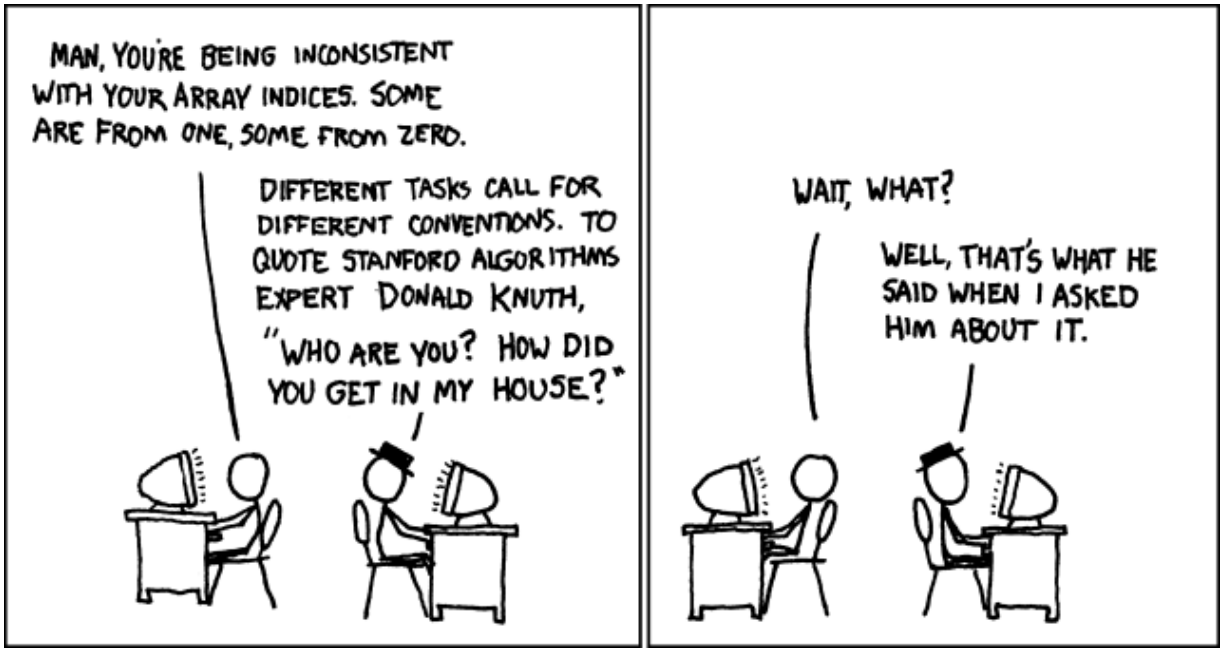
# ARRAY-BASED IMPLEMENTATIONS




# REVIEW OF ARRAYS

# Arrays

- An *array* is an ordered list of values.
- An array has a single name  and holds several values.
  - Each value has a numeric index.
  - An array of size *n* is indexed from 0 to *n-1*.
- The size of an array can be accessed with .length
- Arrays store elements of the same type.
  - Arrays can hold primitives or objects.

# Array Size

- The size of an array cannot be changed once it is set during initialization.
- You can change what is stored in the array, but you **cannot** change *how many* elements can be stored in that contiguous space in memory.

# Arrays are Objects

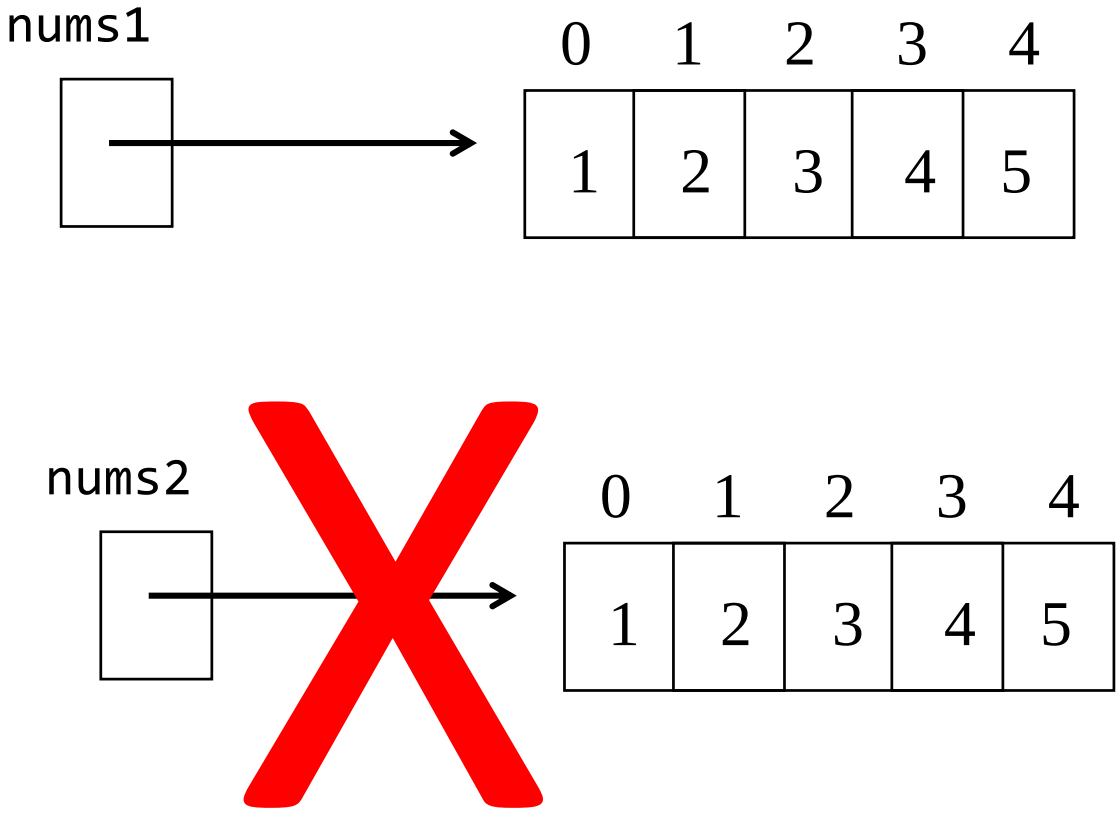
- Remember that everything in Java is either a primitive or an object.
- Arrays are objects!
  - The variable name of an array holds a *reference* or *pointer* to the place in memory where the elements are stored.
- Recall what this means about using direct assignment!

# Arrays are Objects

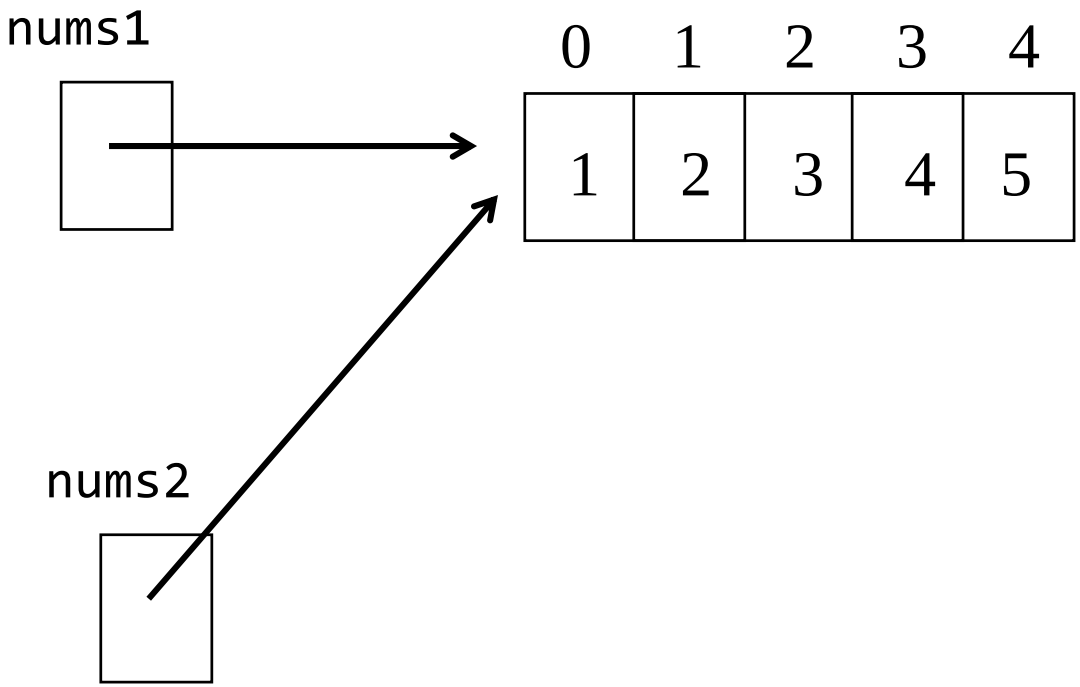
- What is stored in `nums1[0]` after this code?

```
int[] nums1 = {1, 2, 3, 4, 5};
int[] nums2 = nums1;
nums2[0] = 99;
```

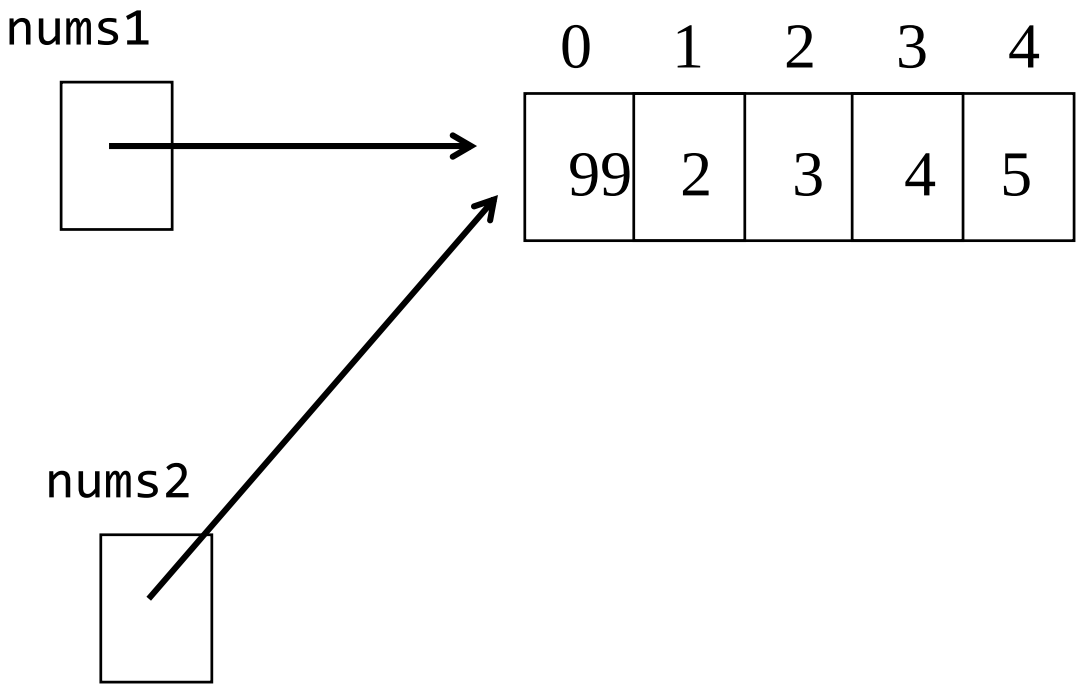
# Arrays as Objects



# Arrays as Objects



# Arrays as Objects



# Copying Arrays

- What is stored in `nums1[0]` after this code?

```
int[] nums1 = {1, 2, 3, 4, 5};
int[] nums2 = nums1;
nums2[0] = 99;
```

- 99 is stored there!
  - `nums1` and `nums2` aliases of each other.
  - Changes to one will affect changes to the other.
- To only affect one of these arrays, we need to make a *copy*:
  - Create a whole new array
  - Copy over *each* element in the array

# Array Size Revisited

- You cannot make an array bigger.
- You *can* create a *new, bigger* array and make your array reference point to it.
  - You need to manually copy over the data.

# Copying an array

```
int[] nums = {1, 2, 3};
```
original array

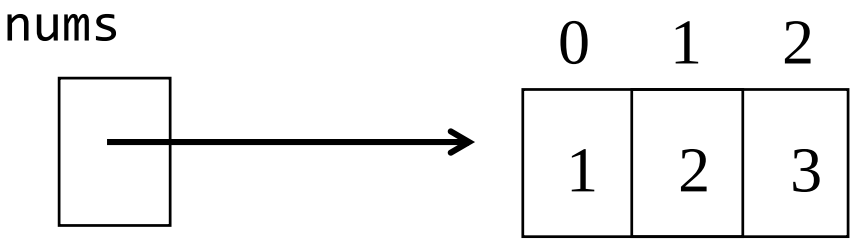
```
int[] moreNums = new int[5];
```
bigger, empty array

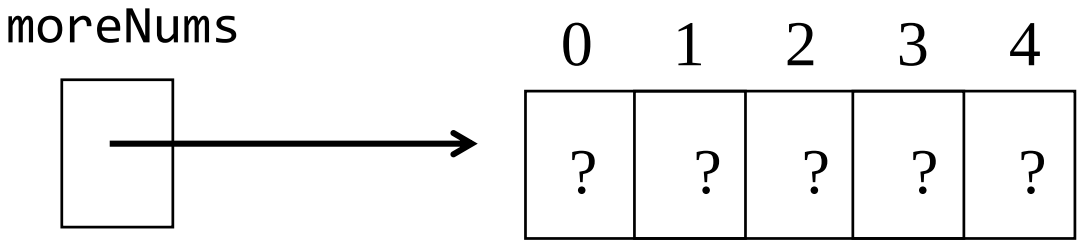
```
for(int i=0; i<nums.length; i++) {
  moreNums[i] = nums[i];
}
```
copy the old array contents into the new array

```
nums = moreNums;
```
point the reference to the new array

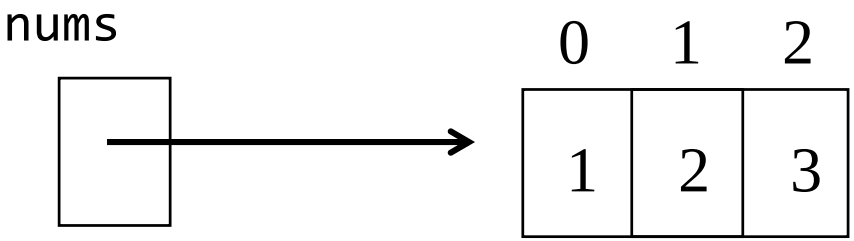
the old “nums” memory location is lost and those contents are garbage collected

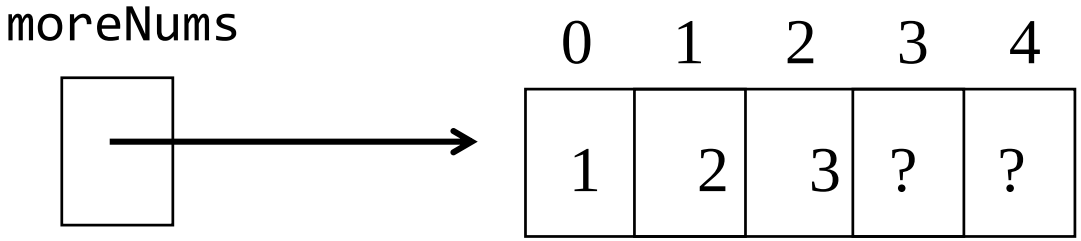
# Arrays as Objects



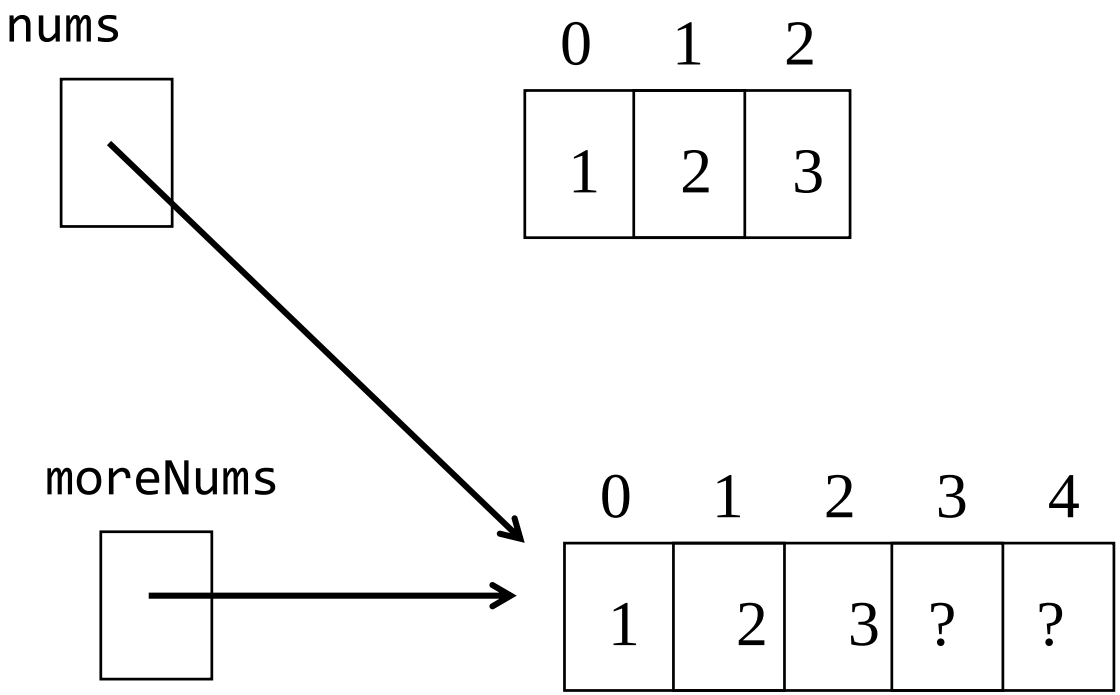



# Arrays as Objects



# Arrays as Objects



# Arrays as Parameters

- An entire array can be passed as a parameter to a method.
- Like any other object, the value is passed.
  - The value is the *reference* to the data in the array.
  - So, the formal and actual parameters (the two array references) become *aliases* of each other.
- This means that changing an array element within the method will also change that element in the original copy of the array.
- Review the example code.

# USING AN ARRAY TO IMPLEMENT A BAG

# BagInterface

- A bag is an unordered organization of items.
- Bags allow you to add items, remove an item, test if an item is in the bag, and determine the size of the bag.
- Bags allow duplicate values.

# Implementing a Bag

- We will now write a class to implement a bag.
- We need a data structure behind the scenes to store the elements in the bag.
  - An array
  - Linked nodes
  - A List!
  - Could be many other ways…
- We get to decide.

# Using an Array to Implement a Bag

- In any class, the instance data variables describe objects of that class.
  - private!
- For a bag, we'll use two variables:

  T[] bag; // hold the items

  int numberOfEntries; // how many items
- The constructor will initialize these variables.

# Overloading Constructors

- We often provide more than one constructor.
  - This allows different ways to set up an object.
  - Best practice is to invoke one constructor from the others when the “this” keyword whenever possible.
  - Usually, this means putting the code in the constructor that takes the most parameters and invoking that constructor from the others sending along parameters and/or default values.
- For the bag class, we can initialize the array based on a size sent by the user or using a default size.

# Creating a Generic Array

- Generics were added in Java 5 and sometimes the code to work with arrays and generics can get strange.
- Unfortunately, we cannot do this:
  - T[] bag = new T[size];
- The syntax instead is this:

```
T[] genericArray = (T[]) new Object[size];
```

# The Bag vs the Array

- In Java, all variables have a type.
- When you create an object of type ArrayBag, its type is ArrayBag.
- Each ArrayBag object has an array as part of what describes it.
  - The array is private and behind the scenes!
  - The array is used to implement the bag.

# The Bag vs the Array

```
BagInterface<Integer> numberBag =
        new ArrayBag<Integer>();
```

numberBag :

- declared type is BagInterface
- actual type is ArrayBag

numberBag.bag type:

- declared and actual type is an array []
- cannot be accessed outside the class!
- inside the class, access “bag” or “this.bag”
  - inside the class, “this” is an ArrayBag object; “this.bag” is an array

# The Bag vs the Array

- The array cannot be resized.
  - In the add method, you can see that if the array is full, you cannot add any more elements to the bag.
  - By using a traditional array as our behind the scenes data structure, we've also made our bag fixed in size.

# The Bag vs the Array

- Bags are unordered!
  - But our array is linear and does have an order!
- Inside the class, we can use the ordered array. But the outward-facing bag remains unordered.

# Adding and Removing

- Adding is straightforward.
  - Check if there is room and then add to the end of the array.
  - No shifting required- very efficient! O(1)
- Removing
  - remove(): the interface doesn't specify what element to remove, so we can decide
    - If we remove the last element, no shifting is required!
  - remove(T): find and remove the specified element
    - Since the bag is unordered, we don't need to shift elements to keep them in order!
    - We can just swap the last element into the place of the removed element! This is more efficient than shifting.
- Make sure to update both the array and numberOfEntries!

# Private Helper Methods

- When implementing an ADT, you'll often write private methods to help you manipulate the behind the scenes data structure.
- If you ever start to write code more than once or (gasp!) copy/paste code, stop!
  - It's almost always best to put this code inside a method and invoke it rather than duplicate code.

# ResizeableArrayBag

- A second class also uses an array to represent the bag, but allows the array to grow as needed.
  - This means the bag is not fixed in size!
  - Behind the scenes, when we need to make the array bigger, we create a bigger array, copy the contents, and update the instance data variable.

# ResizeableArrayBag Design Choices

- We will need to change the add method so that if the array is full, we can expand it.

- Do any other methods need to change?
- Do you notice a lot of repeated code?

# USING AN ARRAY TO IMPLEMENT A LIST

# List

- A list is an ordered, indexed organization of items.
- Lists allow you to add items, remove items, and retrieve items.

# Implementing a List

- We need a data structure behind the scenes to store the elements in the list.
  - An array
  - Linked nodes
  - Another kind of List!
  - Could be many other ways…
- We get to decide.

# Using an Array to Implement a List

- For a list, we'll use two variables:

  T[] list; // hold the items

  int numberOfEntries; // how many items
- The constructor will initialize these variables.

# The List vs the Array

```
ListInterface<Integer> numberList =
        new AList<Integer>();
```

numberList :

- declared type is ListInterface<Integer
- actual type is AList<Integer

numberList.list type:

- declared and actual type is an array []
- cannot be accessed outside the class!
- inside the class, access “list” or “this.list”
  - inside the class, “this” is an AList object; “this.list” is an array

# Adding and Removing

- Adding
  - add(T): add to the end
  - add(int, T): add at a specific position- shifting will be required!
- Removing
  - remove(int): order matters, so shifting is required!
- Make sure to update both the array and numberOfEntries!

# ListInterface Design Choices

- Should the list be fixed in size or expandable?
- ListInterface elements start at position 1.
  - Where should we store things in our list array?
- Private helper methods
  - We'll be shifting in multiple places…
  - We'll need to often check if indices are valid

# List and ArrayList

- The Java standard library provides a class that implements the List interface using an array behind the scenes- ArrayList
  - Expandable
  - Efficient!

# Programming to the Interface

- It’s often best to declare the type to be the most general type you need.

- List<Integer> numberList = new ArrayList<>();

# DEVELOPER PERSPECTIVE

# Writing Code as the Developer

- What behind-the-scenes data structures and other variables do I need to represent my ADT?
- How will I respond to unusual conditions?
- Check that I am:
  - Following the specifications described in the interface
  - Being consistent within my own code
  - Documenting design decisions
  - Reducing repeated code

# Respond to Unusual Conditions

- You have to account for a user not acting as you intend!
  - Removing from an empty bag
  - Adding to a full list
- You can:
  - Ignore it (not good!)
  - Return a value to indicate a problem (null, -1, false, etc.)
  - Throw an exception
- Sometimes these are specified by the interface, so you must follow that.