Java Review

Reading (Ch1 from Weiss)

you already know...

- Classes and objects
- Static vs instance fields and methods
- Primitive vs reference types
- Private vs public vs package
- Constructors
- Method signatures
- Local variables
- Arrays
- File Operations
- Polymorphism and inheritance, shadowing
- Exceptions

Review: What is an Object?

- An **object** groups together:
  - one or more data values (the object's fields – also known as instance variables)
  - a set of operations that the object can perform (the object's methods)
- In Java, we use a **class** to define a new type of object.
  - serves as a "blueprint“ for objects of that type
  - simple example:

```
public class Rectangle {
    // fields
    private int width;
    private int height;
    // methods
    public int area() {
            return width * height;
    }
    …
```

Class vs Object

- The `Rectangle` class is a blueprint:

```
public class Rectangle {
    // fields
    private int width;
    private int height;

    // methods
    ...
}
```

- `Rectangle` objects are built according to that blueprint:

width	10
height	12

width	55
height	72

width	40
height	13

(You can also think of the methods as being inside the object, but we won't show them in our diagrams.)

Creating and Using an Object

- We create an object by using the `new` operator and a special method known as a *constructor*:

```
Rectangle r1 = new Rectangle(10, 30);
```

- Once an object is created, we can call one of its methods by using *dot notation*:

```
int a1 = r1.area();
```

- The object on which the method is invoked is known as the *called object* or the *current object.*

Two Types of Methods

- Methods that belong to an object are referred to as *instance methods* or *non-static methods*.
  - they are invoked on an object

    ```
    int a1 = r1.area();
    ```
  - they have access to the fields of the called object

- *Static* methods do *not* belong to an object – they belong to the class as a whole.
  - they have the keyword static in their header:

    ```
    public static int max(int num1, int num2) {
        …
    ```
  - they do *not* have access to the fields of the class
  - outside the class, they are invoked using the class name:

    ```
    int result = Math.max(5, 10);
    ```

Abstract Data Types

- An *abstract data type* (ADT) is a model of a data structure that specifies:
  - the characteristics of the collection of data
  - the operations that can be performed on the collection

- It's *abstract* because it doesn't specify *how* the ADT will be implemented.

- A given ADT can have multiple implementations.

A Simple Abstract Data Type: Bag

- A bag is just a container for a group of data items.
  - analogy: a bag of candy
- The positions of the data items don't matter (unlike a list).
  - {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

Bag

- The operations supported by our Bag ADT:
  - `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

- Note that we *don't* specify *how* the bag will be implemented.

Specifying an ADT Using an Interface

- In Java, we can use an interface to specify an ADT:

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

}
```

- An interface specifies a set of methods.
    - includes only the method headers
    - *cannot* include the actual method definitions

Implementing an ADT Using a Class

- To implement an ADT, we define a class:

```
public class ArrayBag implements Bag {
    private Object[] items;
    private int numItems;
    …
    public boolean add(Object item) {
        …

}
```

- When a class header includes an `implements` clause, the class must define all of the methods in the interface.

Encapsulation

- Our implementation provides proper *encapsulation*.
  - a key principle of object-oriented programming
  - also known as *information hiding*

- We prevent direct access to the internals of an object by making its fields *private*.

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

- We provide limited *indirect* access through methods that are labeled *public*.

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

All Interface Methods are Public

- Methods specified in an interface *must* be `public`, so we don't need to use the keyword `public` in the interface definition.

- For example:

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

- However, when we actually implement one of these methods in a class, we *do* need to explicitly use the keyword `public`:

```
public class ArrayBag implements Bag {
    …
    public boolean add(Object item) {
        …
```

Inheritance

- We can define a class that explicitly *extends* another class:

```
public class Animal {
    private String name;
    …
    public String getName() {
        return name;
    }
    …
}

public class Dog extends Animal {
    …
```

- We say that `Dog` is a *subclass* of `Animal`, and `Animal` is a *superclass* of `Dog`.

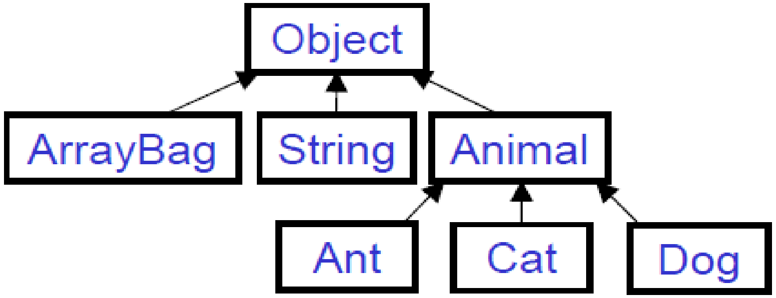
- A class *inherits* the instance variables and methods of the class that it extends.

The Object Class

- If a class does not explicitly extend another class, it implicitly extends Java's `Object` class.

- The `Object` class includes methods that all classes must possess. For example:
  - `toString()`: returns a string representation of the object
  - `equals()`: is this object equal to another object?

- The process of extending classes forms a hierarchy of classes, with the `Object` class at the top of the hierarchy:



Polymorphism

- An object can be used wherever an object of one of its superclasses is called for.
- For example:

```
Animal a = new Dog();

Animal[] zoo = new Animal[100];
zoo[0] = new Ant();
zoo[1] = new Cat();
…
```

- The name for this capability is *polymorphism.*
  - from the Greek for "many forms"
  - the same code can be used with objects of different types

Sorting Items in an ArrayBag

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

```
public class ArrayBag implements Bag {
    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 Double(3.1416));
```

Another Example of Polymorphism

- An interface name can be used as the type of a variable.

```
Bag b;
```

- Variables that have an interface type can hold references to objects of any class that implements the interface.

```
Bag b = new ArrayBag();
```

- Using a variable that has the interface as its type allows us to write code that works with any implementation of an ADT.

```
public void processBag(Bag b) {
    for (int i = 0; i < b.numItems(); i++) {
      …
}
```

  - the param can be an instance of *any* `Bag` implementation
  - we must use method calls to access the object's internals, because we can't know for certain what the field names are

Memory Management

- In order to understand the implementation of the data structures we'll cover in this course, you'll need to have a good understanding of how memory is managed.

- There are three main types of memory allocation in Java.

- They correspond to three different regions of memory.

MM: Type 1: Static Storage

- Static storage is used in Java for *class variables*, which are declared using the keyword `static`:

```
public static final PI = 3.1495;
public static int numCompares;
```

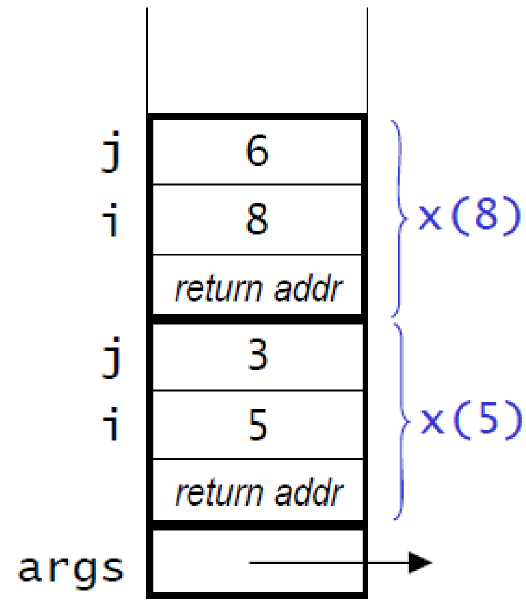
- There is only one copy of each class variable; it is shared by all instances (i.e., all objects) of the class.

- The Java runtime system allocates memory for class variables when the class is first encountered.
  - this memory stays fixed for the duration of the program

MM: Type 2: Stack Storage

- Method parameters and local variables are stored in a region of memory known as *the stack*.
- For each method call, a new *stack frame* is added to the top of the stack.

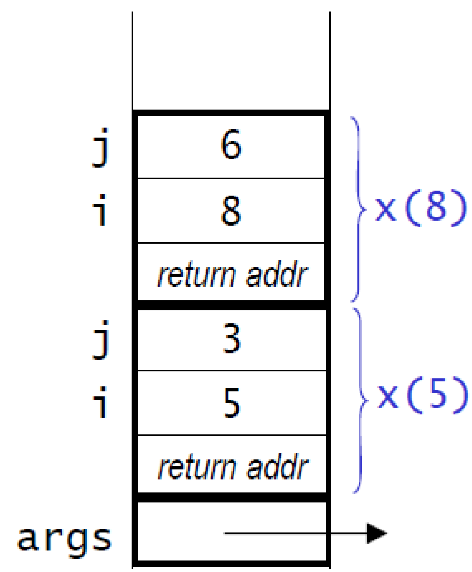
```
public class Foo {
  static void x(int i) {
      int j = i - 2;
      if (i >= 6) return;
      x(i + j);
  }
  public static void
    main(String[] args) {
      x(5);
  }
}
```



- When a method completes, its stack frame is removed. The values stored there are *not* preserved.

Stack Storage

- Memory allocation on the stack is very efficient, because there are only two simple operations:
  - add a stack frame to the top of the stack
  - remove a stack frame from the top of the stack
- Limitations of stack storage:
  It can't be used if
  - the amount of memory needed isn't known in advance
  - we need the memory to persist after the method completes
- Because of these limitations, Java never stores arrays or objects on the stack.

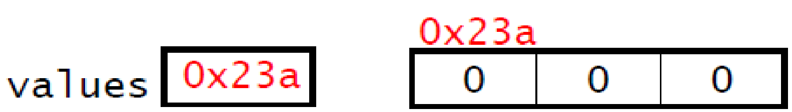


MM: Type 3: Heap Storage

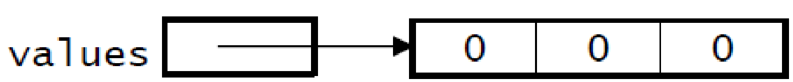
- Arrays and objects in Java are stored in a region of memory known as *the heap*.
- Memory on the heap is allocated using the new operator:

```
int[] values = new int[3];
ArrayBag b = new ArrayBag();
```

- `new` returns the memory address of the start of the array or object on the heap.
- This memory address – which is referred to as a *reference* in Java – is stored in the variable that represents the array/object:



- We will often use an arrow to represent a reference:



Heap Storage

- In Java, an object or array persists until there are no remaining references to it.
- You can explicitly drop a reference by setting the variable equal to `null`. For example:

```
int[] values = {5, 23, 61, 10};
System.out.println(mean(values, 4));
values = null;
```

- Unused objects/arrays are *automatically* reclaimed by a process known as garbage collection.
    - makes their memory available for other objects or arrays

Constructors for ArrayBag Class

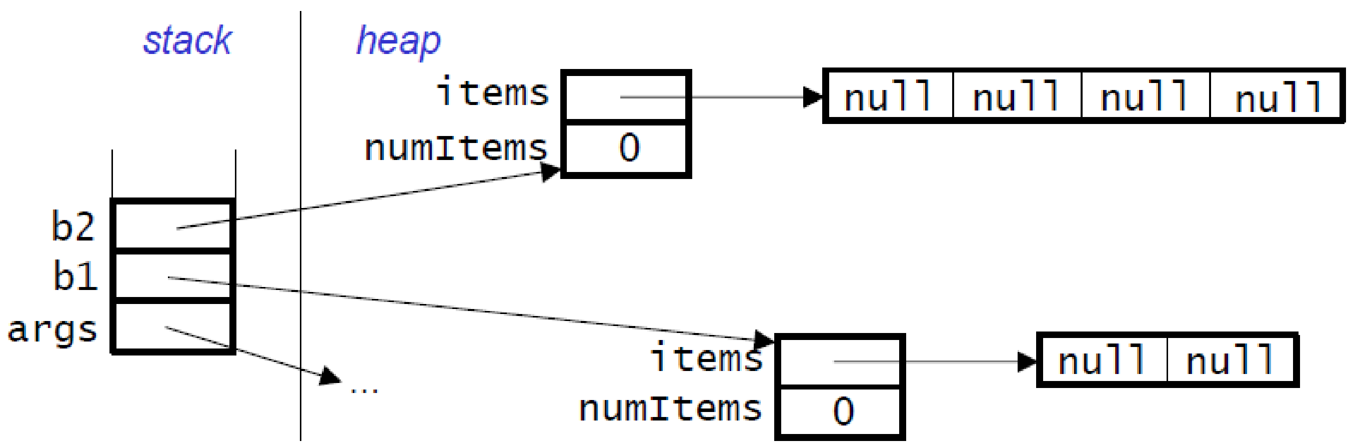
```
public class ArrayBag implements Bag {
    private Object[] items;
    private int numItems;
    public static final int DEFAULT_MAX_SIZE = 50;

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

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

Creating Two ArrayBag Objects

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

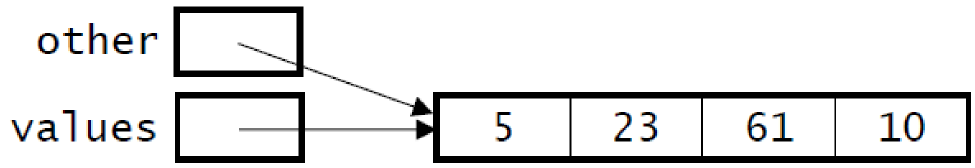
- After the objects have been created, here's what we have:



Copying References

- A variable that represents an array or object is known as a *reference variable*.
- Assigning the value of one reference variable to another reference variable copies the reference to the array or object. It does *not* copy the array or object itself.

```
int[] values = {5, 23, 61, 10};
int[] other = values;
```



- Given the lines above, what will the lines below output?

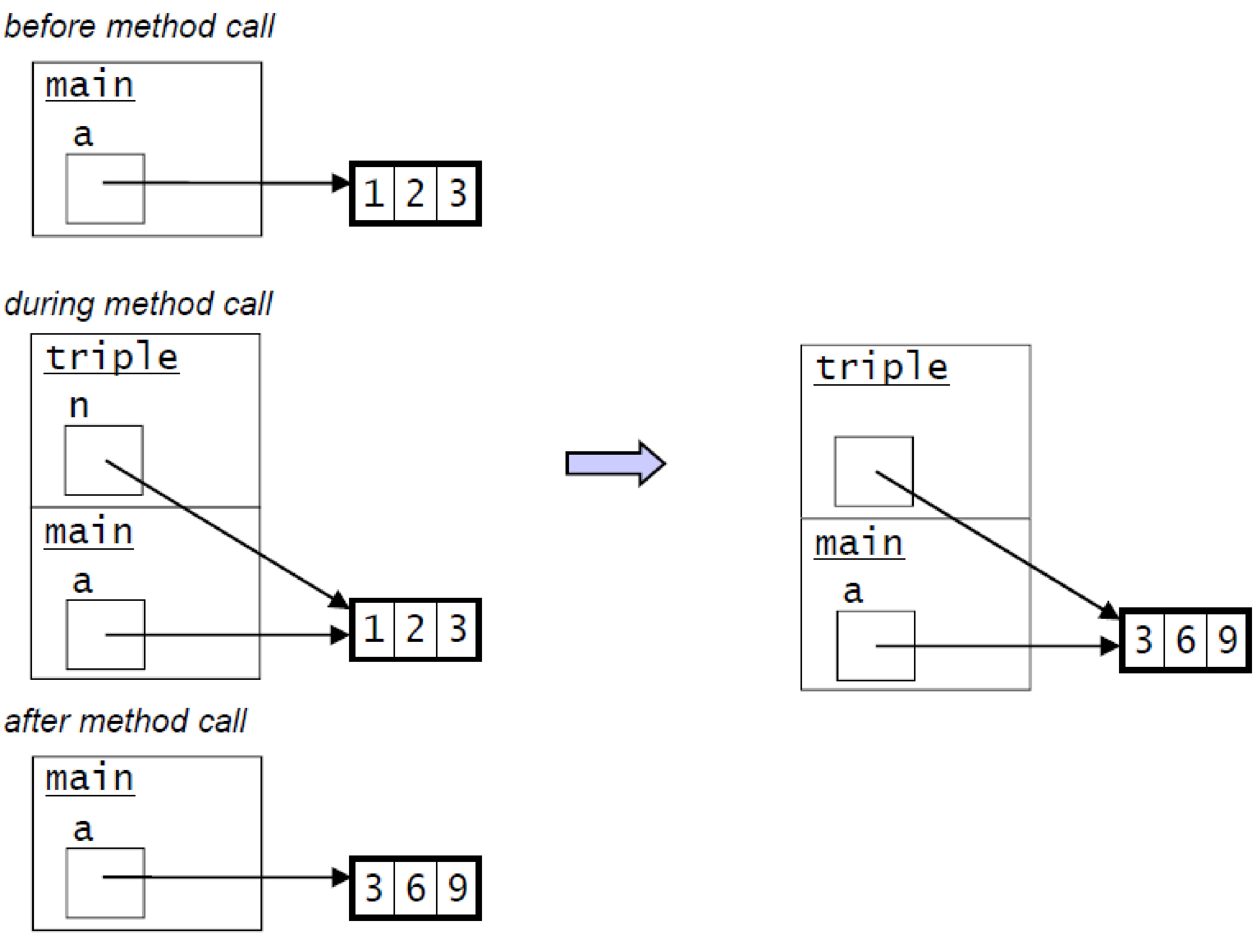
```
other[2] = 17;
System.out.println(values[2] + " " + other[2]);
```

Passing an Object/Array to a Methods

- When a method is passed an object or array as a parameter, the method gets a copy of the *reference* to the object or array, *not* a copy of the object or array itself.
- Thus, any changes that the method makes to the object/array will still be there when the method returns.
- Consider the following:

```
public static void main(String[] args) {
    int[] a = {1, 2, 3};
    triple(a);
    System.out.println(Arrays.toString(a));
}

public static void triple(int[] n) {
    for (int i = 0; i < n.length; i++) {
        n[i] = n[i] * 3;
    }
}
```

Passing an Object/Array to a Methods



A Method for Adding an Item to A Bag

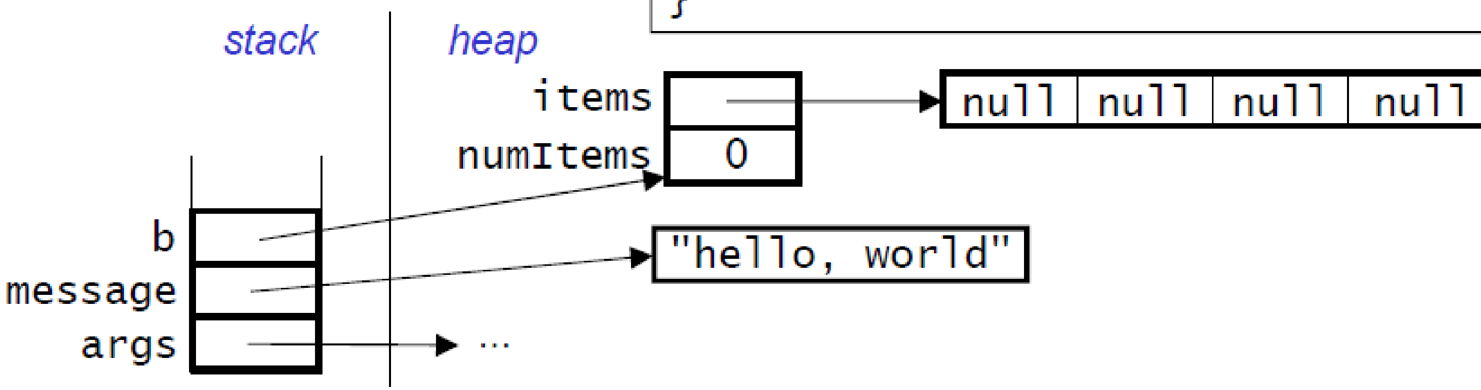
```
public class ArrayBag implements Bag {
    private Object[] items;
    private int numItems;
    …
    public boolean add(Object item) {
        if (item == null)
            throw new IllegalArgumentException();
        if (numItems == items.length)
            return false;  // no more room!
        else {
            items[numItems] = item;
            numItems++;
            return true;
        }
    }
    …
}
```

- `add()` is an instance method (a.k.a. a non-static method), so it has access to the fields of the current object.

Example: Adding an Item

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

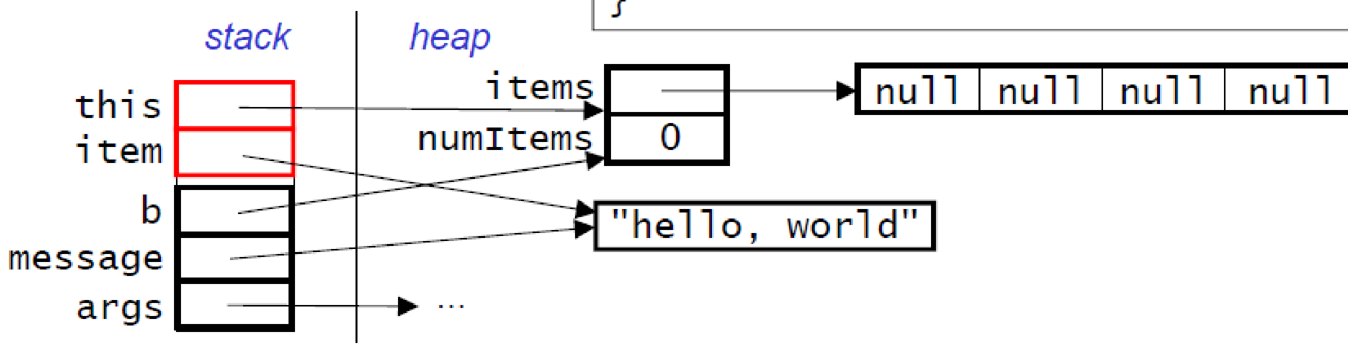
```
public boolean add(Object item) {
    …
    else {
        items[numItems] = item;
        numItems++;
        return true;
    }
}
```



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 {
        items[numItems] = item;
        numItems++;
        return true;
    }
}
```

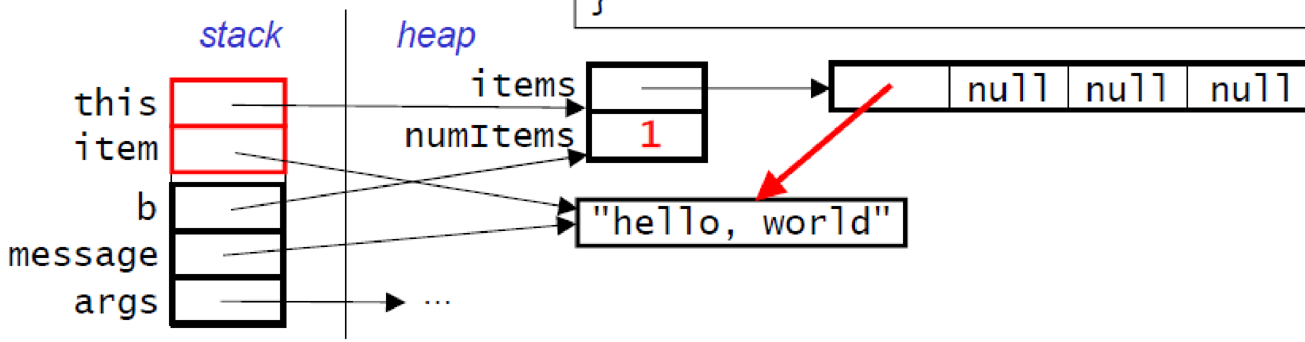


- add's stack frame includes:
  - `item`, which stores a copy of the reference passed as a param.
  - `this`, which stores a reference to the called/current object

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 {
        items[numItems] = item;
        numItems++;
        return true;
    }
}
```

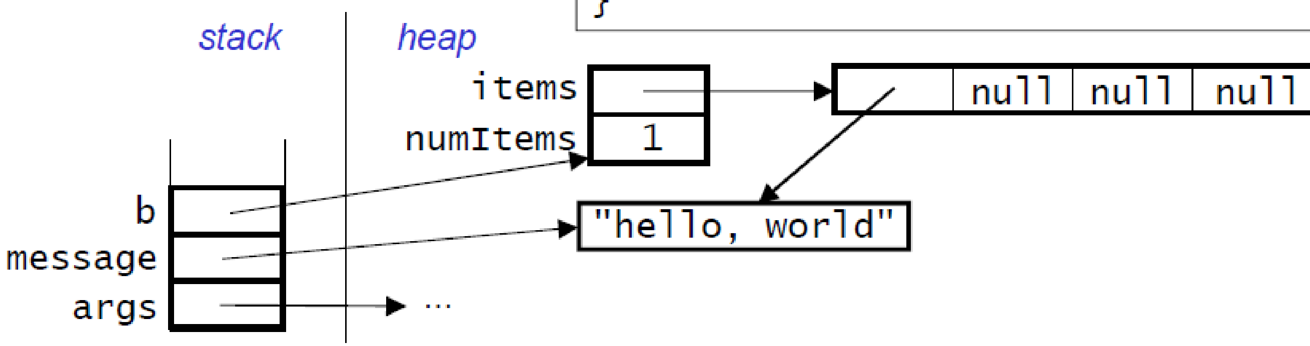


- The method modifies the `items` array and `numItems`.
  - note that the array holds a copy of the *reference* to the item, not a copy of the item itself.

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 {
        items[numItems] = item;
        numItems++;
        return true;
    }
}
```



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

Using the Implicit Parameter

```
public class ArrayBag implements Bag {
    private Object[] items;
    private int numItems;
    …
    public boolean add(Object item) {
        if (item == null)
            throw new IllegalArgumentException();
        if (this.numItems == this.items.length)
            return false;  // no more room!
        else {
            this.items[this.numItems] = item;
            this.numItems++;
            return true;
        }
    }
    …
}
```

- We can use `this` to emphasize the fact that we're accessing fields in the current object.

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) {
```

An incorrect version

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

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

- When would it work?

A Method That Takes a Bag As Parameter

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

```
public boolean containsAll(Bag otherBag) {
    if (otherBag == null || otherBag.numItems() == 0)
        return false;

    Object[] otherItems = otherBag.toArray();
    for (int i = 0; i < otherItems.length; i++) {
        if (!contains(otherItems[i]))
            return false;
    }

    return true;
}
```

- We use `Bag` instead of `ArrayBag` as the type of the parameter.
  - allows this method to be part of the `Bag` interface
  - allows us to pass in *any* object that implements `Bag`
- Because the parameter may not be an `ArrayBag`, we can't assume it has `items` and `numItems` fields.
  - instead, we use `toArray()` and `numItems()`

A Need for Casting

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

- Let's say that we want to store a collection of `String` objects in an `ArrayBag`.

- `String` is a subclass of `Object`, so we can store `String` objects in the bag without doing anything special:

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

- `Object` isn't a subclass of `String`, so this will <u>not</u> work:

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

- Instead, we need to use casting:

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