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 fac that the items don't have a position (and thus we can ay "get the 5th item in the Bag")
  - `carry (or move) t`

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

A Bag Data Structure

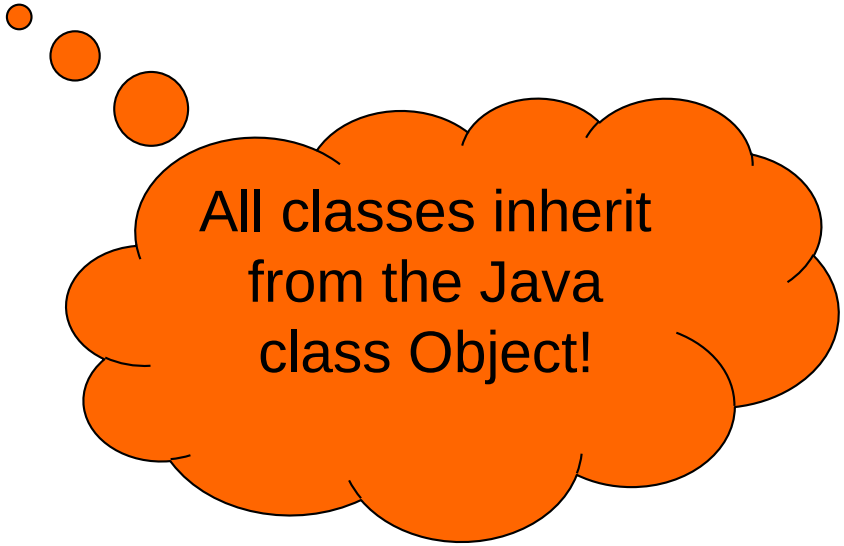
- The operations we want our `Bag` to support:
  - `add(`*`item`*`)`: add `item` to the `Bag`
  - `remove(`*`item`*`)`: remove one occurrence of `item` (if any) from the `Bag`
  - `contains(`*`item`*`)`: check if `item` is in the `Bag`
  - `numItems()`: get the number of items in the `Bag`
  - `grab()`: 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

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;
    ...
}
```



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;
    ...
}
```

- This allows us to store *ar* ... y, thanks to the power of polymor

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

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

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

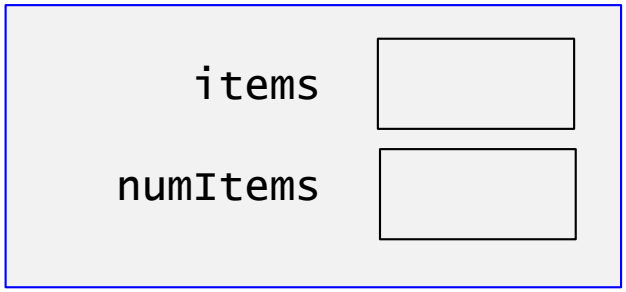
an array of any object type

- 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

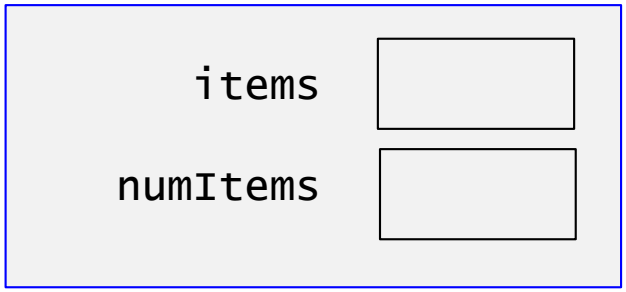
- We store the items in an array of

```
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 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

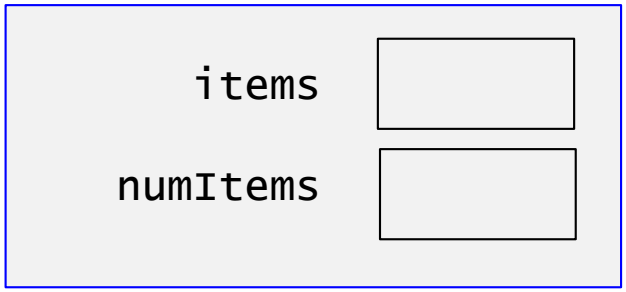
- We store the items in an array of

```
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 of object in the `items` array, thanks to the power of polymorphism:

```
ArrayBag bag = new ArrayBag(5);
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() {
        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 co
  - the parameters must differ
- The first constructor is useful for s
  - creates an array with room for 50 item

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

- We can have two different co
  - the parameters must differ
- The first constructor is useful for s
  - creates an array with room for 50 item

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 Obje
        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 > 0");
        }
        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_MA
        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:

an alternative using *constructor chaining*

```
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(DEFAULT_MAX_SIZE); // call custom constructor

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

Two Constructors

an alternati

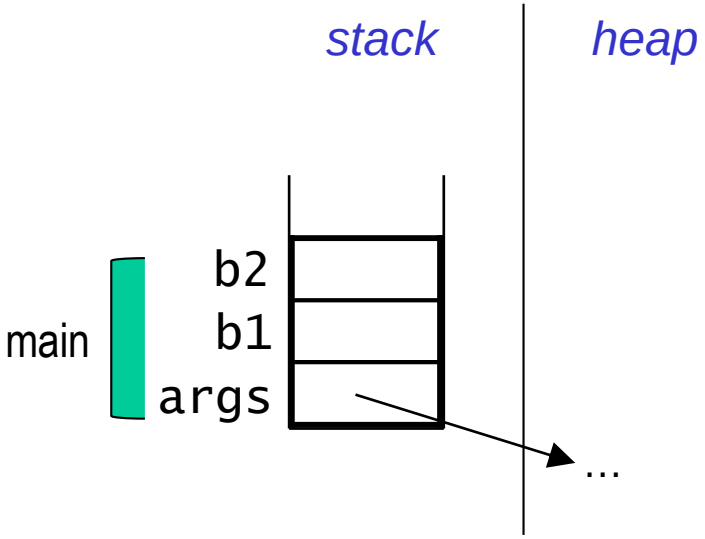
Rather than duplicating the code in both constructors, have one constructor call the other!

```
public class Array
    private Objec
    private int numIte
    public static final int DEFAULT_MAX_SIZE = 50;
    private static int numBagsCreated = 0;
    public ArrayBag() {

        this(DEFAULT_MAX_SIZE); // call custom constructor

    }
    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);

    …
}
```

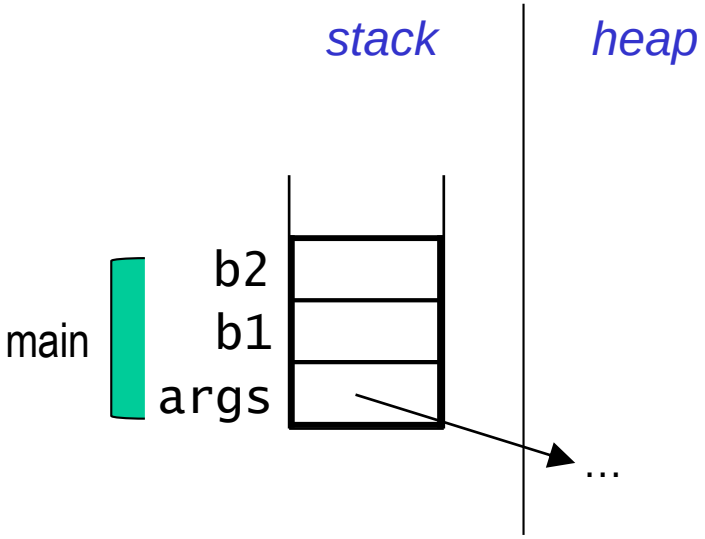
```
// constructor
public ArrayBag(int maxSize) {
   ...  // error-checking
   this.items = new Object[maxSize];
   this.numItems = 0;
}
```

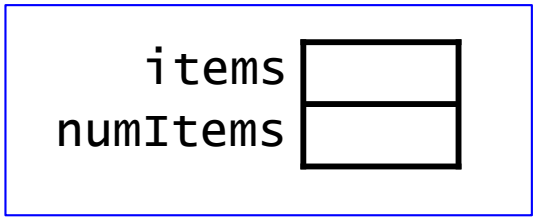


Example: Creating Two `ArrayBag` Objects

```
// 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;
}
```

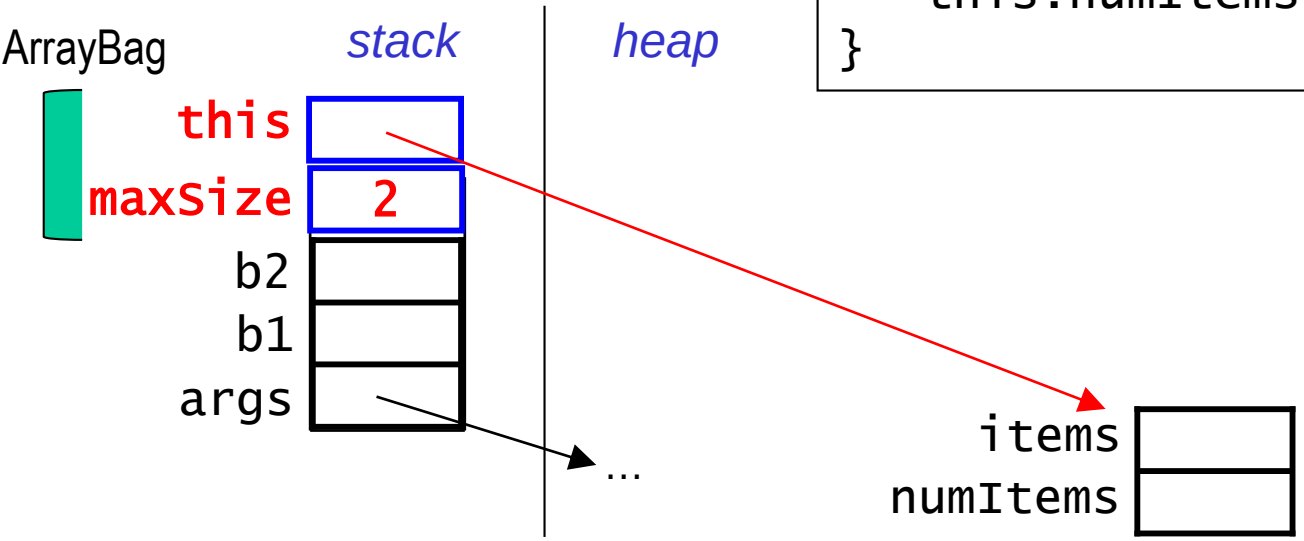




Example: Creating Two `ArrayBag` Objects

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

    …
}
```

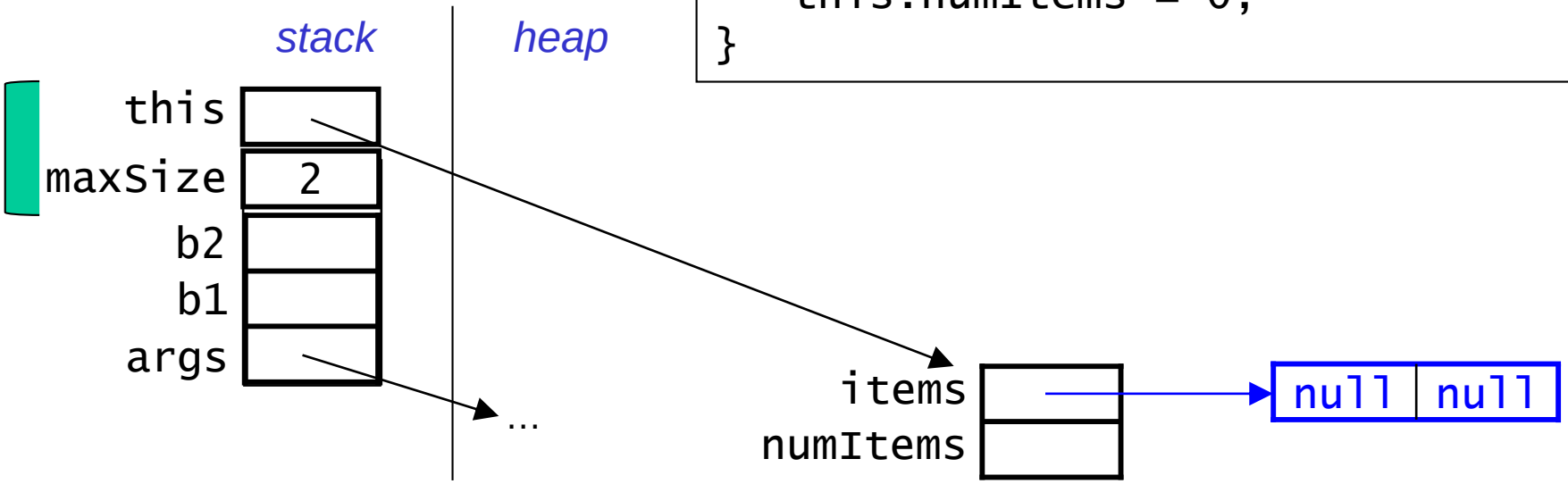
```
// constructor
public ArrayBag(int maxSize) {
    ...  // error-checking
    this.items = new Object[maxSize];
    this.numItems = 0;
}
```



Example: Creating Two `ArrayBag` Objects

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

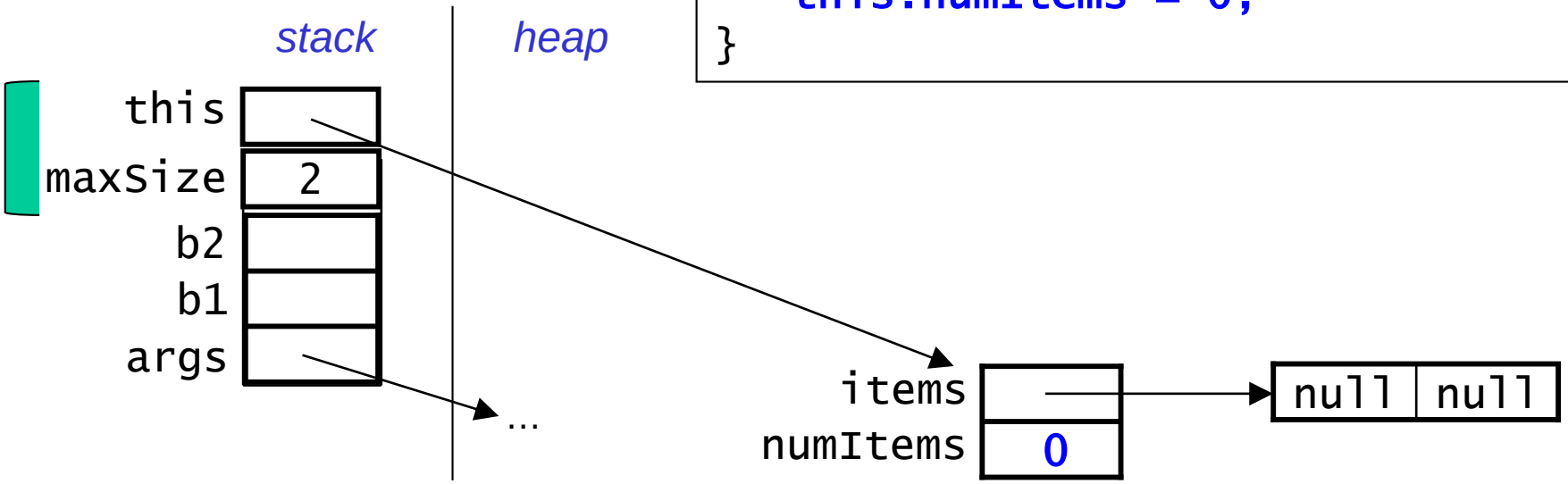
```
// constructor
public ArrayBag(int maxSize) {
    ...  // error-checking
    this.items = new Object[maxSize];
    this.numItems = 0;
}
```



Example: Creating Two `ArrayBag` Objects

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

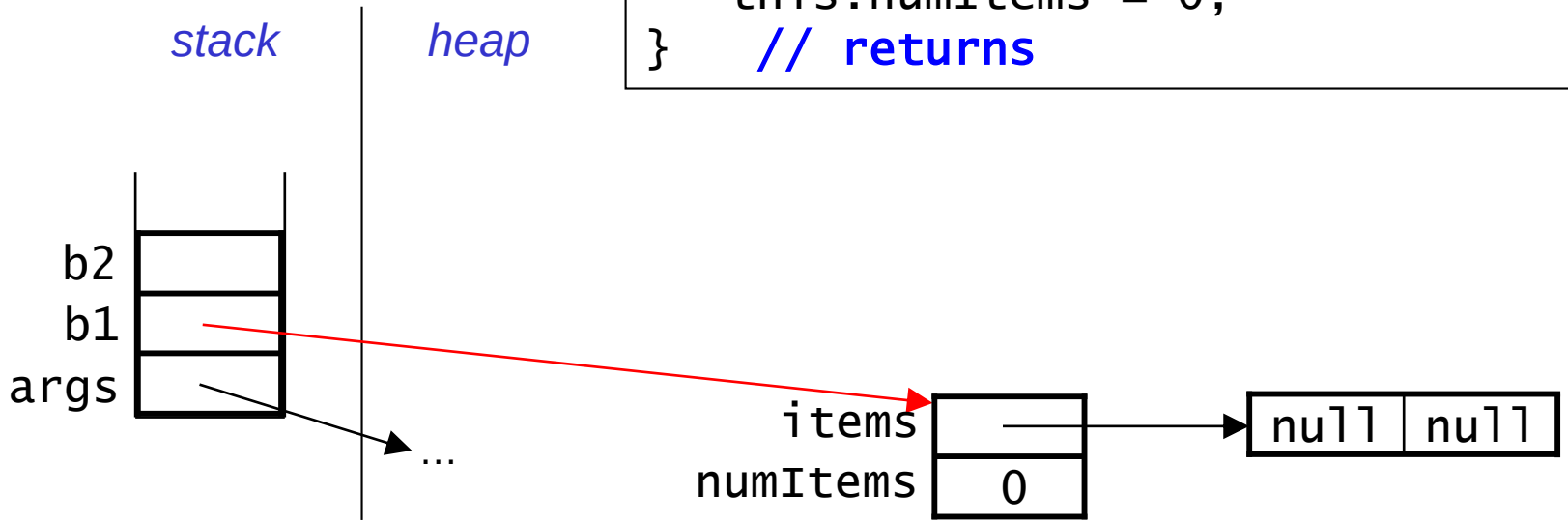
```
// constructor
public ArrayBag(int maxSize) {
    ...  // error-checking
    this.items = new Object[maxSize];
    this.numItems = 0;
}
```



Example: Creating Two `ArrayBag` Objects

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

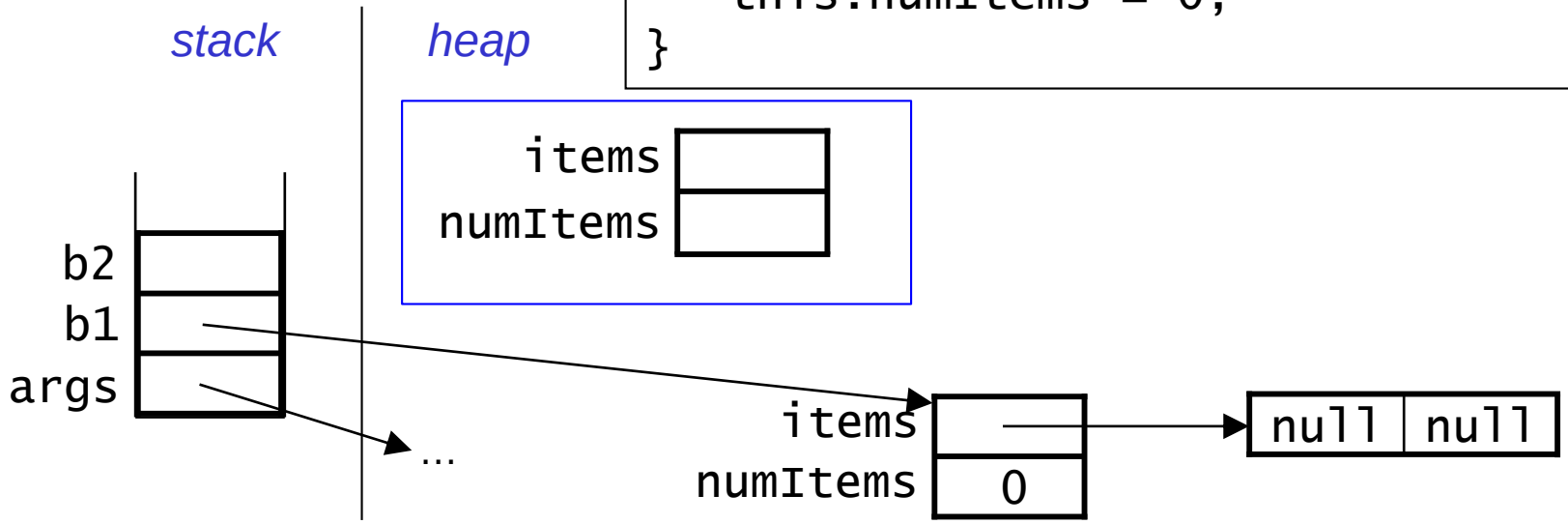
```
// constructor
public ArrayBag(int maxSize) {
   ...  // error-checking
   this.items = new Object[maxSize];
   this.numItems = 0;
}   // returns
```



Example: Creating Two `ArrayBag` Objects

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

    …
}
```

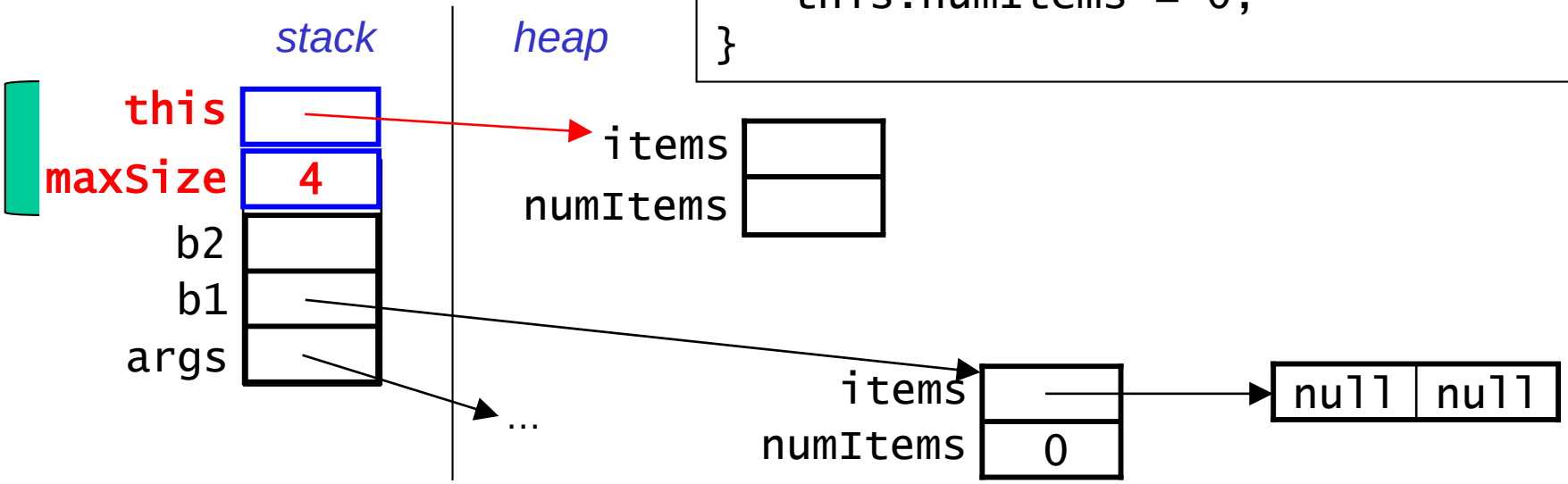
```
// constructor
public ArrayBag(int maxSize) {
    ...  // error-checking
    this.items = new Object[maxSize];
    this.numItems = 0;
}
```



Example: Creating Two `ArrayBag` Objects

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

    …
}
```

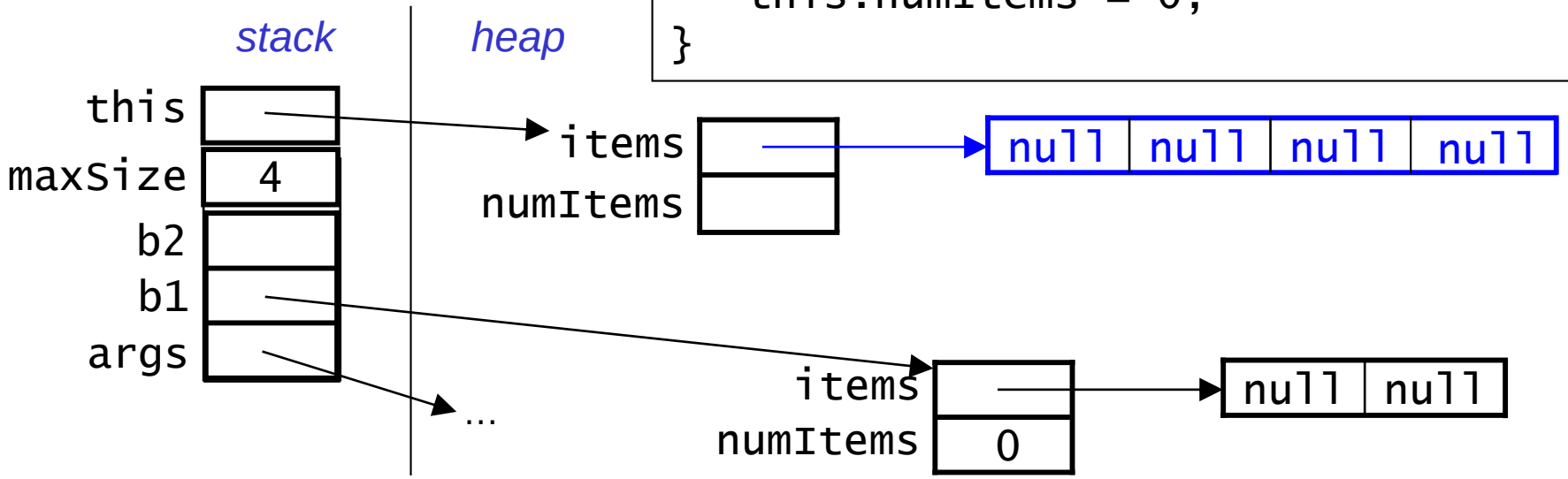
```
// constructor
public ArrayBag(int maxSize) {
    ...  // error-checking
    this.items = new Object[maxSize];
    this.numItems = 0;
}
```



Example: Creating Two `ArrayBag` Objects

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

    …
}
```

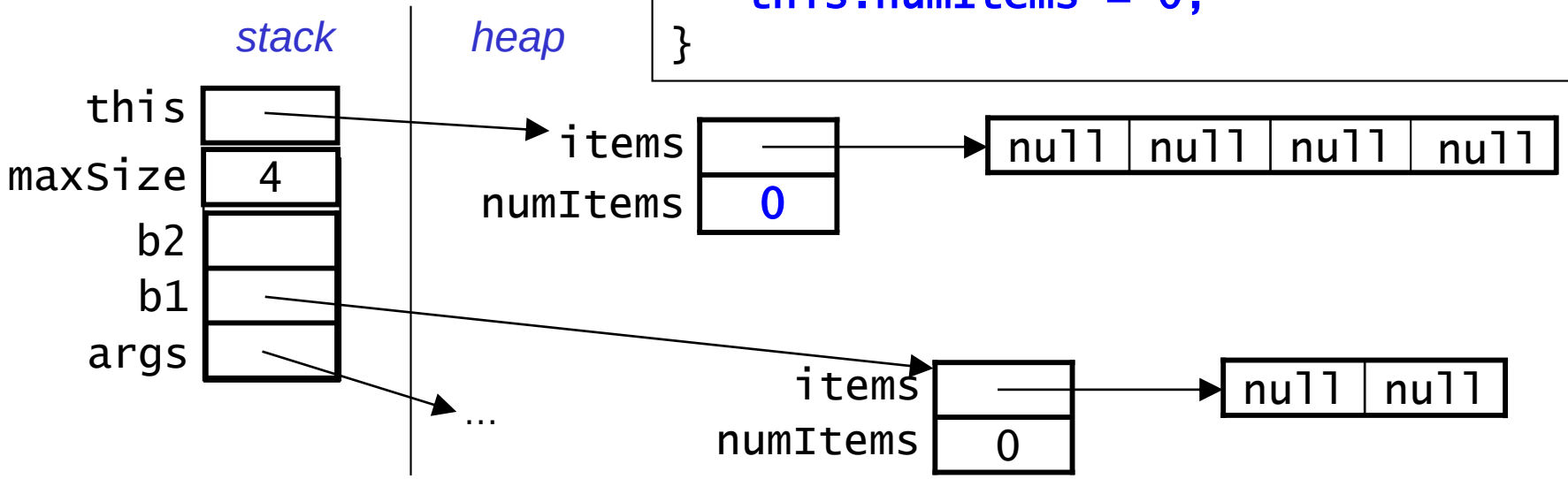
```
// constructor
public ArrayBag(int maxSize) {
    ...  // error-checking
    this.items = new Object[maxSize];
    this.numItems = 0;
}
```



Example: Creating Two `ArrayBag` Objects

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

    …
}
```

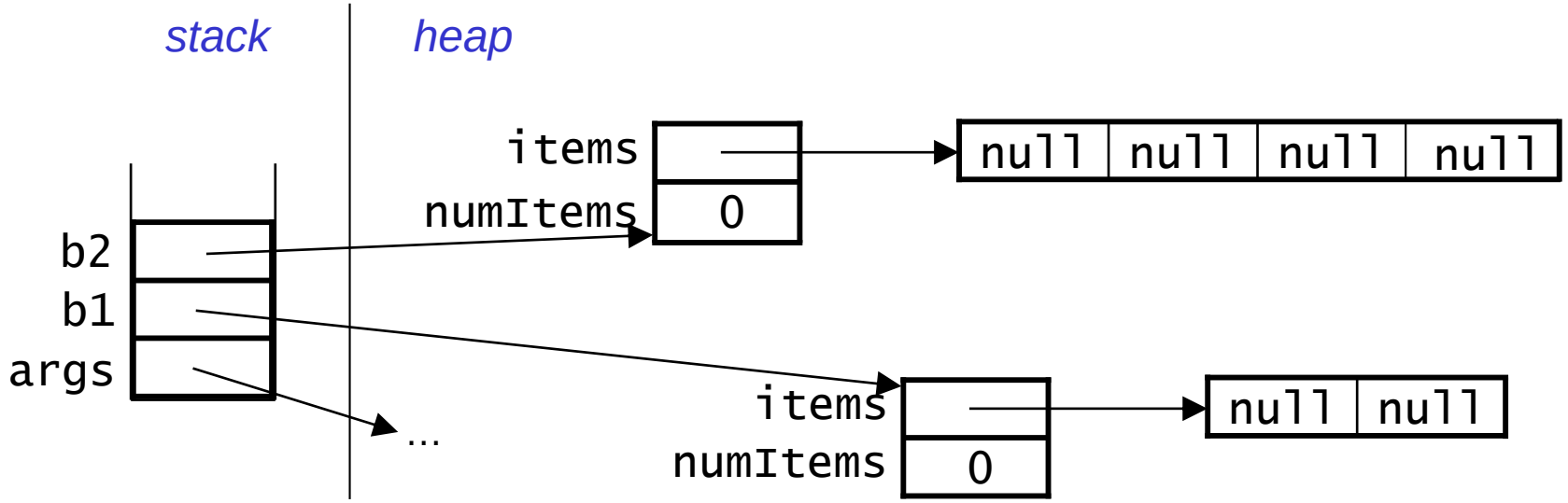
```
// constructor
public ArrayBag(int maxSize) {
    ...  // error-checking
    this.items = new Object[maxSize];
    this.numItems = 0;
}
```



Example: Creating Two `ArrayBag` Objects

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

    …
}
```

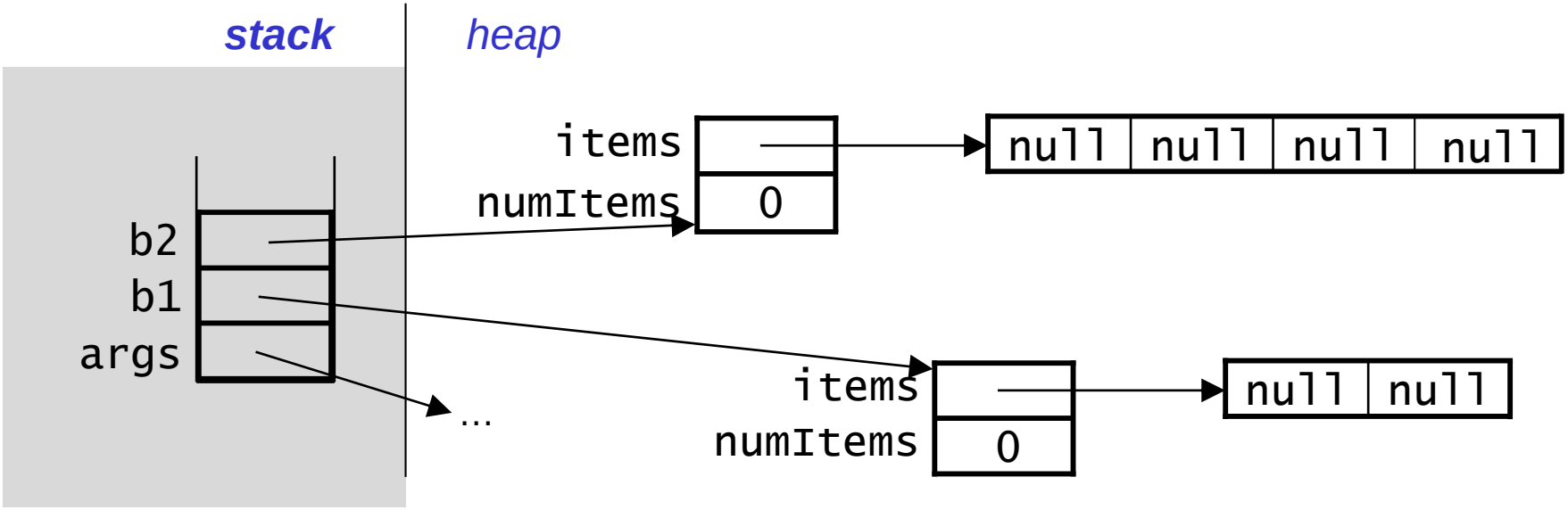
- After the objects have been created, here's what we have:



Example: Creating Two `ArrayBag` Objects

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

    …
}
```

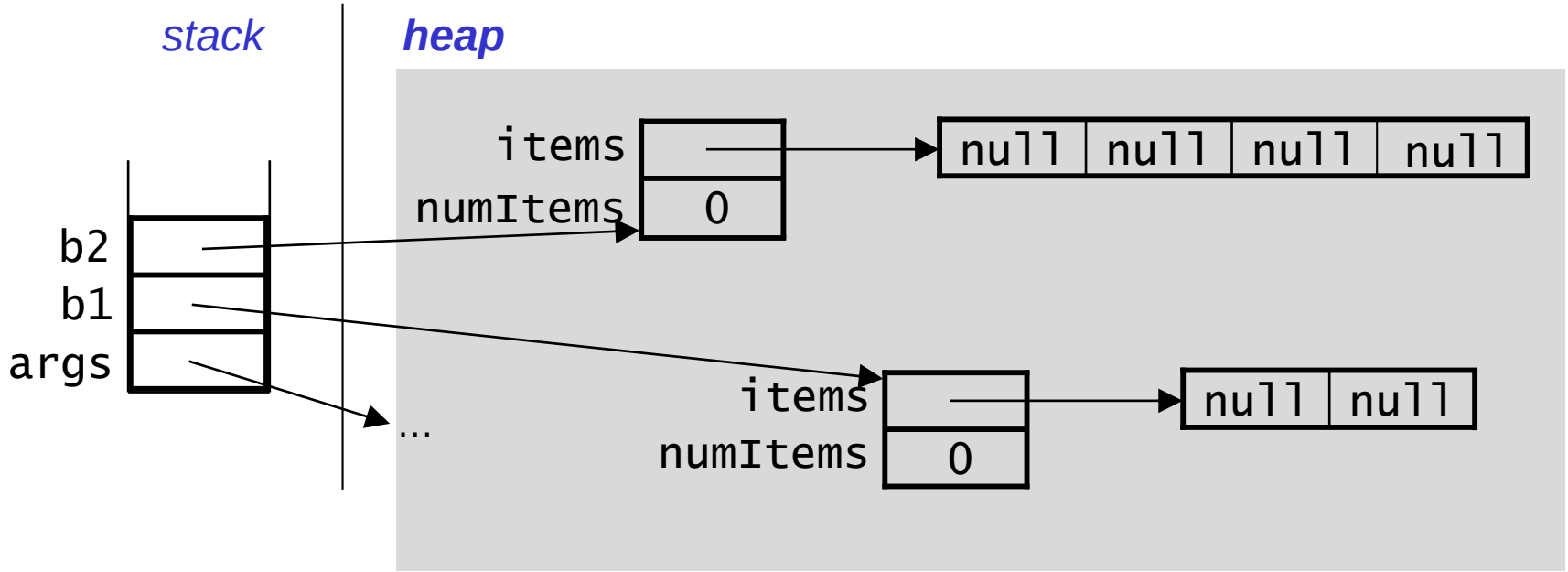
- After the objects have been created, here's what we have:



Example: Creating Two `ArrayBag` Objects

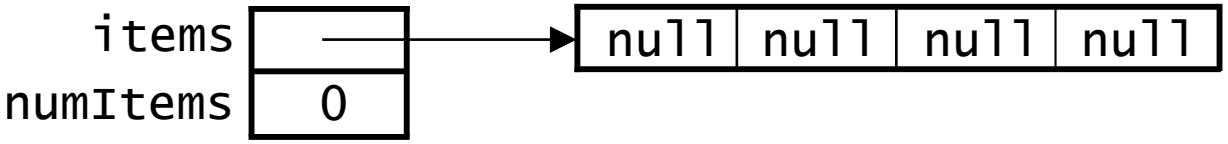
```
// client
public static void main(String[] args) {
    ArrayBag b1 = new ArrayBag(2);
    ArrayBag b2 = new ArrayBag(4);

    …
}
```

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

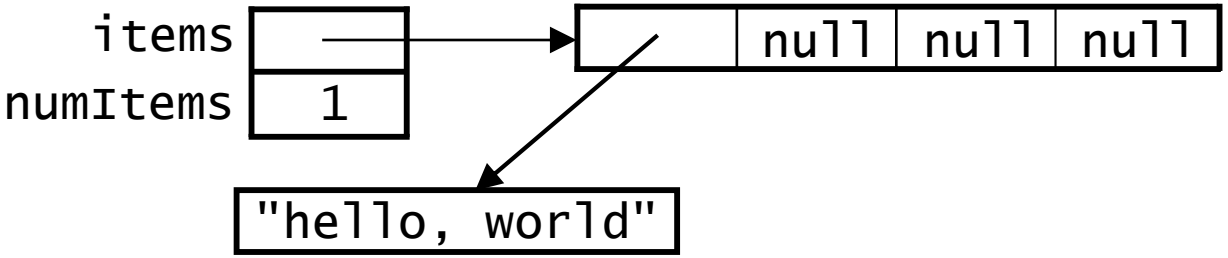


Adding Items

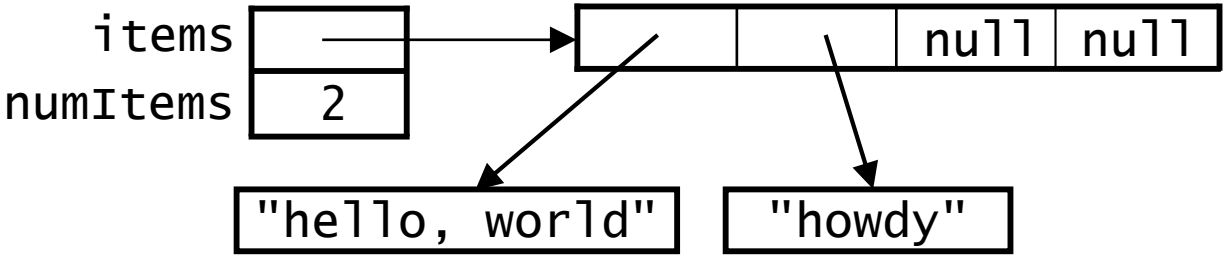
- We fill the array from left to right. Here's an empty bag:



- After adding the first item:

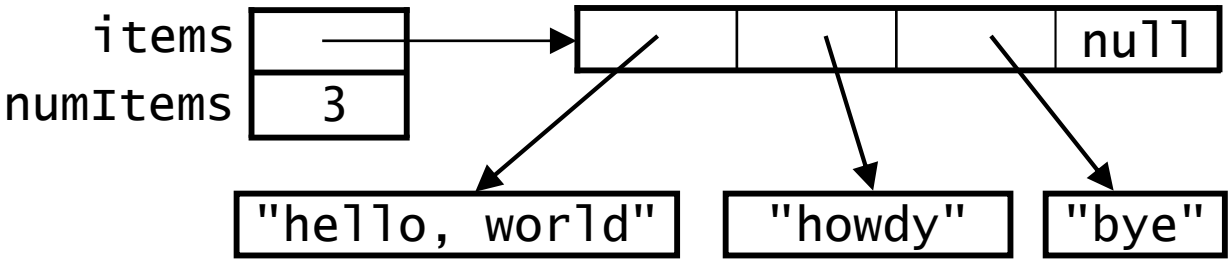


- After adding the second item:



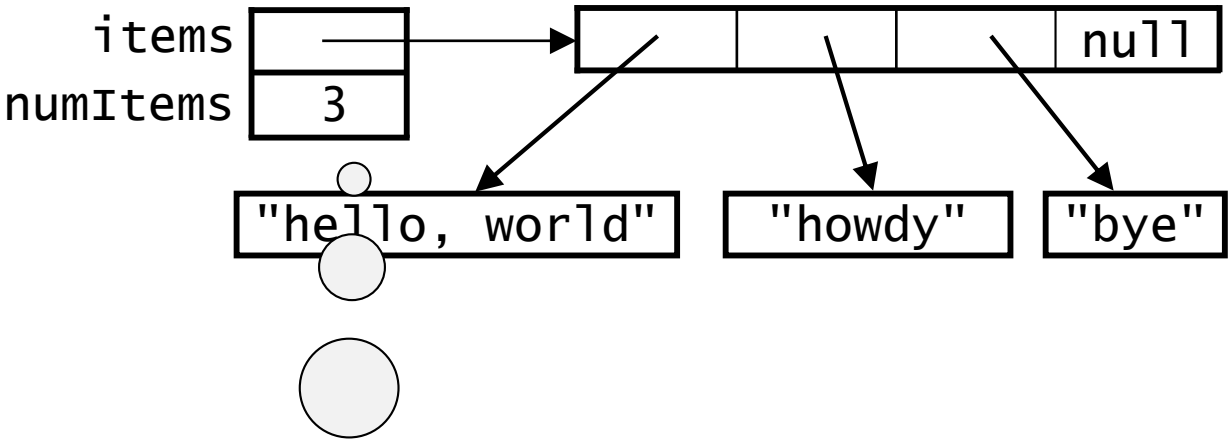
Adding Items (cont.)

- After adding the third item:



Adding Items (cont.)

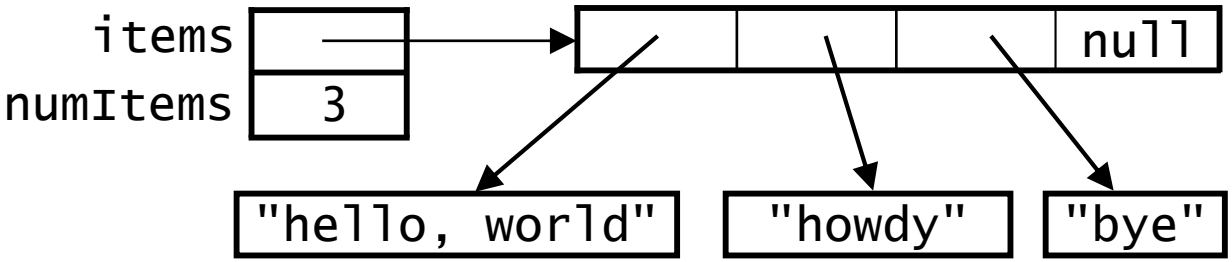
- After adding the third item:



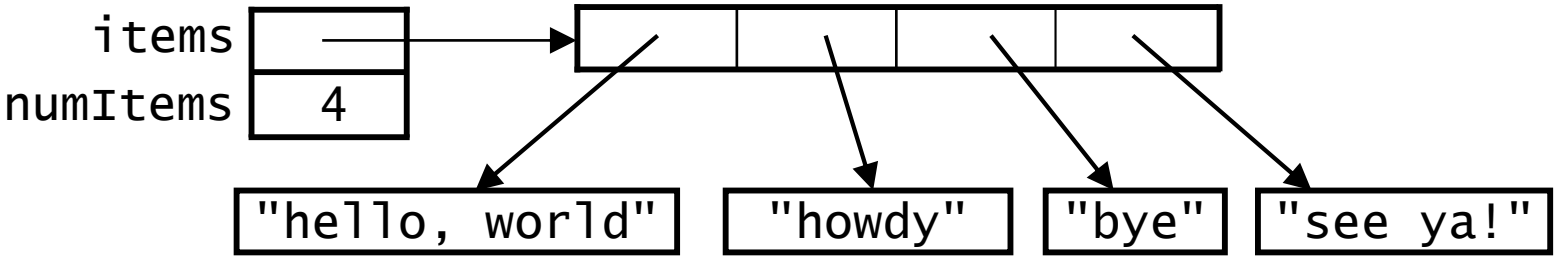
Note the correlation between the *number* of items currently in the bag and the *index* (or offset) we will use to add the next item!

Adding Items (cont.)

- After adding the third item:



- After adding the fourth item:



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

A Method for Adding an Item to a Bag

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

```
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);
    }
    ...
}
```

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!

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 when the array is empty and there are no items, the value of *numItems* is 0!

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

Can take advantage of post increment operator to combine the two statements!

A Method for Adding an Item to a Bag

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

```
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;         // indicate success
        }
        return(item_added);
    }
    ...
}
```

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;         // indicate success
        }
        return(item_added);
    }
    ...
}
```

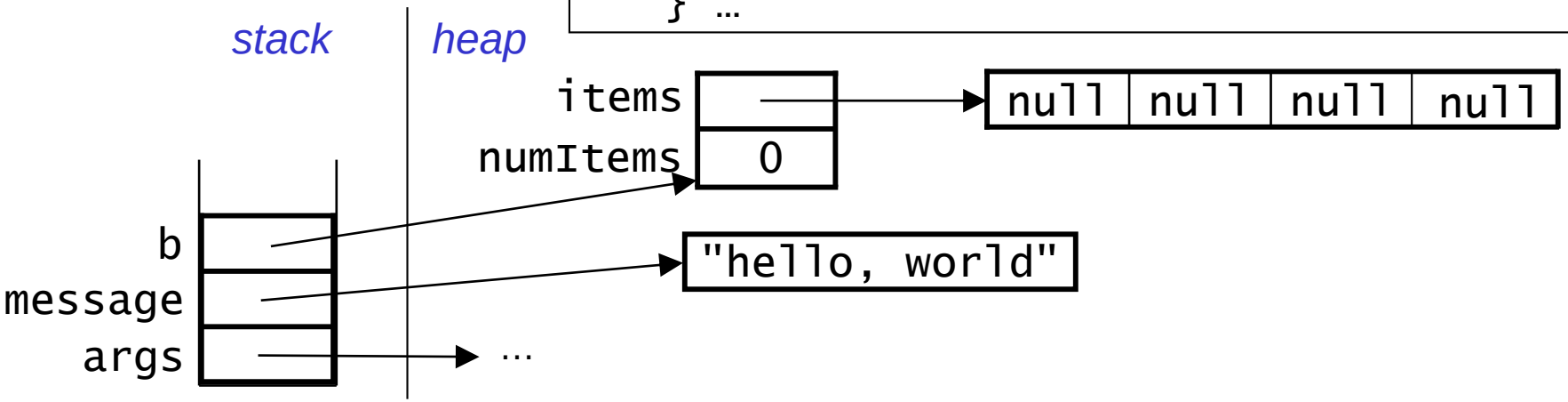
- takes an object of any type!
- returns a `boolean` to indicate whether the operation succeeded

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);
    …
}
```

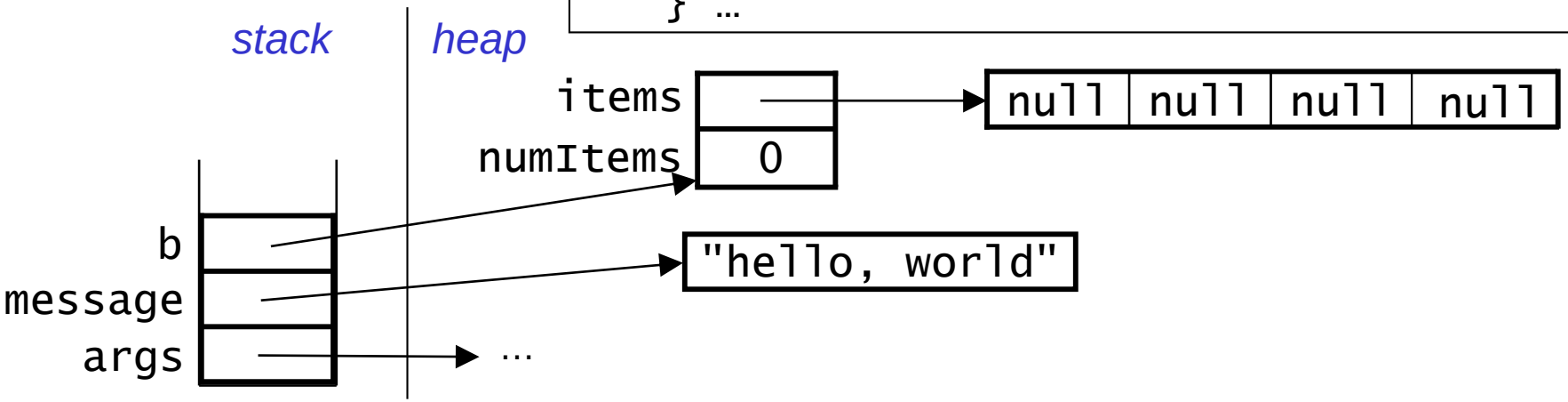
```
public boolean add(Object item) {
    …
    else {
        this.items[this.numItems] = item;
        this.numItems++;
        item_added = true;
    } …
```



Example: Adding an Item

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

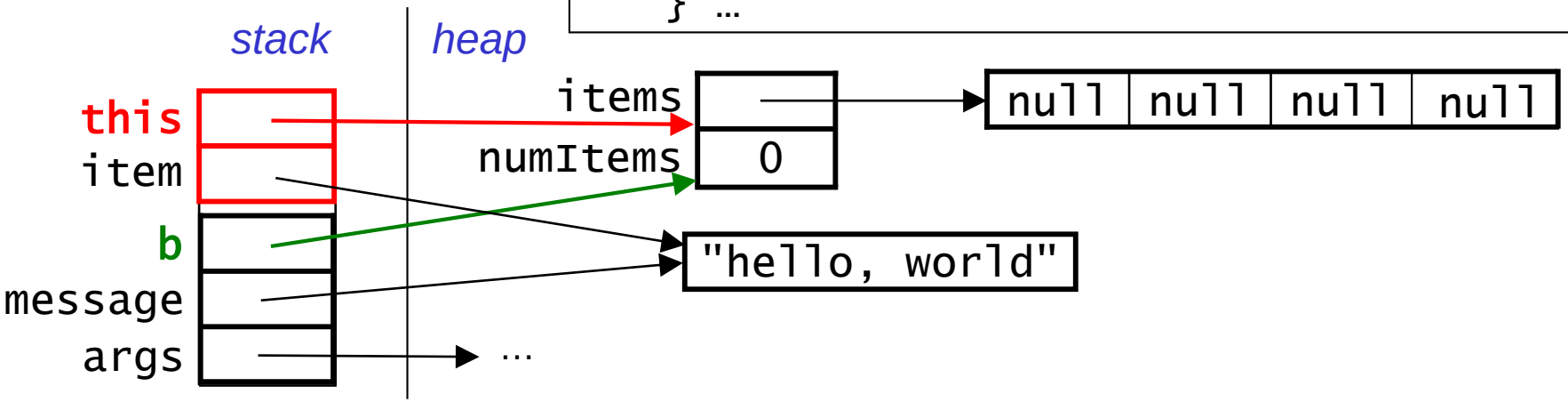
```
public boolean add(Object item) {
    …
    else {
        this.items[this.numItems] = item;
        this.numItems++;
        item_added = true;
    } …
```



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;
    } …
```

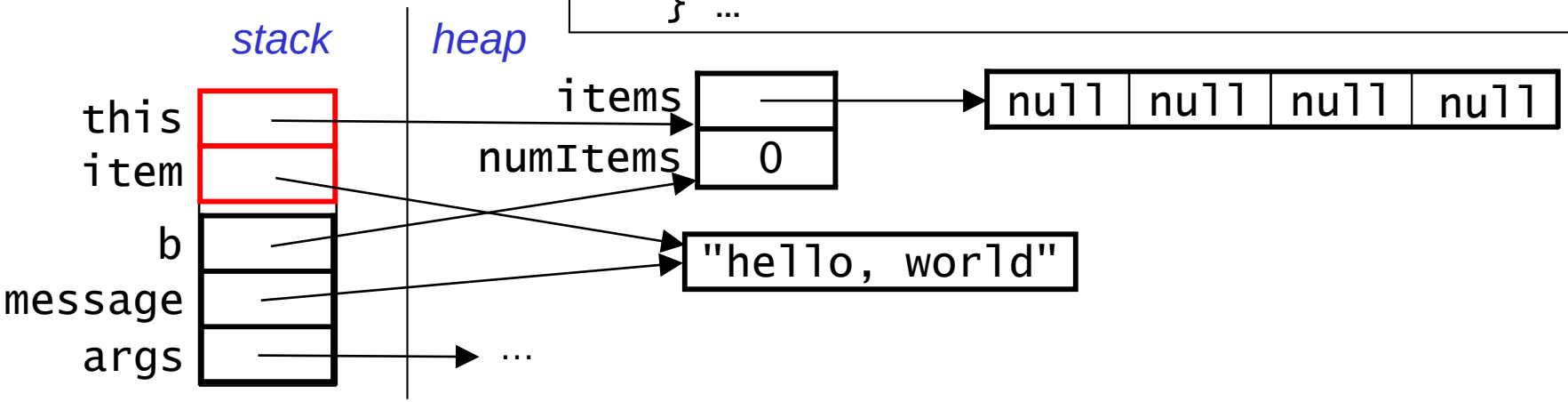


- add's stack frame includes:
  - `item`, which stores a copy of the reference passed as a param.
  - `this`, which stores a reference to the called `ArrayBag` object

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;
    } …
```

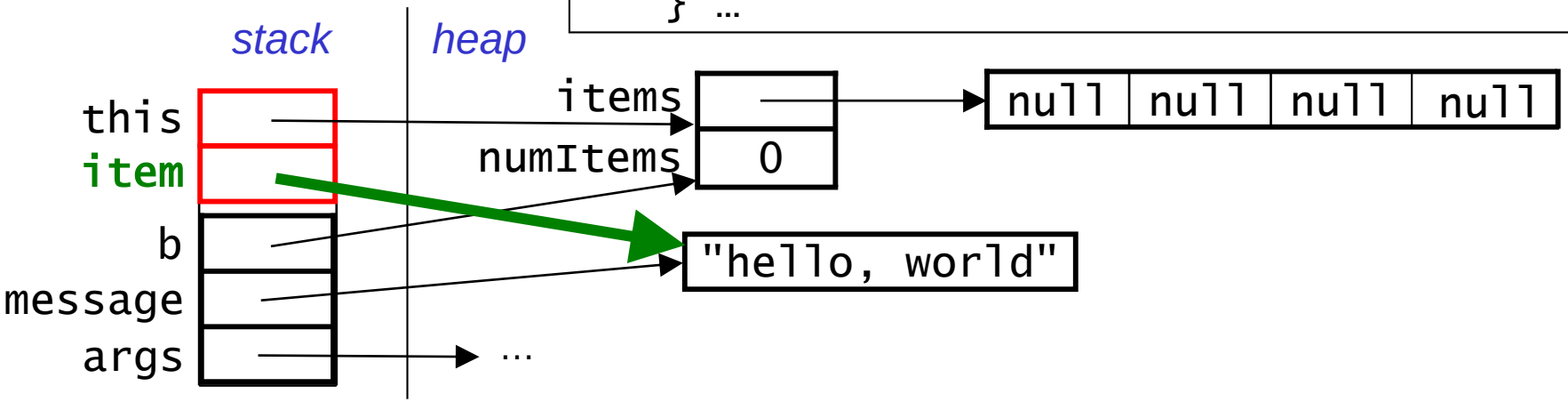


- The method modifies the `items` array and `numItems`.

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;
    } …
```

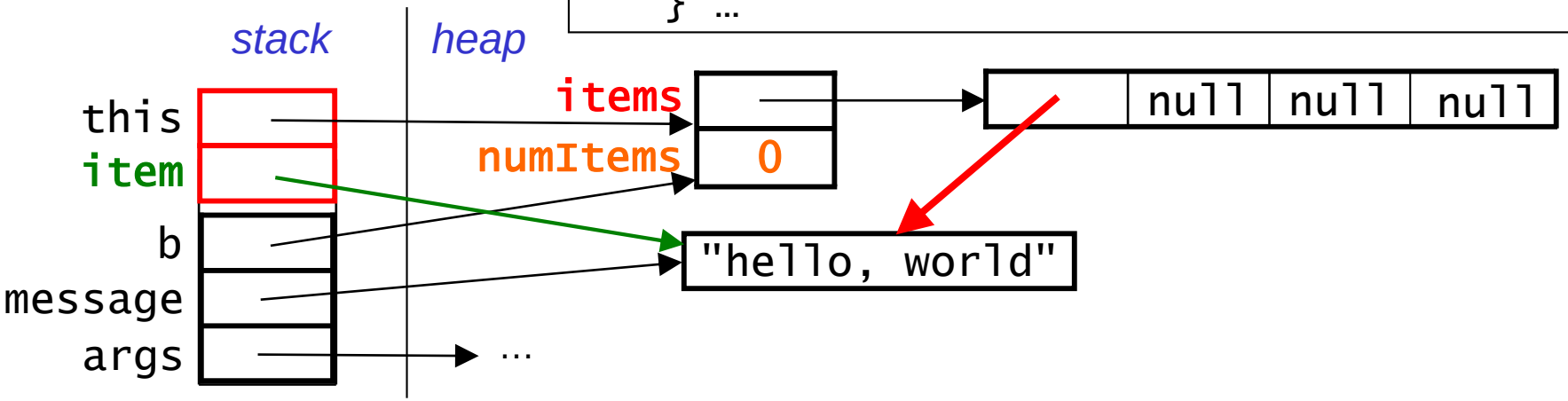


- The method modifies the `items` array and `numItems`.

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;
    } …
```

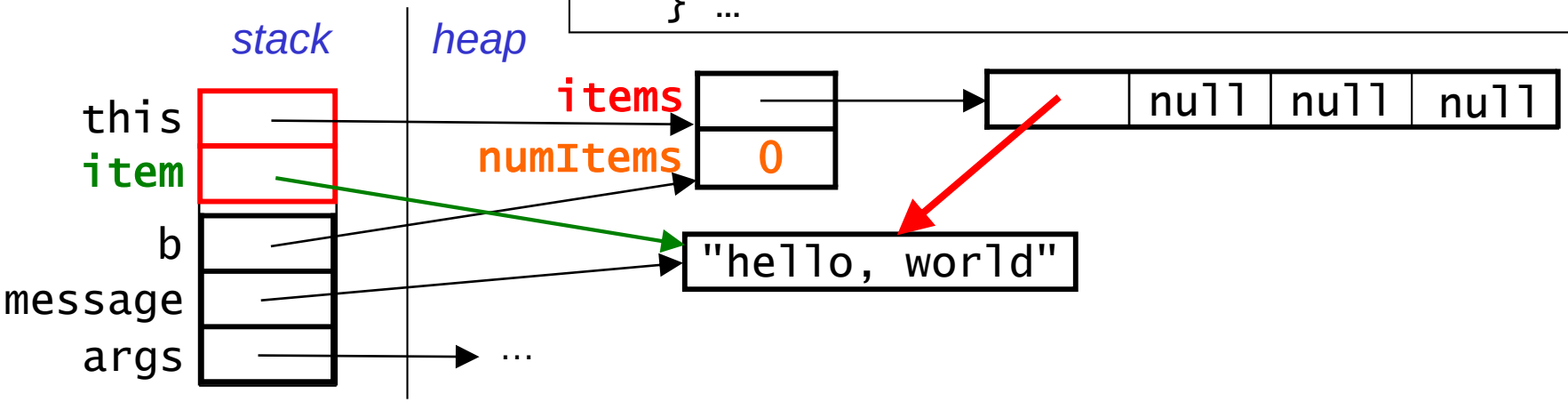


- The method modifies the `items` array and `numItems`.

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;
    } …
```

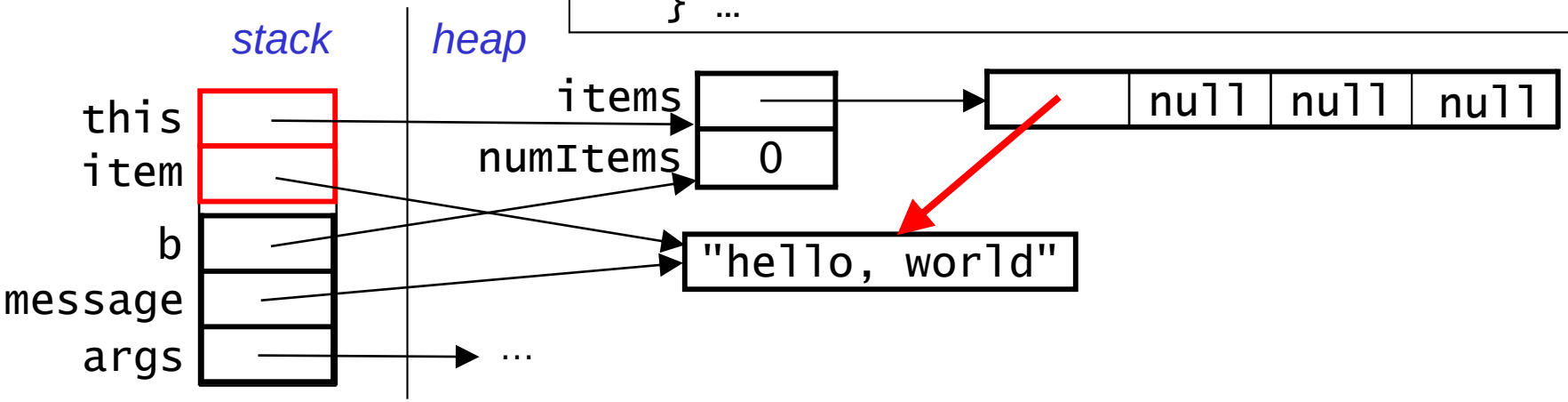


- The method modifies the `items` array and `numItems`.
  - note that the array stores a copy of the *reference* to the item, not a copy of the item itself.

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;
    } …
```

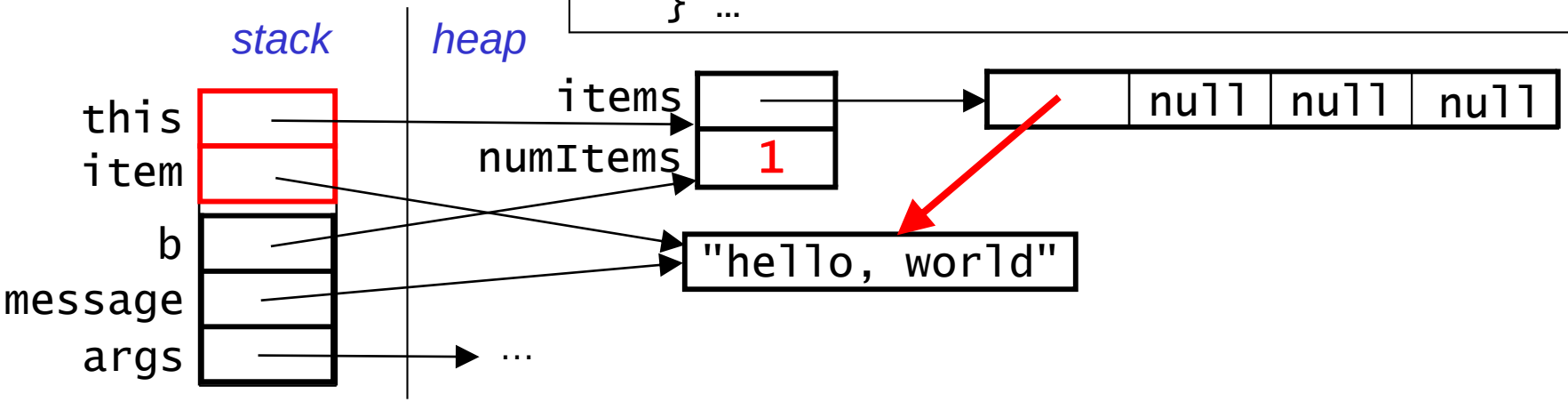


- The method modifies the `items` array and `numItems`.
  - note that the array stores a copy of the *reference* to the item, not a copy of the item itself.

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;
    } …
```

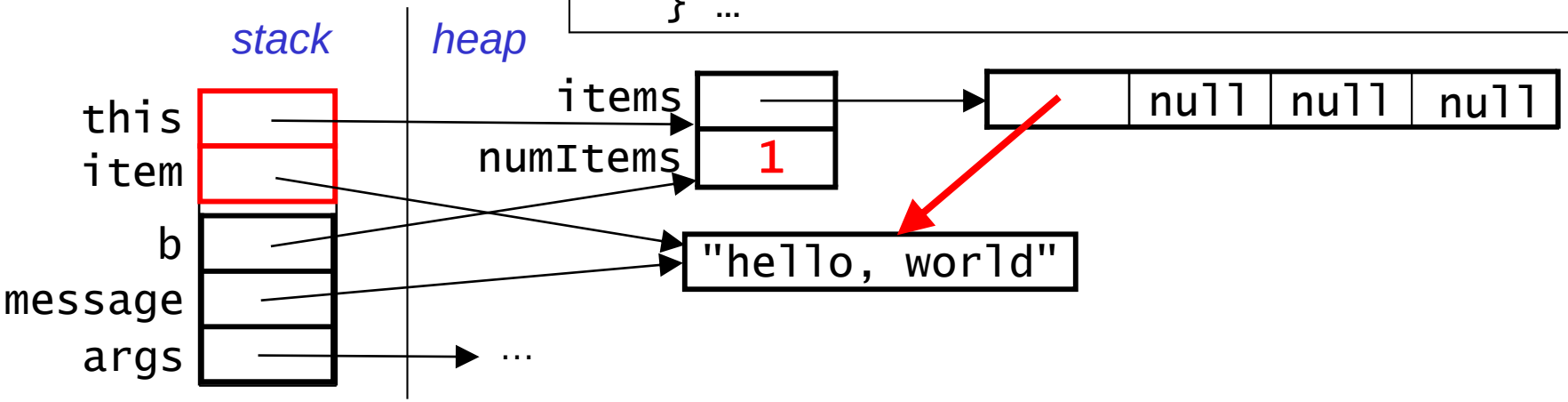


- The method modifies the `items` array and `numItems`.
  - note that the array stores a copy of the *reference* to the item, not a copy of the item itself.

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;
    } …
```

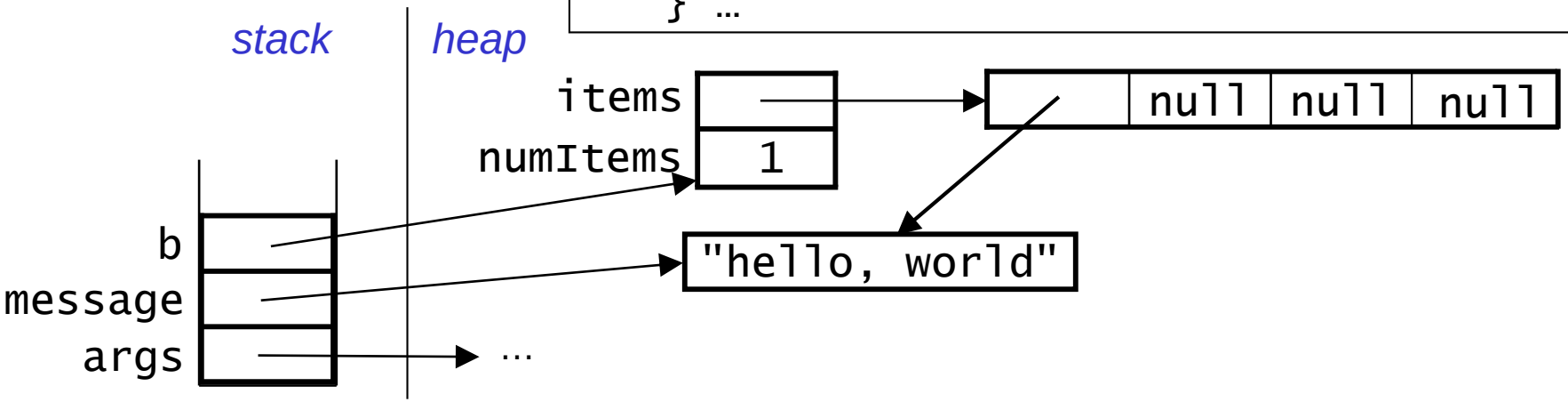


- The method modifies the `items` array and `numItems`.
  - note that the array stores a copy of the *reference* to the item, not a copy of the item itself.

Example: Adding an Item (cont.)

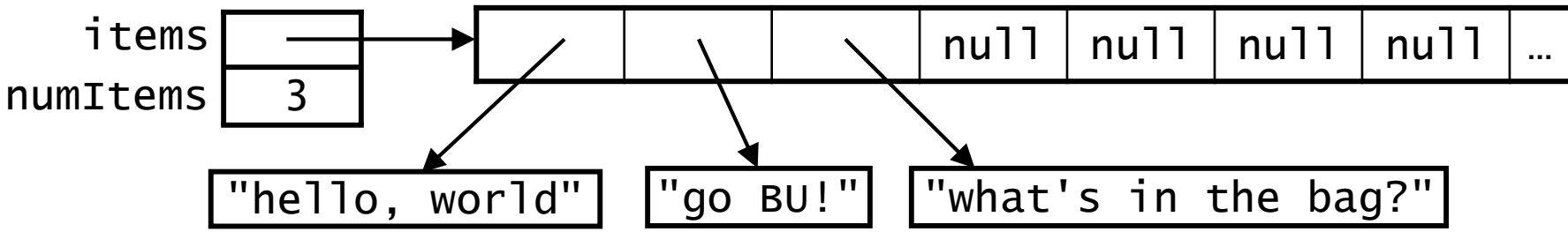
```
public static void main(String[] args) {
    String message = "hello, world";
    ArrayBag b = new ArrayBag(4);
    b.add(message);
    …
}
```

```
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.

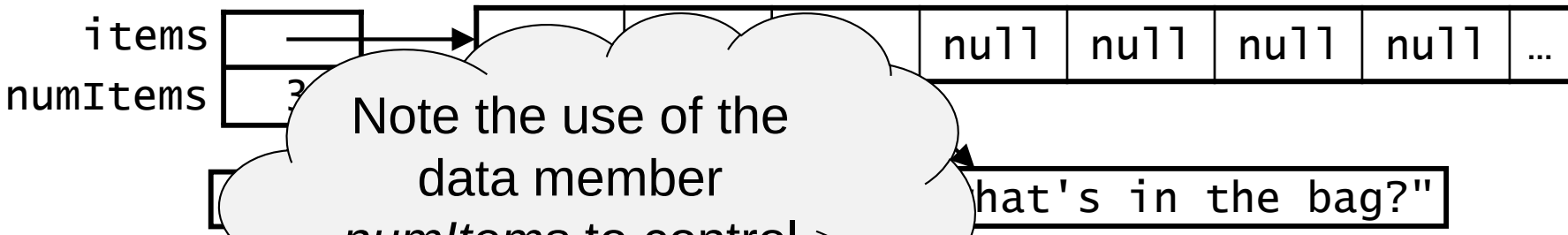
Extra Practice: Determining if a Bag Contains an Item



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

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

Extra Practice: Determining if a Bag Contains an Item

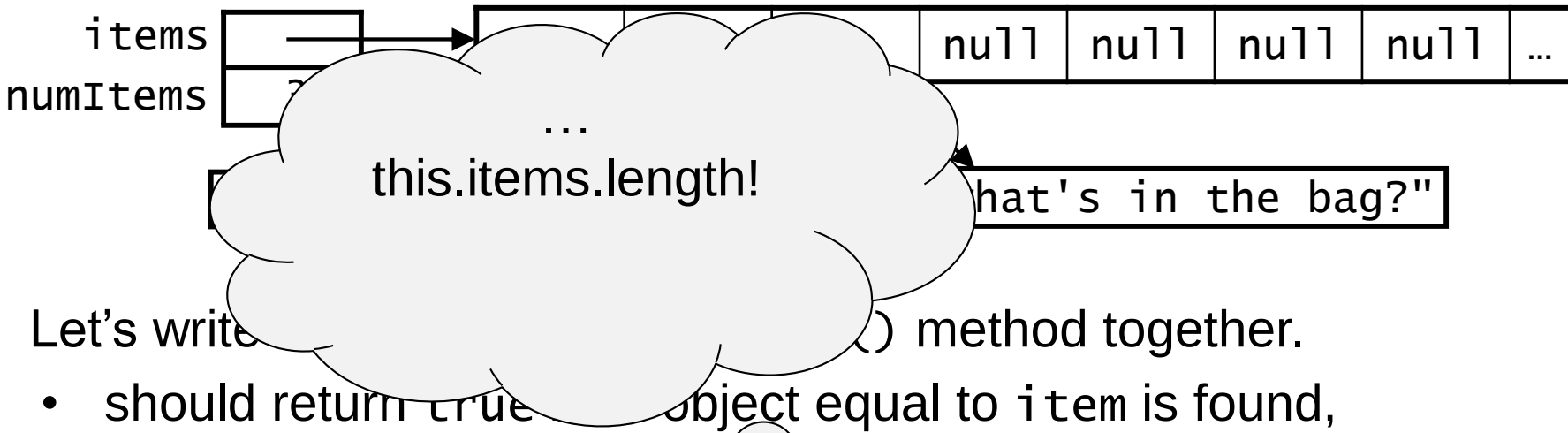


Note the use of the data member *numItems* to control the loop and not…

- Let's write … () method together.
  - should return true … 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;
}
```

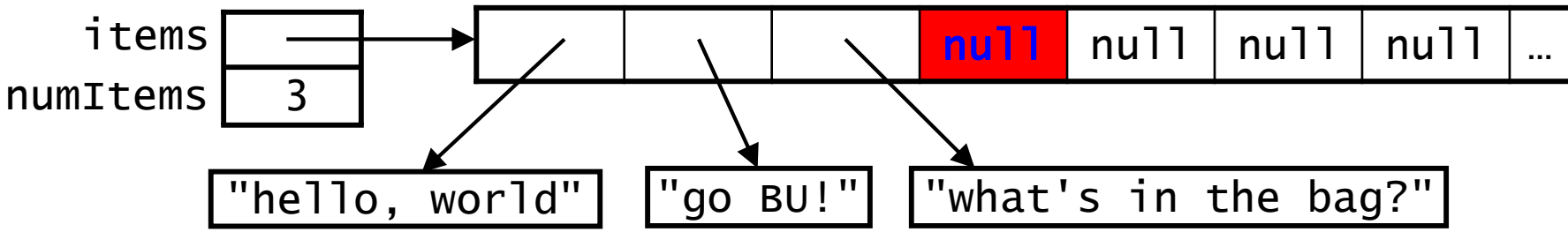
Would this work instead?



- Let's write () method together.
  - should return true 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!*

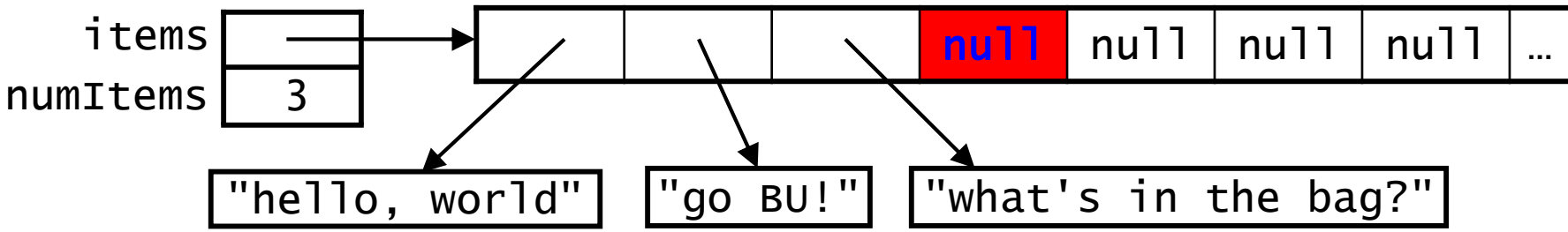


- Let's write the `ArrayBag contains()` method together.
  - should return `true` if an object equal to `item` is found, and `false` otherwise.

```
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!

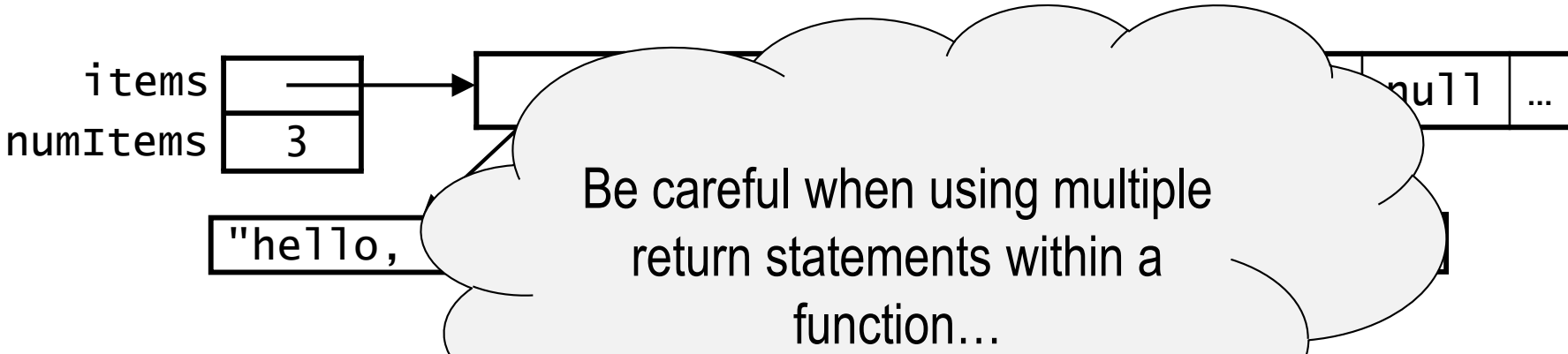
Would this work instead? *no!*



- 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;
}
```

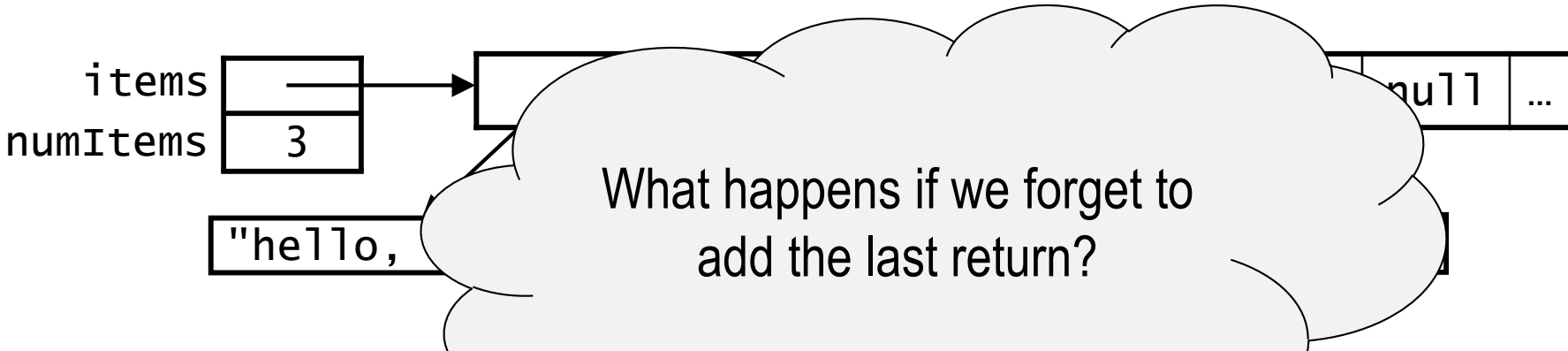
Would this work instead? *no!*



- Let's write the `Array` …
  - should return `true` if an object … `em` 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? *no!*



- Let's write the `Array` [illegible] .
  - should return `true` if an [illegible] `tem` 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;
        }
    }

}
```

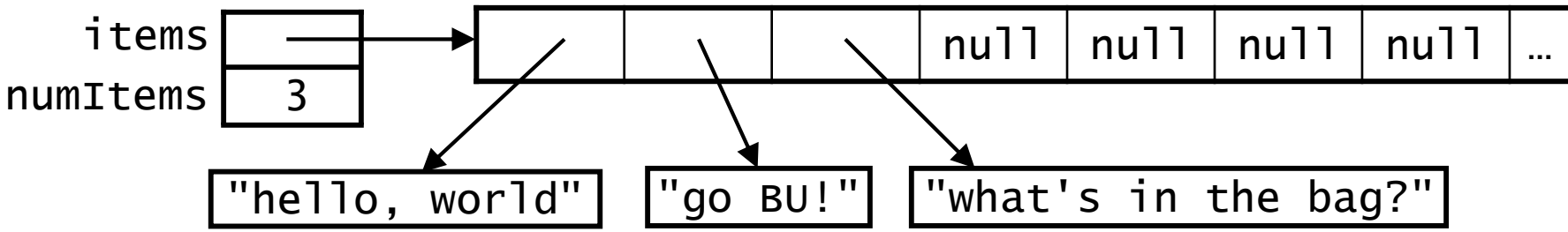
Another Incorrect `contains()` Method

```
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*

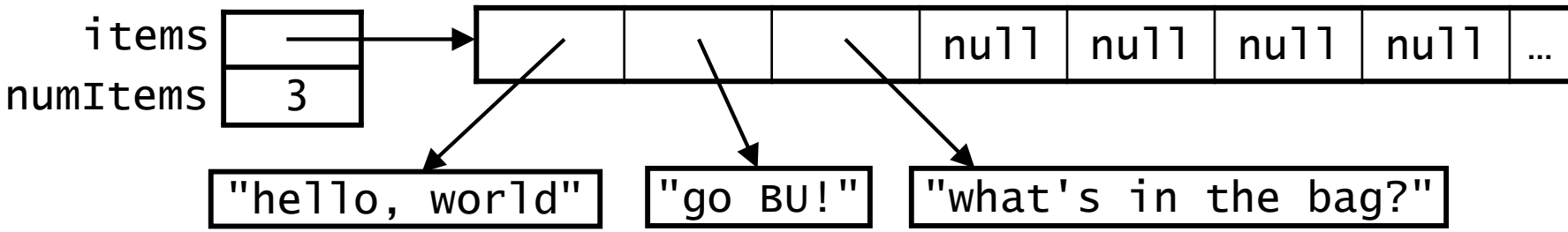
- When would it work? *If the first item in the array is the item we are looking for.*

An alternative version…



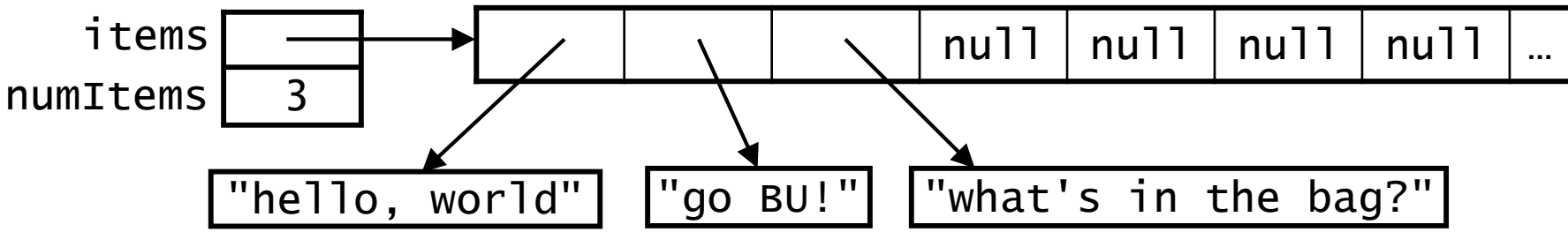

```
public boolean contains(Object item) {

    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;

        } // 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 (!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 it

    if (other == null || oth
        // If the array bag
        // then there is no
        is_inthere = fal
    else {
        // check that eac
        // is contained
        for (int i = 0; i
            if (!contains(oth
                // an item in the other          not in
                // this bag, no need to check further
                is_inthere = false;
                break;
            }
     }
     return (is_inthere);
}
```

Note that *our* bag refers to the bag that the method was called on (*this*) and the *other* bag refers to the object that is being passed to this method.

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 (!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
        // is
        for (
                bre
            }
    }
    return (is_inthere);
}
```

Can also add a check
`numItems > other.numItems`.
If more items are in our bag than the other bag, then our bag cannot be a subset of the other bag.

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 (!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 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

    if (other == null ||
        // If the array
        // then there i
        is_inthere = fal
    else {
        // check that each
        // is contained i 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);
}
```

When calling the method contains(), we are invoking it on the object that the method was called on (*this*).

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)
                    ay bag that is passed is empty
                    s no need to check further
                    se;
                    t each item in the other bag
            //      contai d 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);
}
```

And passing to the method each item in the *other* bag.

A Method That Takes Another Bag as a Parameter:

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

Note that this method has direct access to the data members of the *other* object which was passed in!

```
pub[illegible]tainsAll(ArrayBag other) {
    [illegible]re = true;    // assume we will find
                             // all items
    [illegible]|| other.numItems <= 0)
        [illegible] bag that is passed is empty
        [illegible]s no need to check further
        [illegible]lse;

    [illegible]at ea[illegible]item in the other bag
    [illegible]tained in this bag.
    [illegible](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 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 <u>not</u> 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

```
publi
publ
```

- Poly

```
Arr
st
str
```

- 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

Recall, *we can explicitly change one of the operands:*

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

A Type Mismatch

- Here are the headers of t

```
publi
publ
```

- Poly

```
Arr
st
str
```

- However, this will <u>not</u> 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

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

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

A Type Mismatch

- Here are the headers of t

```
publi
publ
```

- Poly

```
Arr
st
str
```

Similar concept!

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()` `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