

Introduction: Course Overview, Java Review

Computer Science E-22/S-22
Harvard Extension School

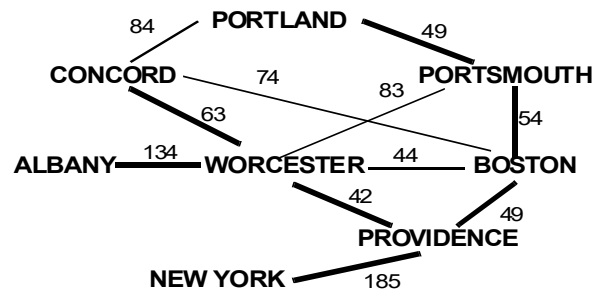
David G. Sullivan, Ph.D.

Welcome to Computer Science E-22/S-22!

- We will study fundamental *data structures*.
 - ways of imposing order on a collection of information
 - sequences: lists, stacks, and queues
 - trees
 - hash tables
 - graphs
- We will also:
 - study *algorithms* related to these data structures
 - learn how to *compare* data structures & algorithms
- Goals:
 - learn to think more intelligently about programming problems
 - acquire a set of useful tools and techniques

Sample Problem I: Finding Shortest Paths

- Given a set of routes between pairs of cities, determine the shortest path from city A to city B.



Sample Problem II: A Data "Dictionary"

- Given a large collection of data, how can we arrange it so that we can efficiently:
 - add a new item
 - search for an existing item
- Some data structures provide better performance than others for this application.
- More generally, we'll learn how to characterize the *efficiency* of different data structures and their associated algorithms.

Prerequisites

- A good working knowledge of Java
 - comfortable with object-oriented programming concepts
 - comfortable with arrays
 - some prior exposure to recursion would be helpful
 - if your skills are weak or rusty, you may want to consider first taking CSCI E-10b
- Reasonable comfort level with mathematical reasoning
 - mostly simple algebra, but need to understand the basics of logarithms (we'll review this)
 - will do some simple proofs

Requirements

- Lectures
- Sections
 - optional but highly recommended
 - start this week
 - on Zoom; one section will be recorded
- Five problem sets
 - plan on 10-20 hours per week!
 - code in Java
 - must be your own work
 - see syllabus for the collaboration policy
 - grad-credit students will do extra problems
- Midterm exam
- Final exam

Additional Administtrivia

- Instructor: Dave Sullivan
- TAs: see website mentioned below
- Office hours and contact info. will be available on the Web:
<https://cscie22.sites.fas.harvard.edu>
- For questions on content, homework, etc.:
 - use Ed Discussion on Canvas
 - send e-mail to cscie22-staff@lists.fas.harvard.edu

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:

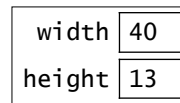
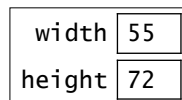
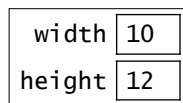
```
public class Rectangle {  
    // fields  
    private int width;  
    private int height;  
  
    // methods  
    public int getWidth() {  
        return this.width;  
    }  
    ...  
}
```

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:



(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 width1 = r1.getWidth();
```

- 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 width1 = r1.getwidth();
```
 - 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);
```

Encapsulation

- Our classes should provide proper *encapsulation*.
- We limit direct access to the internals of an object by making the fields *private*:

```
public class Rectangle {  
    private int width;  
    private int height;  
    ...  
}
```

 - private components of a class can only be accessed directly by code within the class itself
- We provide limited *indirect* access through methods that are labeled *public*.

```
public int getwidth() {  
    return this.width;  
}
```

 - public components can be accessed anywhere

Encapsulation (cont.)

- `getWidth()` is an *accessor* method that can be used to obtain information about an object, but *not* to change it:

```
public int getWidth() {  
    return this.width;  
}
```
- we use the keyword `this` to access the fields and methods that are inside the called object
- A class can also provide *mutator* methods that change the called object, but only in appropriate ways:

```
public void setWidth(int newWidth) {  
    if (newWidth <= 0) {  
        throw new IllegalArgumentException();  
    }  
    this.width = newWidth;  
}
```
- throwing an exception prevents an inappropriate change by ending the method prematurely

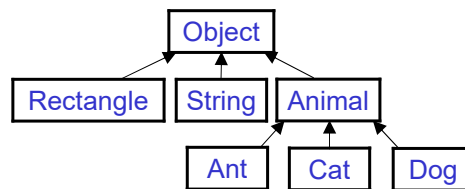
Inheritance

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

```
public class Animal {  
    private String name;  
    // other field definitions go here  
  
    public String getName() {  
        return this.name;  
    }  
    // other method definitions go here  
}  
  
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 fields 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

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

A Bag Data Structure (cont.)

- 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

Implementing a Bag Using a Class

- To implement our bag data structure, we define a class:

```
public class ArrayBag {  
    private Object[] items;  
    private int numItems;  
    // constructors go here  
    public boolean add(Object item) {  
        ...  
    }  
}
```

- Each object of this class will represent an entire bag of items.
- The items themselves are stored in an array of type Object.
 - allows us to store *any* type of object in the bag, thanks to the power of polymorphism:

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

Memory Management: Looking Under the Hood

- To understand how data structures are implemented, you need to understand how memory is managed.
- There are three main types of memory allocation in Java.
- They correspond to three different regions of memory.

Memory Management, Type I: Static Storage

- Static storage is used for *class variables*, which are declared *outside any method* using the keyword `static`:

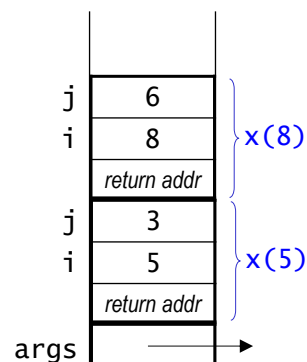
```
public class MyMethods {  
    public static int numCompares;  
    public static final double PI = 3.14159;  
}
```

- There is only one copy of each class variable.
 - shared by all objects of the class
 - Java's version of a global variable
- The Java runtime allocates memory for class variables when the class is first encountered.
 - this memory stays fixed for the duration of the program

Memory Management, Type II: 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 {  
    public static int x(int i) {  
        int j = i - 2;  
        if (i >= 6) {  
            return i;    // return 8  
        }  
        return x(i + j);  
    }  
    public static void  
    main(String[] args) {  
        System.out.println(x(5));  
    }  
}
```



- When a method completes, its stack frame is removed.

Memory Management, Type III: Heap Storage

- Objects (including arrays) 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 object.
- This memory address – which is referred to as a *reference* – is stored in the variable that represents the object:

values

0x23a

0	0	0
---	---	---

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

values

—

 →

0	0	0
---	---	---

Heap Storage (cont.)

- An object remains on the heap until there are no remaining references to it.
- Unused objects are automatically reclaimed by a process known as *garbage collection*.
 - makes their memory available for other objects

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) {
        ...
    }
}
```

- A class can have multiple 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;

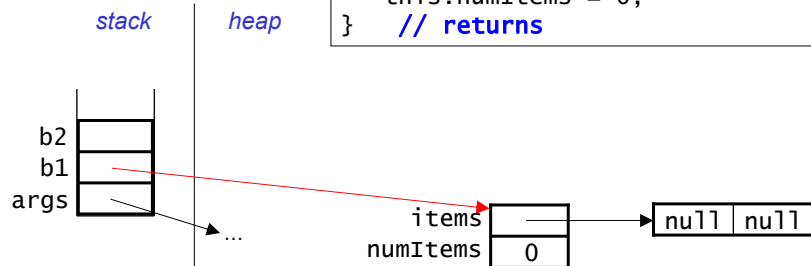
    public ArrayBag() {
        this.items = new Object[DEFAULT_MAX_SIZE];
        this.numItems = 0;
    }
    public ArrayBag(int maxSize) {
        if (maxSize <= 0) {
            throw new IllegalArgumentException(
                "maxSize must be > 0");
        }
        this.items = new Object[maxSize];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

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

Example: Creating Two ArrayBag Objects

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

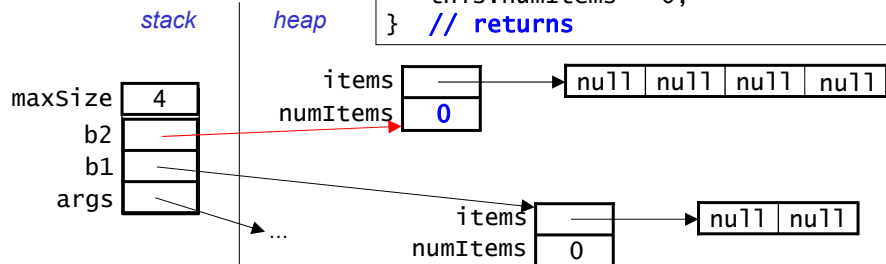
```
// constructor
public ArrayBag(int maxSize) {
    ... // error-checking
    this.items = new Object[maxSize];
    this.numItems = 0;
} // returns
```



Example: Creating Two ArrayBag Objects

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

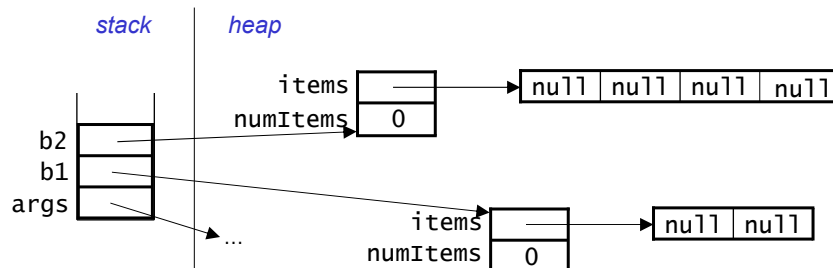
```
// constructor
public ArrayBag(int maxSize) {
    ... // error-checking
    this.items = new Object[maxSize];
    this.numItems = 0;
} // returns
```



Example: Creating Two ArrayBag Objects

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

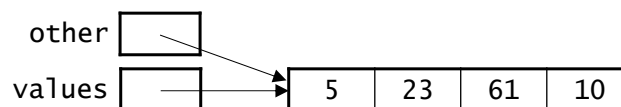
- After the objects have been created, here's what we have:



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 Method

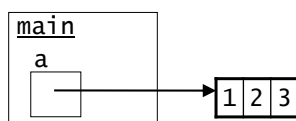
- When a method is passed an object or array as a parameter, the method gets a copy of the *reference* to the object or array, *not* a copy of the object or array itself.
- Thus, any changes that the method makes to the *internals* of the object/array will still be there when the method returns.
- Consider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

What is the output?

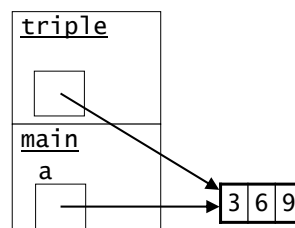
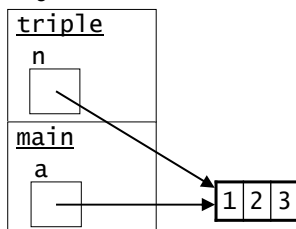
Passing an Object/Array to a Method (cont.)

before method call

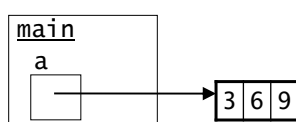


```
public static void main(String[] args) {  
    int[] a = {1, 2, 3};  
    triple(a);  
    System.out.println(...);  
}
```

during method call

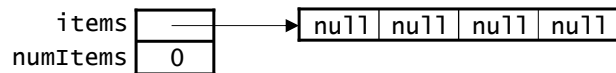


after method call

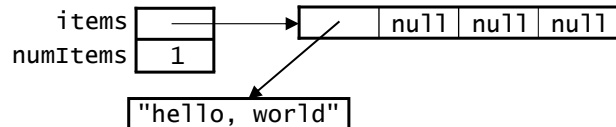


Adding Items to an ArrayBag

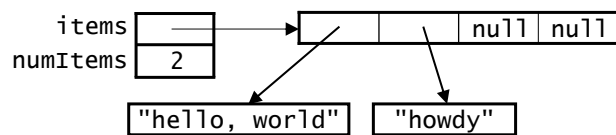
- We fill the array from left to right. Here's an empty bag:



- After adding the first item:

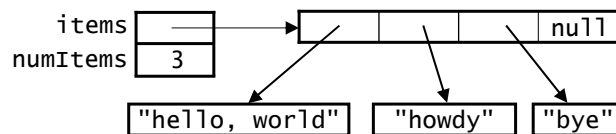


- After adding the second item:

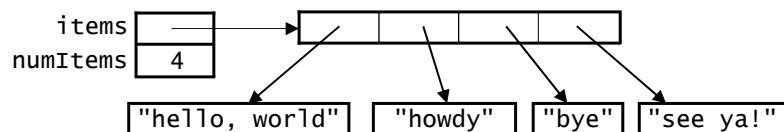


Adding Items (cont.)

- After adding the third item:



- After adding the fourth item:



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

A Method for Adding an Item to a Bag

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

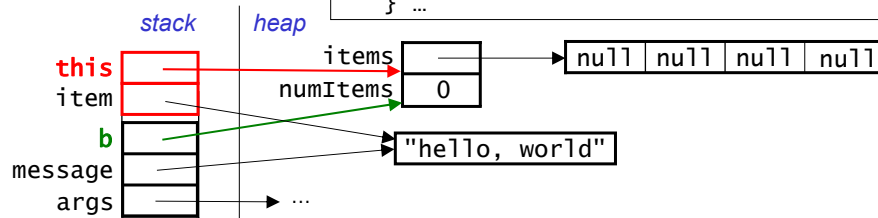
- takes an object of any type!
- returns a boolean to indicate if the operation succeeded

- Initially, `this.numItems` is 0, so the first item goes in position 0.
- We increase `this.numItems` because we now have 1 more item.
 - and so the *next* item added will go in the correct position!

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++;
        return true;
    } ...
}
```

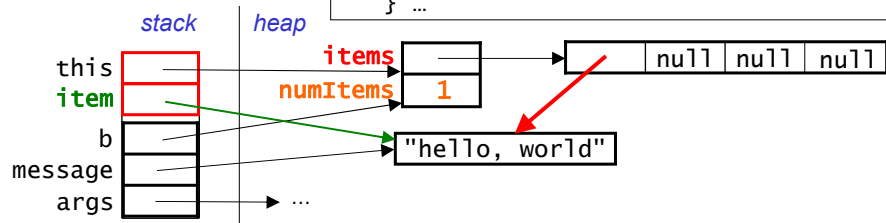


- `add`'s stack frame includes:
 - `item`, which stores a copy of the reference passed as a param.
 - `this`, which stores a reference to the called `ArrayBag` object

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

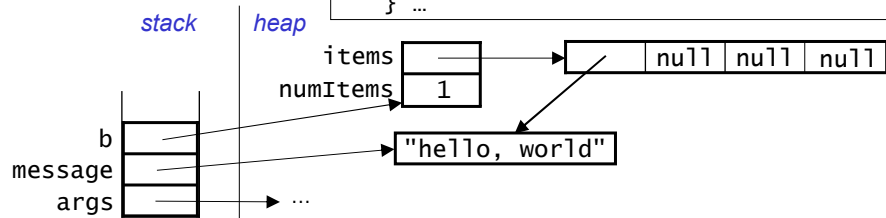


- The method modifies the `items` array and `numItems`.
 - note that the array holds a copy of the *reference* to the item, not a copy of the item itself.

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++;
        return true;
    } ...
}
```



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

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
```

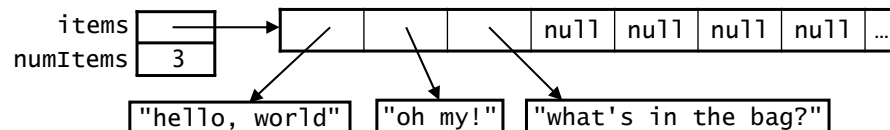
- the return type of grab() is Object
- Object isn't a subclass of String, so polymorphism doesn't help!

- 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

Extra Practice: Determining if a Bag Contains an Item

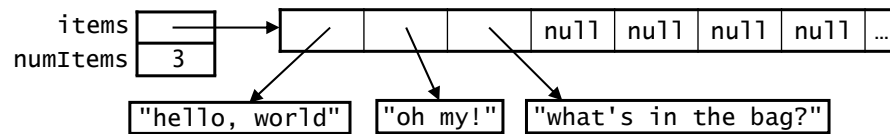


- Let's write the ArrayBag contains() method together.
- should return true if an object equal to item is found, and false otherwise.

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

```
}
```

Would this work instead?



- 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;  
}
```

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;  
}
```

- What's the problem with this?