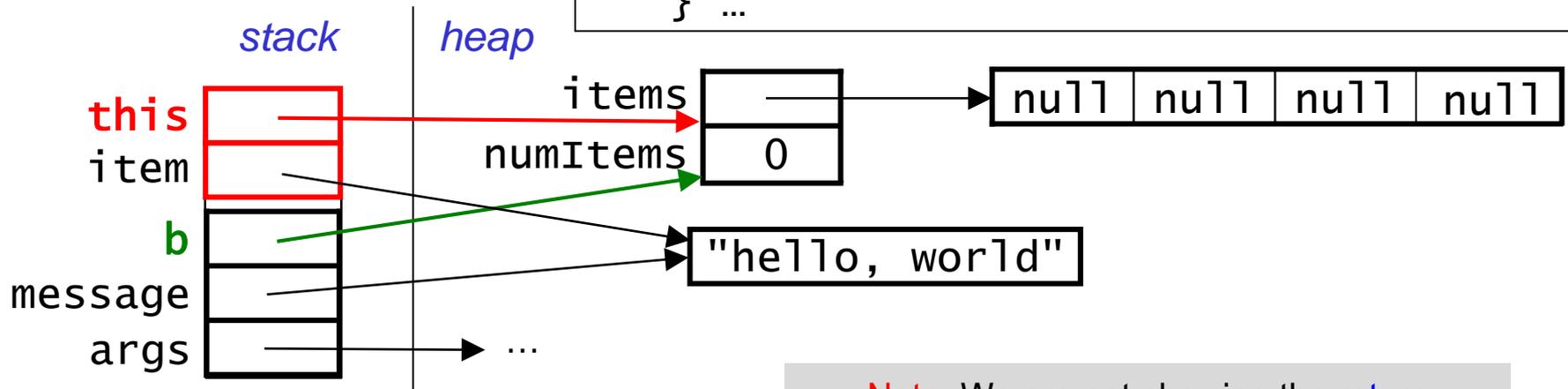


Note: We are not showing the [return address](#) or local variable `item_added`.

Example: Adding an Item (cont.)

```
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++;  
        item_added = true;  
    } ...  
}
```



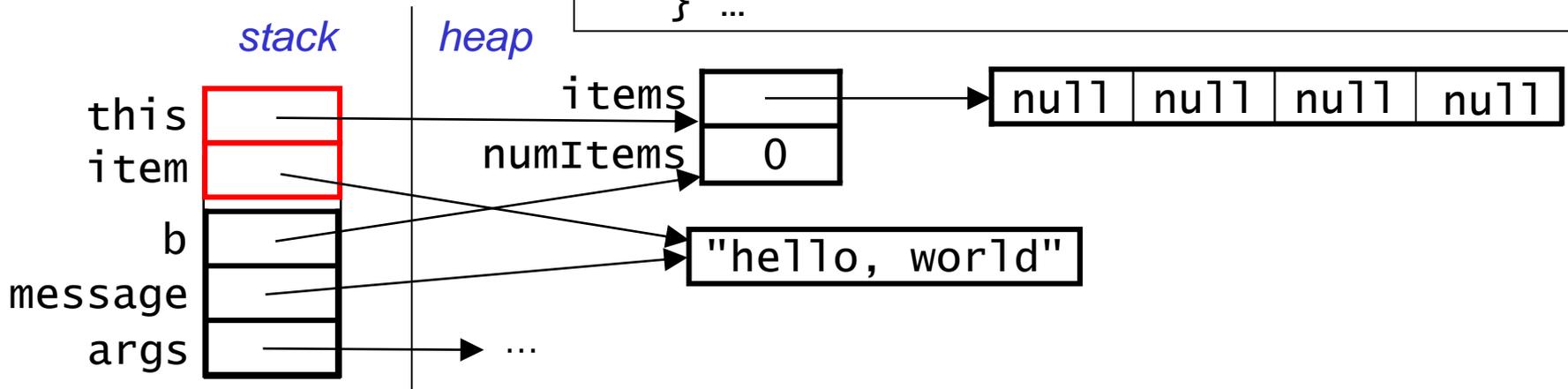
Note: We are not showing the `return address` or local variable `item_added`.

- `add`'s stack frame includes:
 - `item`, which stores a copy of the reference passed as a param.
 - `this`, which stores a reference to the called `ArrayBag` object

Example: Adding an Item (cont.)

```
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++;  
        item_added = true;  
    } ...  
}
```

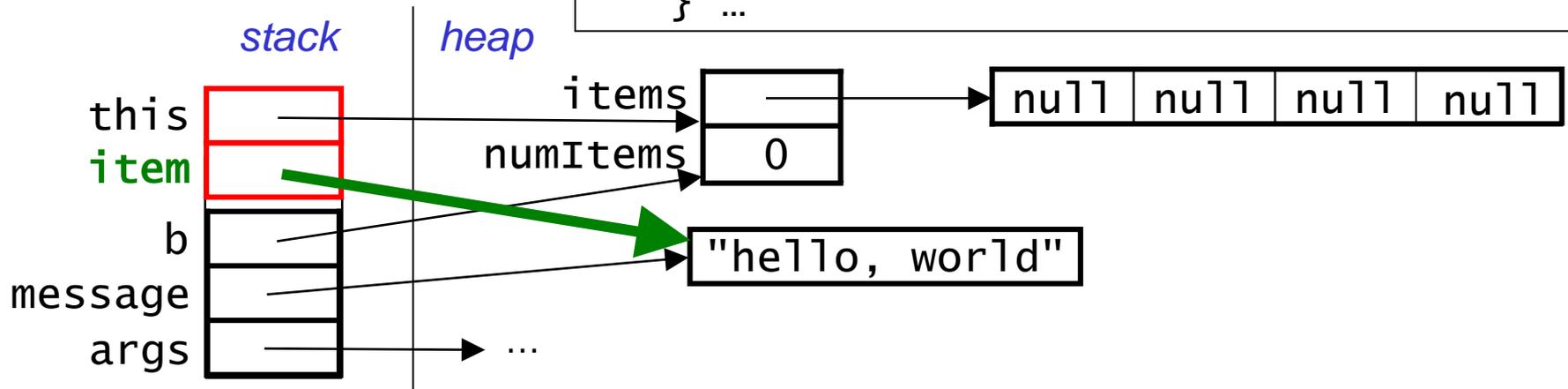


- The method modifies the `items` array and `numItems`.

Example: Adding an Item (cont.)

```
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++;  
        item_added = true;  
    } ...  
}
```

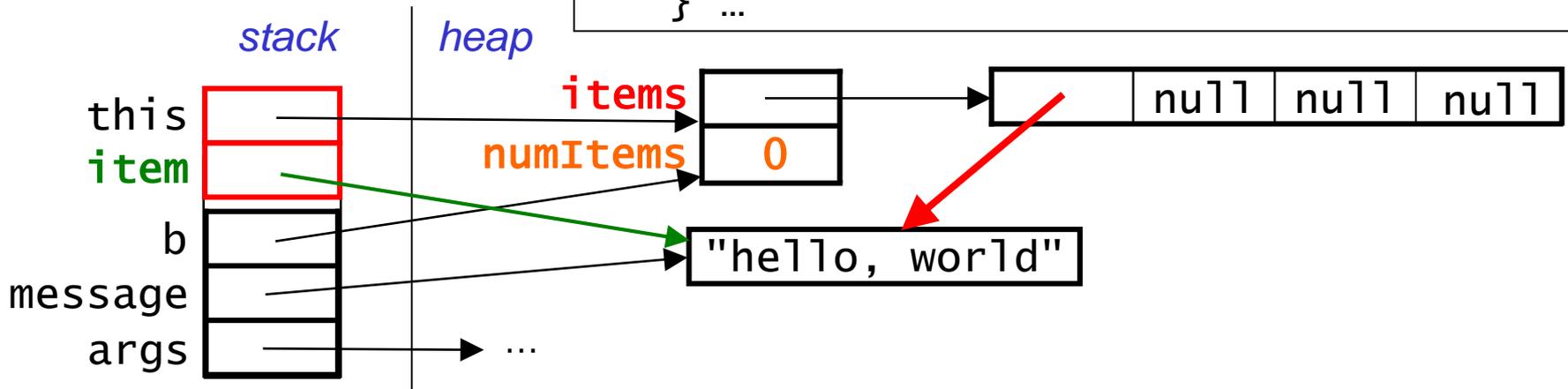


- The method modifies the `items` array and `numItems`.

Example: Adding an Item (cont.)

```
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++;  
        item_added = true;  
    } ...  
}
```

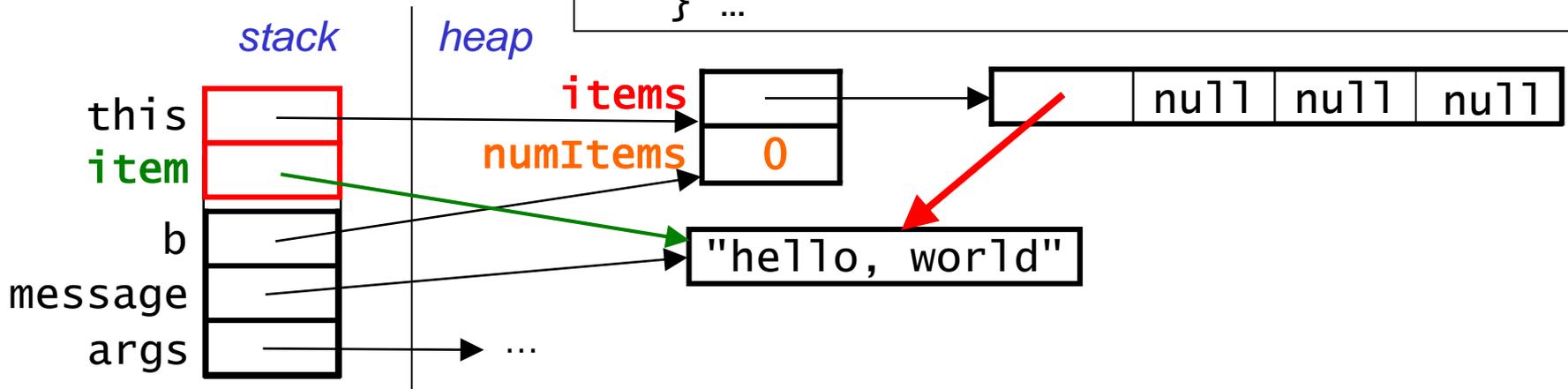


- The method modifies the `items` array and `numItems`.

Example: Adding an Item (cont.)

```
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++;  
        item_added = true;  
    } ...  
}
```

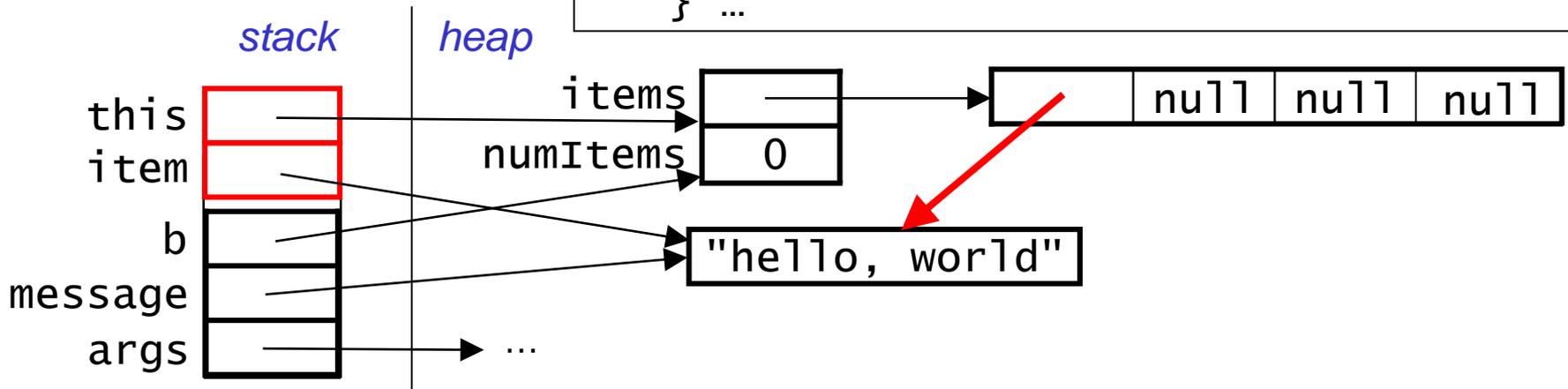


- The method modifies the `items` array and `numItems`.
 - note that the array stores a *copy of the reference* to the item, not a copy of the item itself.

Example: Adding an Item (cont.)

```
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++;  
        item_added = true;  
    } ...  
}
```

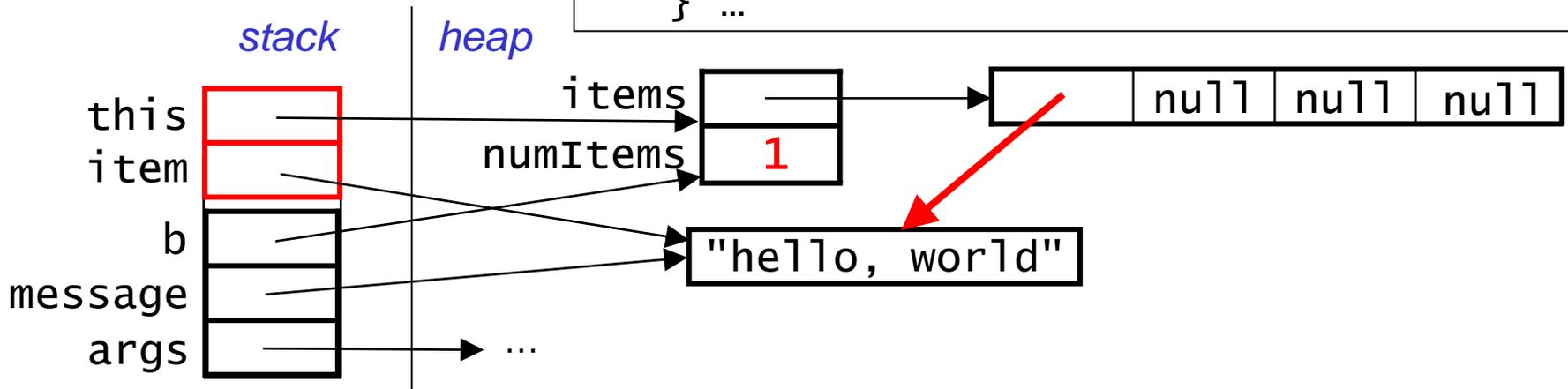


- The method modifies the `items` array and `numItems`.
 - note that the array stores a copy of the *reference* to the item, not a copy of the item itself.

Example: Adding an Item (cont.)

```
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++;  
        item_added = true;  
    } ...  
}
```

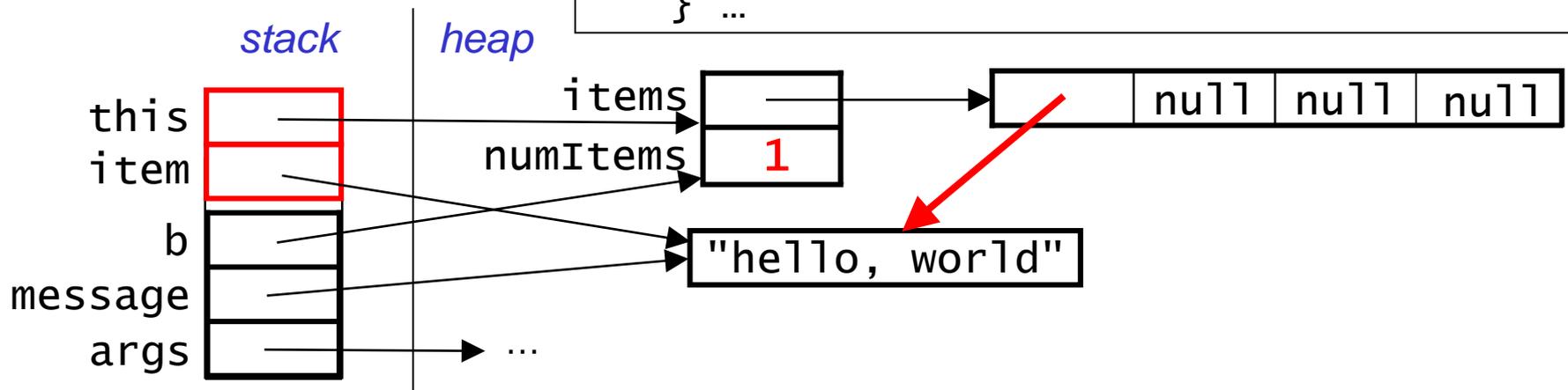


- The method modifies the `items` array and `numItems`.
 - note that the array stores a copy of the *reference* to the item, not a copy of the item itself.

Example: Adding an Item (cont.)

```
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++;  
        item_added = true;  
    } ...  
}
```

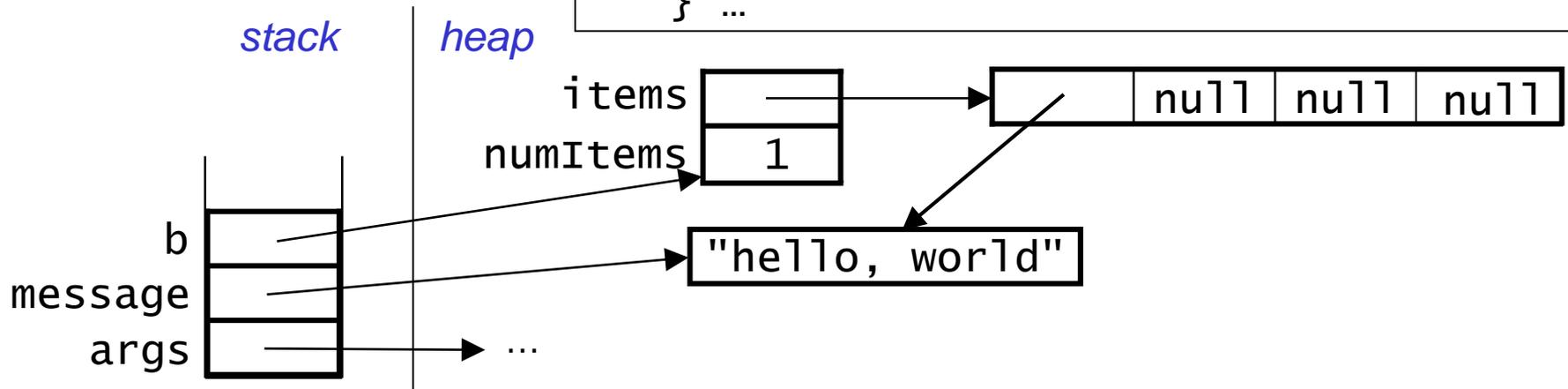


- The method modifies the `items` array and `numItems`.
 - note that the array stores a copy of the *reference* to the item, not a copy of the item itself.

Example: Adding an Item (cont.)

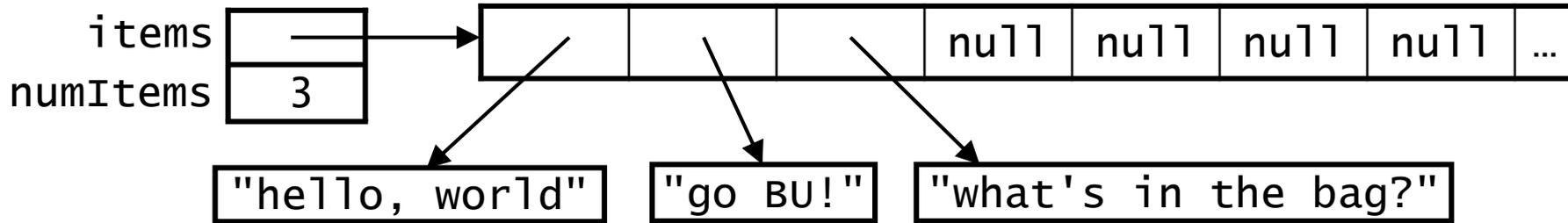
```
public static void main(String[] args) {  
    String message = "hello, world";  
    ArrayBag b = new ArrayBag(4);  
    b.add(message);  
    ...  
}
```

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



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

Extra Practice: Determining if a Bag Contains an Item

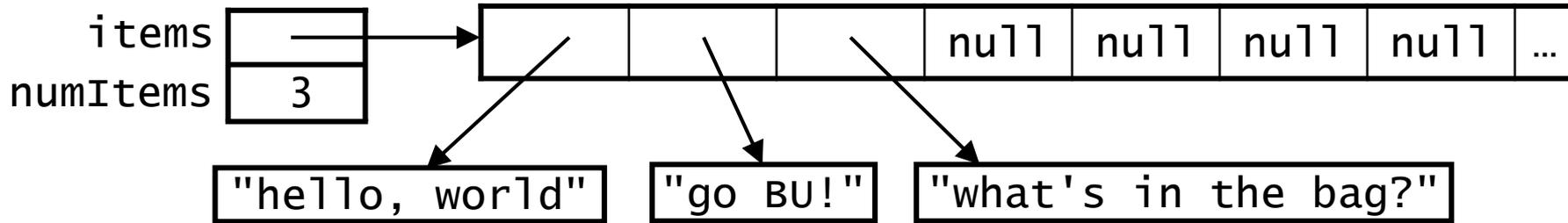


- Let's write the `ArrayBag` `contains()` method together.
 - should return `true` if an object equal to `item` is found, and `false` otherwise.

```
_____ contains(_____ item) {
```

```
}
```

Extra Practice: Determining if a Bag Contains an Item

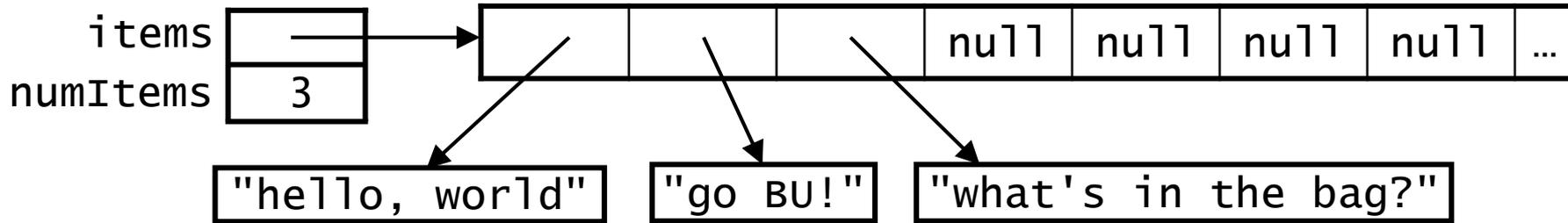


- Let's write the `ArrayBag contains()` method together.
 - should return `true` if an object equal to `item` is found, and `false` otherwise.

```
public boolean contains(_____ item) {
```

```
}
```

Extra Practice: Determining if a Bag Contains an Item

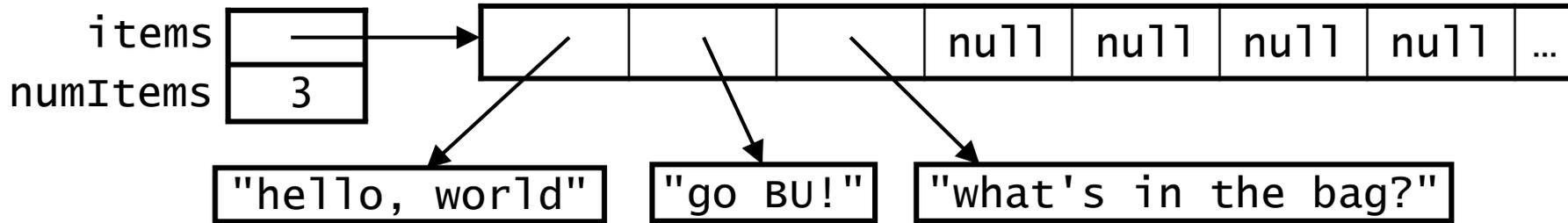


- Let's write the `ArrayBag contains()` method together.
 - should return `true` if an object equal to `item` is found, and `false` otherwise.

```
public boolean contains(Object item) {
```

```
}
```

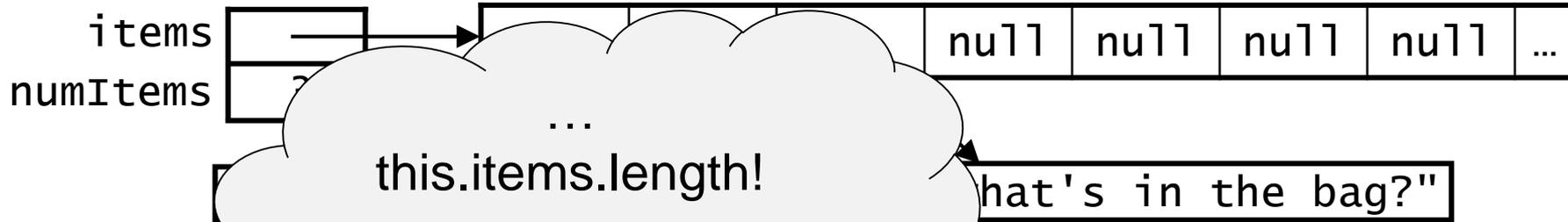
Extra Practice: Determining if a Bag Contains an Item



- Let's write the `ArrayBag contains()` method together.
 - 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.numItems; i++) {
        if (this.items[i].equals(item) { // not ==
            return true;
        }
    }
    return false;
}
```

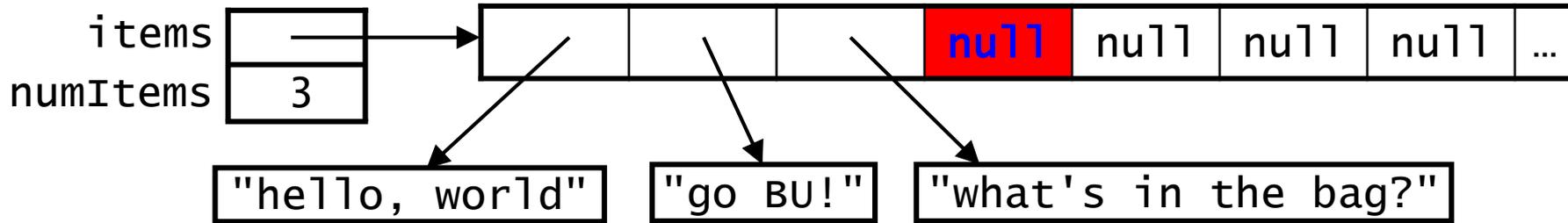

Would this work instead?



- Let's write `contains()` method together.
 - 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) { // not ==  
            return true;  
        }  
    }  
    return false;  
}
```

Would this work instead? *no!*

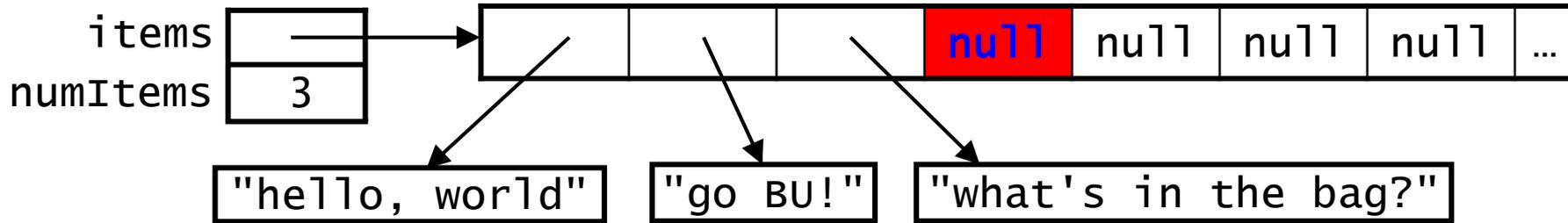


- Let's write the `ArrayBag contains()` method together.
 - should return `true` if an object equal to `item` is found, and `false` otherwise.

```
public boolean contains(Object item) {  
    for (int i = 0; i < this.items.length; i++) {  
        if (this.items[i].equals(item) { // not ==  
            return true;  
        }  
    }  
    return false;  
}
```

- will get a `NullPointerException` from first array element that is still `null`
- even if we check for `null`s, it's more efficient to only look at actual items!

Would this work instead? *no!*

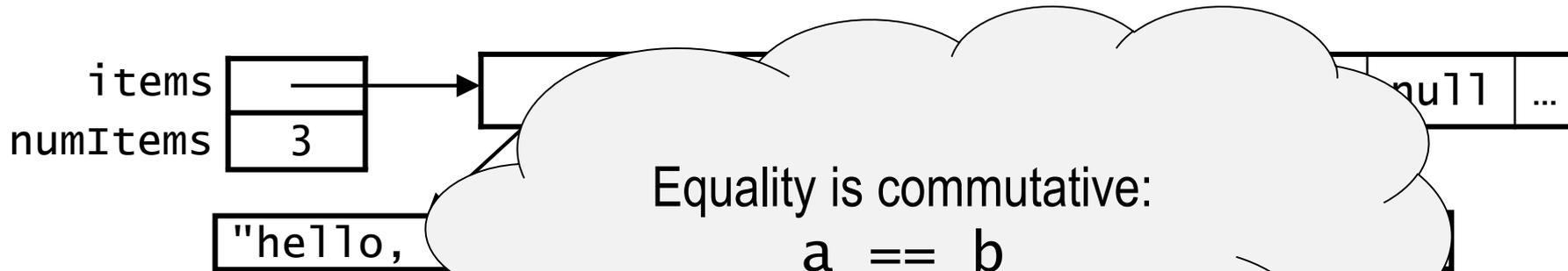


- Let's write the `ArrayBag contains()` method together.
 - should return `true` if an object equal to `item` is found, and `false` otherwise.

```
public boolean contains(Object item) {  
    for (int i = 0; i < this.numItems; i++) {  
        if (this.items[i].equals(item) { // not ==  
            return true;  
        }  
    }  
    return false;  
}
```

- will get a `NullPointerException` from first array element that is still `null`
- even if we check for `null`s, it's more efficient to only look at actual items!

Would this work instead? *no!*

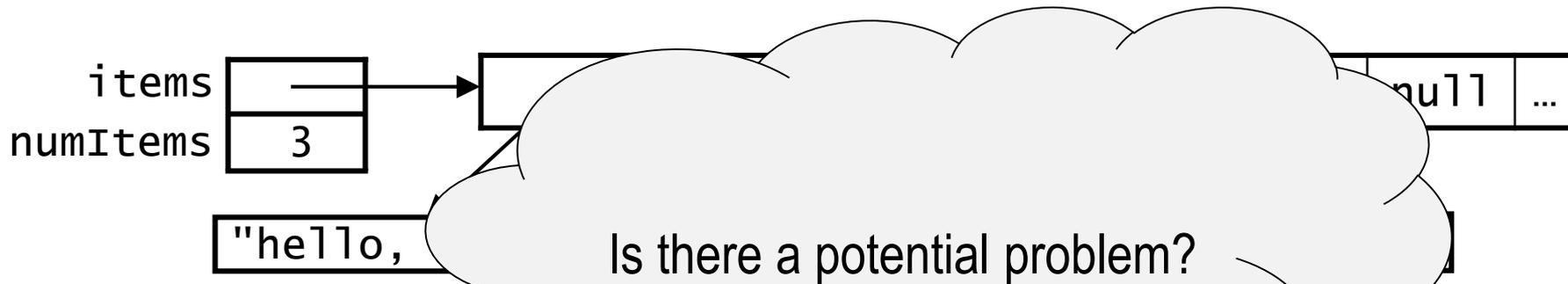


- Let's write the `contains` method.
 - should return true if an object equal to `item` is found, and false otherwise.

```
public boolean contains(Object item) {  
    for (int i = 0; i < this.numItems; i++) {  
        if (this.items[i].equals(item)) { // not ==  
            return true;  
        }  
    }  
    return false;  
}
```

- will get a `NullPointerException` from first array element that is still `null`
- even if we check for `null`s, it's more efficient to only look at actual items!

Would this work instead? *no!*

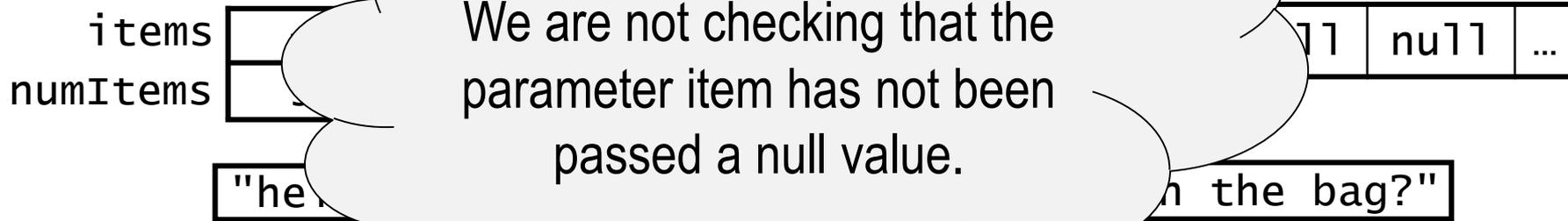


- Let's write the `contains` method.
 - should return `true` if an object equal to `item` is found, and `false` otherwise.

```
public boolean contains(Object item) {  
    for (int i = 0; i < this.numItems; i++) {  
        if (item.equals(items[i]) { // not ==  
            return true;  
        }  
    }  
    return false;  
}
```

- will get a `NullPointerException` from first array element that is still `null`
- even if we check for `null`s, it's more efficient to only look at actual items!

Would this

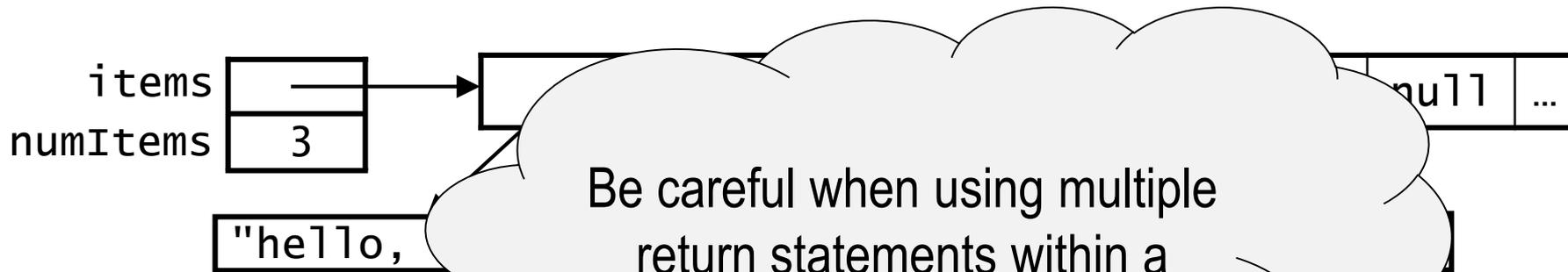


- Let's write the `ArrayBag` `contains()` method together.
 - should return `true` if an object equal to `item` is found, and `false` otherwise.

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

- will get a `NullPointerException` because we are calling a method on a null object,

Would this work instead? *no!*



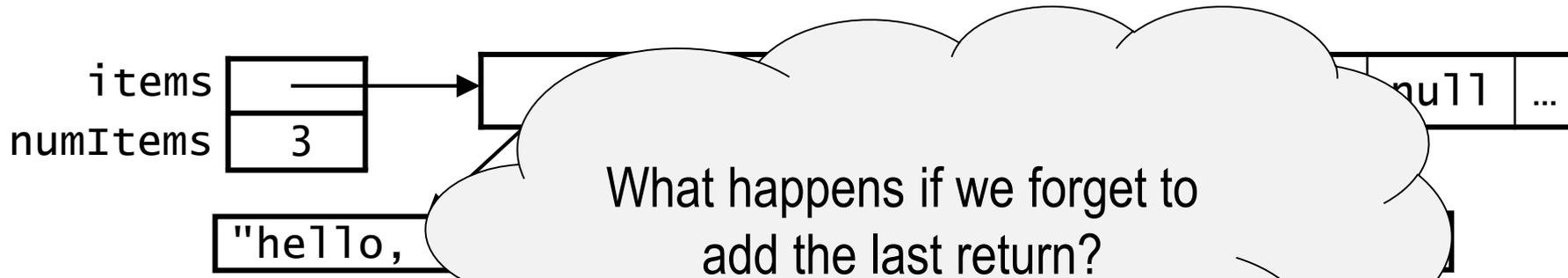
Be careful when using multiple return statements within a function...

- Let's write the Array.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.numItems; i++) {  
        if (this.items[i].equals(item) { // not ==  
            return true;  
        }  
    }  
    return false;  
}
```

- will get a NullPointerException from first array element that is still null
- even if we check for nulls, it's more efficient to only look at actual items!

Would this work instead? *no!*



- Let's write the Array...
- 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.numItems; i++) {  
        if (this.items[i].equals(item) { // not ==  
            return true;  
        }  
    }  
}
```

- will get a `NullPointerException` from first array element that is still `null`
- even if we check for `null`s, it's more efficient to only look at actual items!

Another Incorrect contains() Method

```
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;
}
```

- Why won't this version of the method work in all cases?
- When would it work?

Another Incorrect contains() Method

```
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;
}
```

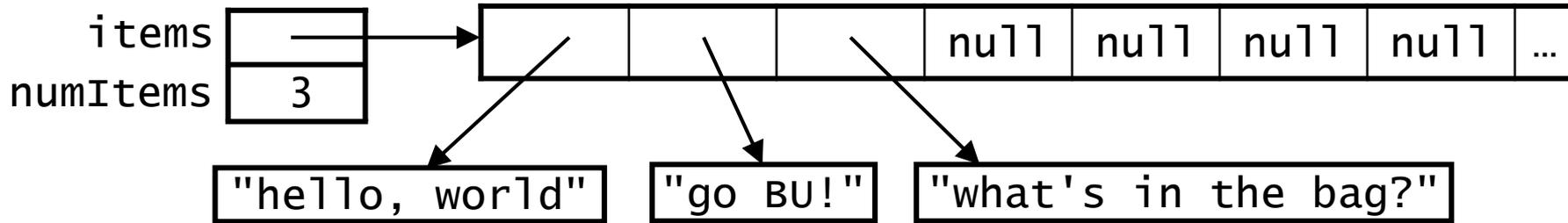
- Why won't this version of the method work in all cases? *When the first item of the array is not the item we are looking for, we return false without looking at the remaining items of the array*
- When would it work?

Another Incorrect contains() Method

```
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;
}
```

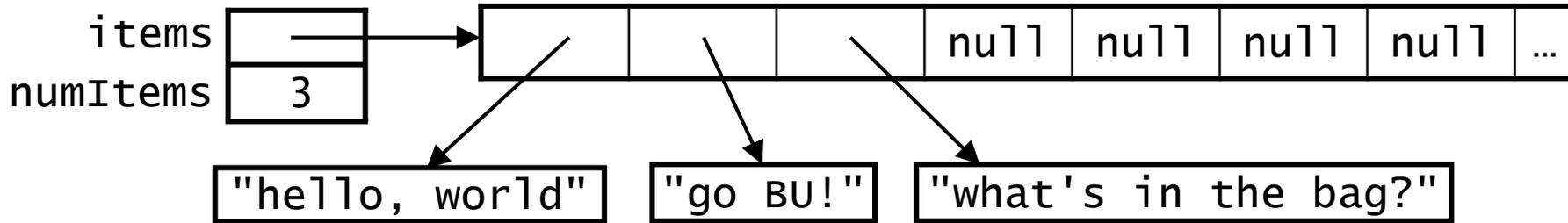
- Why won't this version of the method work in all cases? *When the first item of the array is not the item we are looking for, we return false without looking at the remaining items of the array*
- When would it work? *If the first item in the array is the item we are looking for.*

An alternative version...



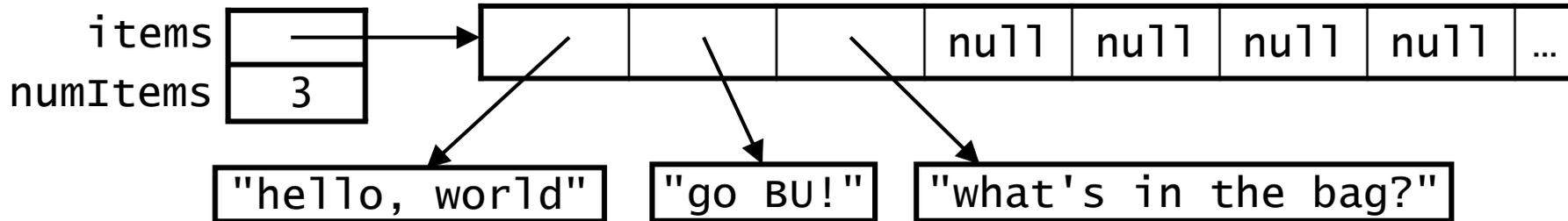
```
public boolean contains(Object item) {  
    boolean found = false;  
    for (int i = 0; i < this.numItems; i++) {  
        if (this.items[i].equals(item)) { // not ==  
            found = true;  
            break;  
        }  
    }  
    return found;  
}
```

An alternative version...



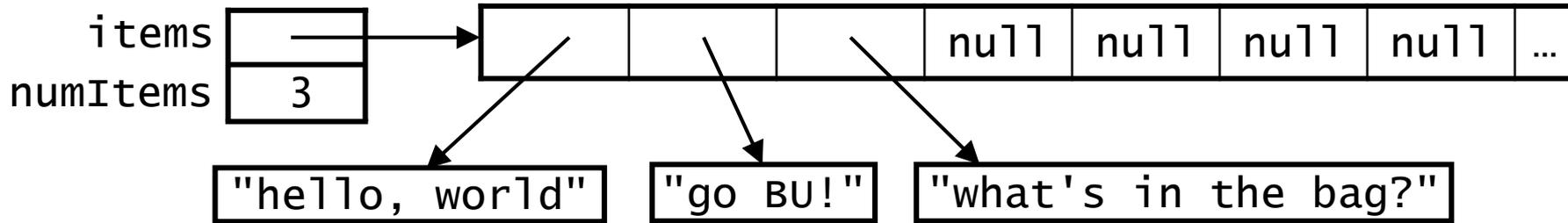
```
public boolean contains(Object item) {  
    boolean found = false;  
    for (int i = 0; i < this.numItems && !found; i++) {  
        if (this.items[i].equals(item)) { // not ==  
            found = true;  
        }  
    }  
    return found;  
}
```

An alternative version...



```
public boolean contains(Object item) {  
    boolean found = false;  
    for (int i = 0; i < this.numItems && !found; i++) {  
        if (this.items[i].equals(item)) { // not ==  
            found = true;  
        }  
    }  
    return found;  
}
```

An alternative version...



```
public boolean contains(Object item) {  
    boolean found = false;  
    for (int i = 0; i < this.numItems && !found; i++) {  
        if (this.items[i].equals(item)) { // not ==  
            found = true;  
        } // if  
    } // for  
    return found;  
} // contains
```

Useful strategy for
keeping track of
brace alignment!

A Method That Takes Another Bag as a Parameter:

check that all items in *our* bag are also items in the *other* bag

```
public boolean containsAll(ArrayBag other) {
    boolean is_inthere = true;    // assume we will find
                                   // all items
    if (other == null || other.numItems <= 0)
        // If the array bag that is passed is empty
        // then there is no need to check further
        is_inthere = false;
    else {
        // check that each item in the other bag
        // is contained in this bag.
        for (int i = 0; i < other.numItems; i++) {
            if (!contains(other.items[i])) {
                // an item in the other bag is not in
                // this bag, no need to check further
                is_inthere = false;
                break;
            }
        }
    }
    return (is_inthere);
}
```

A Method That Takes Another Bag as a Parameter:

check that all items in *our* bag are also items in the *other* bag

```
public boolean containsAll(ArrayBag other) {
    boolean is_inthere = true;    // assume we will find
                                   // all items
    if (other == null || other.numItems <= 0)
        // If the array bag that is passed is empty
        // then there is no need to check further
        is_inthere = false;
    else {
        // check that each item in the other bag
        // is contained in this bag.
        for (int i = 0; i < other.numItems; i++) {
            if (!this.contains(other.items[i])) {
                // an item in the other bag is not in
                // this bag, no need to check further
                is_inthere = false;
                break;
            }
        }
    }
    return (is_inthere);
}
```

A 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!

A 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();
```

A 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

A Type Mismatch

- Here are the headers of `StringBag`:

```
public  
public
```

Recall, *this will perform integer division:*

- Poly

```
Array  
str  
str
```

```
int a = 5;  
double result = a / 2;
```

- 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

A Type Mismatch

- Here are the headers of t

```
public  
public
```

Recall, we can explicitly change one of the operands:

- Poly

```
Arr  
st  
str
```

```
int a = 5;  
double result = a / 2.0;
```

- 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

A Type Mismatch

- Here are the headers of t

```
public  
public
```

Recall, we can also type cast one of the operands!

- Poly

```
Arr  
st  
str
```

```
int a = 5;  
double result = (double) a / 2;
```

- 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

A Type Mismatch

- Here are the headers of t

```
public  
public
```

Similar concept!

- Poly

```
Array  
str  
str
```

We use type casting to allow for our *object* to be treated like a string!

- 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