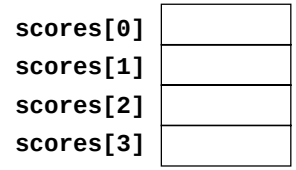
Chapter 3
Collection Classes

- **Java Arrays**
- **ADT for Bag of Integers**
  - Specification
  - `IntArrayBag` ADT
  - Bag ADT
  - Invariants
- **Interactive Testing Applet**
- **Sequence ADT**

Java Arrays

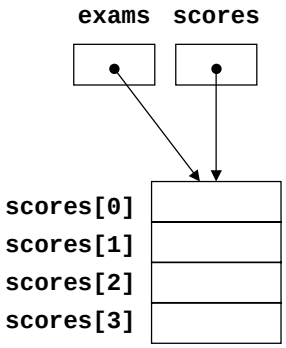
- **Contiguous locations of memory**
  - Same name & type.
- **Elements accessed via subscripts**
  - `scores[2]`
  - Index range is 0 to length – 1
  - `ArrayIndexOutOfBoundsException`
- **Length of an array**
  - Public data member, `length`.

    `scores.length`



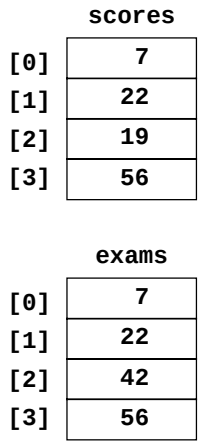
Java Arrays…

- **Declaration**
  - `int scores[] = new int[4];`
  - `int scores[];`
    `scores = new int[4];`
  - `int scores[] = {7, 22, 19, 56};`
- **Assignment**
  - `scores[0] = 7;`
  - `scores[1] = 22;`
  - `scores[2] = 19;`
  - `scores[3] = 56;`
- **Array as Object**
  - `int[] exams = scores;`



Java Arrays…

- **Clones**
  - `int exams[] =`
    `(int[]) scores.clone();`
  - `exams[2] = 42;`
- **Array Initialization**
  - Numeric primitive types: zero.
  - Boolean primitive types: **`false`**.
  - Object references: **`null`**.



Parameter Passing

- **Call-by-value**
  - Caller passes a copy of a parameter to the called method.
  - Changes are not reflected in the original parameter.
- **Call-by-reference**
  - Caller passes a reference to the parameter to the called method.
  - Changes made by the called method are seen by the caller.
- **Java Policy**
  - Primitive types are passed by value—scalar quantities.
  - Object references, arrays, etc., are passed by reference.

Java Arrays…

Array Parameter Example

- **`put42s` Method**

  `public static` *`void`* `put42s (`*`int[]`* `data)`
  - Put 42 in all elements of the array.
- **Parameters**

  `data`—an array of integers.
- **Postcondition**
  - All elements of array `data` have been set to 42.
- **Implementation**

```
public static void puts42
    (int[ ] data) {
  int i;
  for (i = 0;
       i < data.length;
       i++) {
    data[i] = 42;
  }
}
```

- **Using Parameters**

```
int[] example = new int[7];
put42s(example);
```


Bag of Integers

- **Collections**
  - Objects that can hold a group of items.
  - Implemented as a class.
  - Class defines methods to add, remove, & examine items.
- **Bag of Integers**
  - Holds a collection of integers.
  - First implementation uses integer array.
  - Other implementations use linked lists…

Bag of Integers…

- **Creation**
  - Bag starts empty
    - *Initial* state.
- **Modifying Operations**
  - Add number to a bag.
    - Allow multiple copies.
  - Remove number from a bag.
    - If number is present in bag!
    - Only one copy removed if more than one copy in bag.
- **Queries**
  - How many occurrences of a given number?
  - How many numbers in bag?

Bag ADT—Specification

- **Implementation**
  - To be decided!
- **Class Definition**
  - Class header.
  - Constructor(s).
  - Public methods.
- **Public Interface**
  - Specification defines public interface.
  - Can use Bag ADT without knowing implementation details.

Bag ADT—Specification…

- **Class Name**
  - Implemented as class `IntArrayBag`.
- **Constructors**
  - `public IntArrayBag(int initialCapacity)`
    - Create a bag with an initial capacity.
    - Capacity is automatically increased when capacity is reached.
  - `public IntArrayBag()`
    - Initial capacity of 10.

Bag ADT—Specification…

- **Modification Methods**
  - `public void add(int element)`
    - Add the integer, element, to the bag.
    - A new copy is added if already present.
  - `public boolean remove(int target)`
    - Remove the number *target* from the bag if present.
    - Return *true* if number present, *false* otherwise.
  - `public void addAll(IntArrayBag addend)`
    - Add contents of a bag to existing contents of this bag.

Bag ADT—Specification…

- **Query Methods…**
  - `public int getCapacity()`
    - Returns capacity of current bag.
- **Modification Methods…**
  - `public void ensureCapacity(int minimumCapacity)`
    - Increases capacity of bag to a minimum.
  - `public void trimToSize()`
    - Reduces capacity of bag to number of items currently in bag.
    - Minimizes memory usage.

Bag ADT—Specification…

- **Bag Producing Methods**
  - `public static IntArrayBag union (IntArrayBag b1, IntArrayBag b2)`
    - Returns new bag containing numbers in both bags.
  - `public Object clone()`
    - Returns a new bag.
    - Contents same as contents of this bag.

Bag ADT—Specification…

- **Query Methods**
  - `public int size()`
    - Returns number of integers in bag.
  - `public int countOccurrences(int target)`
    - Determines number of times integer *target* appears in bag.
  - `public int getCapacity()`
    - Returns current capacity of bag.

Bag ADT—Code Examples

- **Bag Creation & Adding Elements**

```
                                    // New bag is empty.
IntArrayBag firstBag = new IntArrayBag();
firstBag.add(8);    // firstBag has 1 element: 8.
firstBag.add(4);    // firstBag has 2 elements: 8 & 4.
firstBag.add(8);    // firstBag has 3 elements: Two 8s & one 4.
```

- **At end, Bag `firstBag` contains 3 elements.**
  - Two 8s and one 4.

Bag ADT—Code Examples…

- **Query Methods**

```
System.out.println(firstBag.countOccurrences(1));
System.out.println(firstBag.countOccurrences(4));
System.out.println(firstBag.countOccurrences(8));
```

- **Output**
  - `0`
  - `1`
  - `2`

Bag ADT—Code Examples…

- **Bag Creation & Adding Elements…**

```
IntArrayBag helter  = new IntArrayBag();
IntArrayBag skelter = new IntArrayBag();
helter.add(8);
skelter.add(4);
skelter.add(8);
                  // skelter has two elements: 4 & 8.
                  // helter has one element: 8.
helter.addAll(skelter);
                  // helter has three elements: two 8s & one 4.
```


Bag ADT—Code Examples…

- **Bag Creation…**

```
IntArrayBag part1 = new IntArrayBag();
IntArrayBag part2 = new IntArrayBag();
part1.add(8);
part1.add(9);     // Two elements in part1: 8 & 9.
part2.add(4);
part2.add(8);     // Two elements in part2: 4 & 8.
IntArrayBag total = IntArrayBag.union(part1, part2);
                  // 4 elements in total: 4, 9, & two 8s.
```

- **Equivalent statements**

```
IntArrayBag total = new IntArrayBag();
total.addAll(part1); total.addAll(part2);
```


Bag ADT—Code Examples…

- **Bag Creation…**

```
IntArrayBag b = new IntArrayBag();
b.add(42);
IntArrayBag c = (IntArrayBag) b.clone();
```

- Bag b is different from bag c.
- Bags b & c each contain one integer: 42.

Bag ADT—Specification…

- **Out of Memory Limitation**
  - Store integers in an array.
  - Allocate larger array when more capacity is needed.
  - Array allocated on heap.
  - Exception `OutOfMemoryError` indicates insufficient memory.
- **Other Limitations**
  - Array indexed by integers.
  - Java integers limited to `Integer.MAX_VALUE`.
  - Machines normally run out of memory before exceeding integer limit.

Class `IntArrayBag`—Specification…

- **Class `IntArrayBag`**
  - An `IntArrayBag` is a collection of int numbers. The same number may appear multiple times in a bag.
  - **Limitations**
    1. Bag capacity can change after creation, but maximum capacity is limited by machine memory. The constructor, addItem, clone, and union throw an `OutOfMemoryError` when memory is exhausted.
    2. A bag's capacity cannot exceed the maximum integer 2,147,483,647 (`Integer.MAX_VALUE`). Any attempt to create a larger capacity results in a failure due to an arithmetic overflow.
    3. Because of the slow linear algorithms of this class, large bags will have poor performance.

Class `IntArrayBag`—Specification…

- **First Constructor**

  `public` `IntArrayBag()`
  - Initialize an empty bag with an initial capacity of 10. Note that the `add` method works efficiently (without needing more memory) until this capacity is reached.
  - **Parameters**

    none .
  - **Postcondition**

    This bag is empty and has an initial capacity of 10.
  - **Throws:** `OutOfMemoryError`

    Indicates insufficient memory for: `new int[10]`.

Class `IntArrayBag`—Specification…

- **Second Constructor**

  `public` `IntArrayBag(`*`int`* `initialCapacity)`
  - Initialize empty bag with a specified initial capacity. The `add` method works efficiently (without needing more memory) until this capacity is reached.
  - **Parameters**

    `initialCapacity`—initial capacity of this bag.
  - **Precondition**

    `initialCapacity` is non-negative.
  - **Postcondition**

    This bag is empty and has a capacity of `initialCapacity`.
  - **Throws:**

    `IllegalArgumentException`—initialCapacity is negative.

    OutOfMemoryError—insufficient memory for:
    `new int[initialCapacity]`.

Class `IntArrayBag`—Specification…

- **`add` Method**

  `public` *`void`* `add(`*`int`* `element)`
  - Add a new element to this bag. Increase capacity of bag before adding element if need be.
  - **Parameters**

    `element`—new element that is being added.
  - **Postcondition**

    A new copy of the element has been added to this bag.
  - **Throws:**

    `OutOfMemoryError`—insufficient memory for increasing the capacity.
  - **Note**

    Capacity over `Integer.MAX_VALUE` generates arithmetic overflow.

Class `IntArrayBag`—Specification…

- **`addAll` Method**

  `public` *`void`* `addAll(IntArrayBag addend)`
  - Add the contents of another bag to this bag.
  - **Parameters**
    `addend`—a bag whose contents are being added to this bag.
  - **Precondition**
    The parameter, `addend`, is not null.
  - **Postcondition**
    The elements from `addend` have been added to this bag.
  - **Throws:**
    `NullPointerException`—parameter `addend` is null.
    `OutOfMemoryError`—insufficient memory for increasing the capacity.
  - **Note**
    Capacity over `Integer.MAX_VALUE` generates arithmetic overflow.

Class `IntArrayBag`—Specification…

- **`clone` Method**

  `public` *`Object`* `clone()`
  - Generate a copy of this bag.
  - **Returns**

    A copy of this bag. Changes to copy will not affect this bag or vice versa. Return value must be typecast to an `IntArrayBag` before used.
  - **Throws**
    `OutOfMemoryError`—insufficient memory for creating clone.

Class `IntArrayBag`—Specification…

- **`countOccurrences` Method**

  `public` *`int`* `countOccurrences(int target)`
  - Accessor method to count occurrences of a particular element in this bag.
  - **Parameters**
    `target`—the element that needs to be counted.
  - **Returns**

    The number of times that target occurs in this bag.

Class `IntArrayBag`—Specification…

- **`ensureCapacity` Method**

  `public` *`void`* `ensureCapacity(int minumumCapacity)`
  - Change the current capacity of this bag.
  - **Parameters**
    `minumumCapacity`—the new capacity for this bag.
  - **Postcondition**

    This bag's capacity has been changed to at least `minumumCapacity`. Capacity is unchanged if already greater than parameter.
  - **Throws**
    `OutOfMemoryError`—indicates insufficient memory for:
    `new int[minimumCapacity]`.

Class `IntArrayBag`—Specification…

- **`getCapacity` Method**

  `public` *`int`* `getCapacity()`

  - Accessor method that determines the current capacity of this bag. The `add` method works efficiently (w/o needing more memory) until this capacity is reached.
  - **Returns**

    the current capacity of this bag.

Class `IntArrayBag`—Specification…

- **`remove` Method**

  `public` *`boolean`* `remove(`*`int`* `target)`

  - Remove one copy of a specified element from this bag.
  - **Parameters**

    `target`—the element to remove from this bag.
  - **Postcondition**

    If `target` was found in this bag, then one copy has been removed and the method returns `true`. Otherwise, this bag remains unchanged and the method returns `false`.

Class `IntArrayBag`—Specification…

- **`size` Method**

  `public` *`int`* `size()`

  - Accessor method to determine the number of elements in this bag.
  - **Returns**

    the number of elements in this bag.

Class `IntArrayBag`—Specification…

- **`trimToSize` Method**

  `public` *`void`* `trimToSize()`

  - Reduce the current capacity of this bag to the number of elements in it.
  - **Postcondition**

    This bag's capacity has been changed to its actual size.
  - **Throws**

    `OutOfMemoryError`—indicates insufficient memory for altering capacity.

Class `IntArrayBag`—Specification…

- **`union` Method**

  ```
  public static IntArrayBag union
    (IntArrayBag b1, IntArrayBag b2)
  ```

  - Create a new bag that contains all the elements from two other bags.
  - **Parameters**
    `b1`—the first of two bags.
    `b2`—the second of two bags.
  - **Precondition**
    Neither `b1` nor `b2` is null.
  - **Returns**
    a new bag that is the union of `b1` and `b2`.
  - **Throws**
    `NullPointerException`—indicates that of the arguments is null.
    `OutOfMemoryError`—indicates insufficient memory for the new bag.
  - **Note**
    Capacity over `Integer.MAX_VALUE` generates arithmetic overflow.

Class `IntArrayBag`—Demo

- **Program Task**
  - User types ages with a negative *sentinel*; program enters ages into a bag.
  - User types an age which the program removes from the bag.
- **Session**

```
% java BagDemonstration
Type the ages of your family members.
Type a negative number at the end and press return.
5   19    47   -1
Type those ages again. Press return after each age.
Age: 19
Yes, I've got that age and will remove it.
Age: 36
No, that age does not occur!
Age: 5
Yes, I've got that age and will remove it.
Age: 47
Yes, I've got that age and will remove it.
May your family live long and prosper.
%
```


Class `BagDemonstration`

```
// This small demonstration program shows how to use the
// IntArrayBag class from the edu.colorado.collections package.
import edu.colorado.collections.IntArrayBag;
import edu.colorado.io.EasyReader;
class BagDemonstration
{
  private static EasyReader stdin = new EasyReader(System.in);

  public static void main(String[ ] args)
  {
    IntArrayBag ages = new IntArrayBag( );
    getAges(ages);
    checkAges(ages);
    System.out.println("May your family live long and prosper.");
  }
...
}
```


Class `BagDemonstration`...

```
// The getAges method prompts the user to type in the ages of family members. These ages
// are read and placed in the ages bag, stopping when the user types a negative number.
// This program does not worry about the possibility of running out of memory.
public static void getAges(IntArrayBag ages)
{
  int userInput;    // An age from the user's family

  System.out.println("Type the ages of your family members.");
  System.out.println
    ("Type a negative number at the end and press return.");
  userInput = stdin.intInput( );
  while (userInput >= 0)
  {
    ages.add(userInput);
    userInput = stdin.intInput( );
  }
}
```

Class `BagDemonstration`...

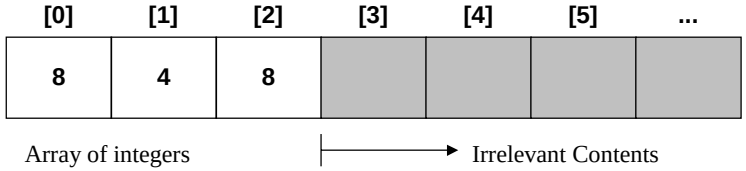
```
//The checkAges method prompts the user to type in the ages of family members once
// again. Each age is removed from the ages bag when it is typed, stopping when the bag is
// empty (or the user types a negative number).
public static void checkAges(IntArrayBag ages)
{
  int userInput;       // An age from the user's family
  System.out.print("Type those ages again. ");
  System.out.println("Press return after each age.");
  while (ages.size( ) > 0) {
    userInput = stdin.intQuery("Age: ");
    if (ages.countOccurrences(userInput) == 0)
       System.out.println("No, that age does not occur!");
    else {
      System.out.println("Yes, I've got that age and will remove it.");
      ages.remove(userInput);
    }
  }
}
```


Class BagDemonstration...

- **See `BagDemonstration.java` listing**

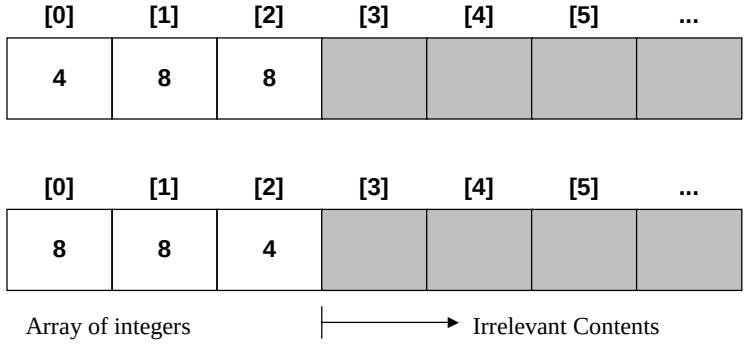
Class `IntArrayBag`—Design

- **Basic Structure**
  - Store numbers in front part of array.
  - Order is irrelevant.
- **Example**
  - Bag with three entries: two 4s & one 8.



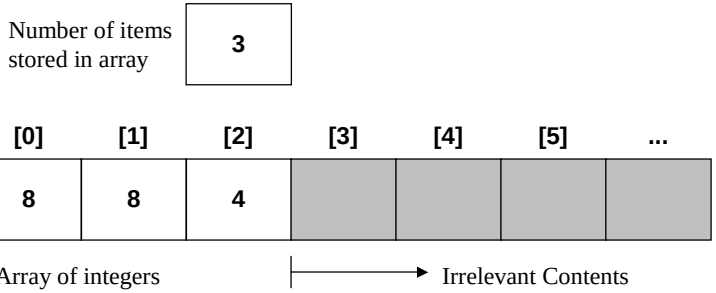
Class `IntArrayBag`—Design...

- **Example…**
  - Valid storage schemes.



Class `IntArrayBag`—Design…

- **Basic Structure…**
  - Store numbers in array.
  - Keep track of number of items in array.



Class `IntArrayBag`—Design…

- **Basic Class Definition**

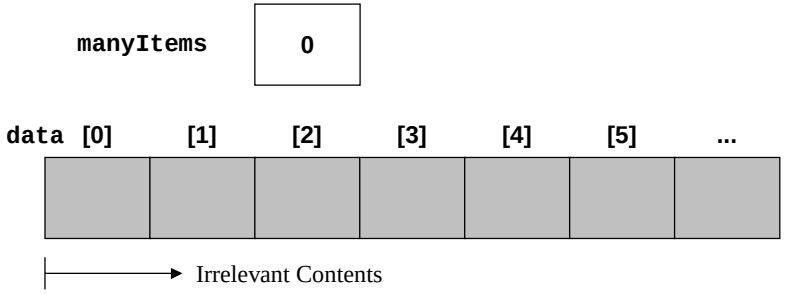
```
public class IntArrayBag implements Cloneable
{
  private int[] data;      // An array to store elements
  private int manyItems;   // How much of the array is used?

  // public methods go here.

}
```


Invariant of an ADT

- **Definition**
  - Rules governing the use of instance variables by methods.
  - Invariant is valid when a method is invoked.
    - Constructors are the exception.
  - Invariant is valid when a method call completes.
  - Document in class declaration.
- **Example—Class `IntArrayBag`**
  - Number of elements is stored in instance variable `manyItems`.
  - Numbers stored in `data[0]` through `data[manyItems – 1]`.
  - For *empty* bag, do not care about items stored in array `data`.

Class `IntArrayBag`—Design…

- **Constructor**
  - Initializes instance variables to represent an *empty* bag.
  - Array size is initial capacity of bag.



Class `IntArrayBag`—Design…

- **Constructor—Implementation**

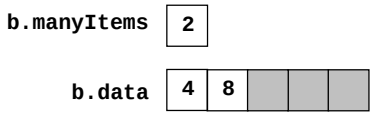
```
public IntArrayBag(int initialCapacity)
{
  if (initialCapacity < 0)
    throw new IllegalArgumentException
    ("initialCapacity is –ve: " + initialCapacity");

  manyItems = 0;
  data = new int[initialCapacity];
}
```


Class `IntArrayBag`—Design…

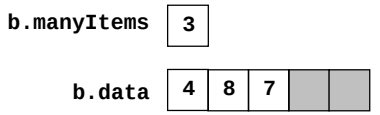
- **`add` Method**
  - Check capacity & double existing capacity if needed.
  - Store number in next free slot.
  - Increment element count.

```
public void add(int element)
b.add(42);
```

- **Before**



- **After**



Class `IntArrayBag`—Design…

- **`add` Method—Implementation**

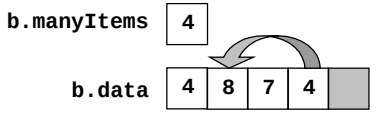
```
public void add(int element)
{
  if (manyItems == data.length)
  {
    // Double the capacity and add 1: this works even if manyItems is 0.
    // However, in the case that manyItems*2+1 is beyond
    // Integer.MAX_VALUE, there will be an arithmetic overflow and
    // the bag will fail.
    ensureCapacity(manyItems * 2 + 1);
  }
  data[manyItems++] = element;
}
```


Class `IntArrayBag`—Design…

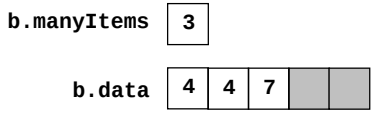
- **`remove` Method**
  - Find index of target value in array.
  - Store item in last slot of array in target slot.
  - Decrement element count.

```
public boolean remove(int element)
b.remove(8);
```

- **Before**



- **After**



Class `IntArrayBag`—Design…

- **`remove` Method**

```
public boolean remove(int target) {
  int  index;        // Location of target in data array.
  // First, set index to location of target in data array; range is 0 to manyItems – 1;
  // If target is not in array, index will be set to manyItems.
  for (index = 0;
       (index < manyItems) && (target != data[index]);
       index++)
    ; // Do nothing
  if (index == manyItems)
    return false;   // Target not found; nothing is removed.
  else {
    manyItems--;
    data[index] = data[manyItems];
    return true;
  }
}
```


Class `IntArrayBag`—Design…

- **`countOccurrences` Method**

```
public int countOccurrences(int target)
{
  int answer;
  int index;

  answer = 0;
  for (index = 0; index < manyItems; index++)
  {
    if (target == data[index])
      answer++;
  }
  return answer;
}
```


Class `IntArrayBag`—Design…

- **`addAll` Method**
  - Ensure capacity is big enough.
  - Copy items from `addend.data` to bag's own `data` array.
  - Update number of items in bag.

```
public void addAll(IntArrayBag addend)
{
  // If addend is null, a NullPointerException is thrown.
  // If total number of items > Integer.MAX_VALUE, overflow failure.
  ensureCapacity(manyItems + addend.manyItems);

  System.arrayCopy(addend.data, 0, data, manyItems,
                   addend.manyItems);
  manyItems += addend.manyItems;
}
```


Class `IntArrayBag`—Design…

- **`union` Method**

```
      IntArrayBag b3 = IntArrayBag.union(b1, b2);
public static IntArrayBag(IntArrayBag b1,
                          IntArrayBag b2) {
  // Throw NullPointerException if b1 or b2 are null.
  // Failure by arithmetic overflow if total number of items is too big.
  IntArrayBag answer = new
     IntArrayBag(b1.getCapacity() + b2.getCapacity());
  System.arrayCopy(b1.data, 0, answer.data, 0,
                   b1.manyItems);
  System.arrayCopy(b2.data, 0, answer.data,
                   b1.manyItems, b2.manyItems);
  answer.manyItems = b1.manyItems + b2.manyItems;
  return answer;
}
```

Class `IntArrayBag`—Design…

- **`ensureCapacity` Method**

```
public void ensureCapacity(int minimumCapacity)
{
  int[] biggerArray;

  if (data.length < minimumCapacity)
  {
    biggerArray = new int[minimumCapacity];
    System.arrayCopy(data, 0, biggerArray, 0,
                     manyItems);
    data = biggerArray;
  }
}
```


Class `IntArrayBag`—Design…

- **`getCapacity` Method**

```
public int getCapacity()
{
  return data.length;
}
```

- **`size` Method**

```
public int size()
{
  return manyItems;
}
```

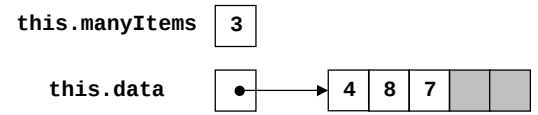

Class `IntArrayBag`—Design…

- **`trimToSize` Method**

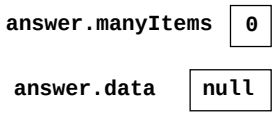
```
public void trimToSize()
{
  int trimmedArray;

  if (data.length != manyItems)
  {
    trimmedArray = new int[manyItems];
    System.arrayCopy(data, 0, trimmedArray, 0,
                     manyItems);
  }
}
```


Class `IntArrayBag`—Design…

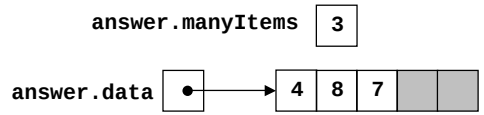
- **`clone` Method**
  - Create copy of bag.
  - Create copy of objects referred by bag's instance variables.



- **`super.clone();`**



- **`System.arrayCopy(…);`**



Class `IntArrayBag`—Design…

- **`clone` Method—Implementation**

```
public Object clone() {
  // Clone an IntArrayBag object.
  IntArrayBag answer;
  try {
    answer = (IntArrayBag) super.clone();
  }
  catch (CloneNotSupportedException e) {
    // Exception implies a programming error, most likely forgot "implements
    // Cloneable" in class header.
    throw new RuntimeException
      ("Class does not implement Cloneable");
  }
  answer.data = (int[]) data.clone();
  return answer;
}
```


Bag ADT—Time Analysis

- **Counting Operations**
  - Expressed as a function of bag capacity (c) or size (n).
- **Loop**

```
for (index = 0; index < manyItems; index++)
   if (target == data[index])
      answer++;
```

  - $n \times$ (number of statements in loop) $\Rightarrow O(n)$
- **Constructor**
  - Allocates array of `initialCapacity` integers.
  - Integers initialized to zero.

  $\Rightarrow O(c)$

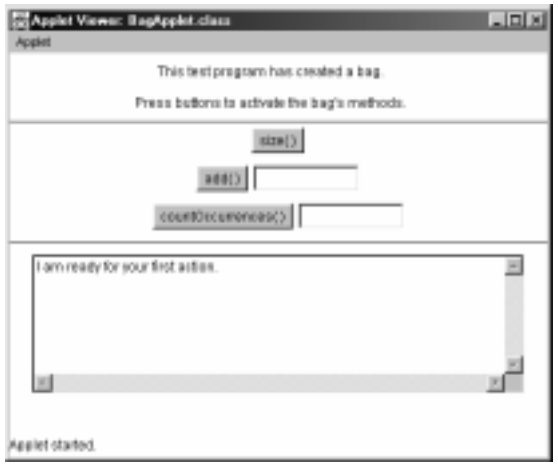
Bag ADT—Time Analysis…

Operation	Time Analysis	
Constructor	$O(c)$	c is the initial capacity
add without capacity increase	$O(1)$	Constant time
add with capacity increase	$O(n)$	Linear time
b1.addAll(b2) without capacity increase	$O(n_2)$	Linear in the size of the added bag
b1.addAll(b2) with capacity increase	$O(n_1+n_2)$	n_1 & n_2 are the sizes of the bags
clone	$O(c)$	c is the bag's capacity

Bag ADT—Time Analysis…

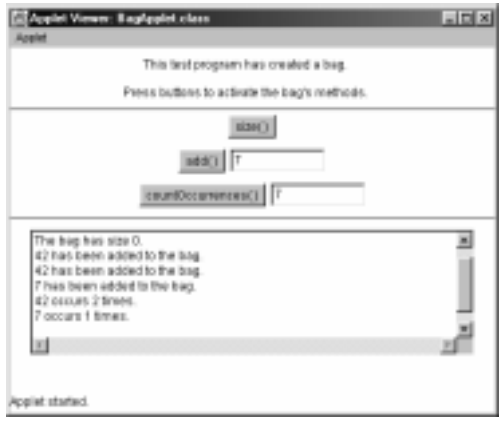
Operation	Time Analysis	
countOccurrences	$O(n)$	Linear time
ensureCapacity	$O(c)$	c is the specified minimum capacity
getCapacity	$O(1)$	Constant time
remove	$O(n)$	Linear time
size	$O(1)$	Constant time
trimToSize	$O(n)$	Linear time
union of b1 and b2	$O(c_1+c_2)$	c_1 & c_2 are the bags' capacities

Applets for Testing



Applets for Testing

- **Use *public interface* to exercise class methods.**
  - Add elements.
  - Obtain size.
  - Count occurrences.



Applets for Testing…

- **Template**
  - Import statements
  - Class definition
    - `IntArrayBag` object
    - GUI components of applet.
    - `init` method.
    - Implementations of action listeners
      - Events: buttons or input fields.
    - Implementations of other methods.

Applets for Testing…

- **Import Statements**
  - Class being tested.
    - Not needed if class is in same directory.

```
import edu.colorado.collections.IntArrayBag;
```

  - Java classes needed by applet.

```
import java.applet.Applet;  // Provides Applet class
import java.awt.*;          // Provides Button class…
import java.awt.event.*;    // ActionEvent, ActionListener.
```

Applets for Testing…

- **Class Definition**

```
public class BagApplet extends Applet {
  // IntArrayBag object for applet to manipulate
  IntArrayBag b = new IntArrayBag();
  // Declare applet's components: buttons, text fields, & other GUI comps
  ...
  public void init() {
    ...
  }
  // Implementation of action listeners.
  ...
  // Implementation of other methods.
  ...
}
```


Applets for Testing…

- **Applet's GUI Components**
  - Buttons, labels, text fields, and a text area.

```
// These are the interactive Components that will appear in the Applet.
// We declare one Button for each IntArrayBag method that we want
// to be able to test. If the method has an argument, then there is also a
// TextField where the user can enter the value of the argument. At the
// bottom, there is a TextArea to write messages.
Button    sizeButton        = new Button("size( )");
Button    addButton         = new Button("add( )");
TextField elementText       = new TextField(10);
Button    countOccurrencesButton =
                  new Button("countOccurrences( )");
TextField targetText        = new TextField(10);
TextArea  feedback          = new TextArea(7, 60);
```


Applets for Testing…

- **`init` Method**
  - Called by applet viewer after creating applet object.
    - Equivalent to an application's **`main`** method.
  - Adds GUI components.
    - Sets up layout of applet.
  - Activates GUI components.
    - Creates & attaches *listeners* to selected GUI components.

Applets for Testing…

- **Adding GUI Components**
  - `Use add method.`

    ```
    public void add(Component)
    ```

  - Adds a GUI component to applet object.
    - Buttons, labels,…
  - `add(sizeButton);`
  - `add(new Label("Test program created a bag"));`
  - Typical components:
    - `Button`—activates action when pressed.
    - `Label`—displays message on screen.
    - `TextField`—accepts or displays user input (one line).
    - `TextArea`—accepts or displays multiple lines of text.

Applets for Testing…

- **Activating GUI Components**
  - Create *listeners*.
  - Attach listeners to components.
- **Listeners**
  - Describe actions to be taken in reaction to *events*.
    - Button presses, CR in a text field, *etc.*
  - `SizeListener`—`sizeButton` is clicked.
  - `AddListener`—`addButton` is clicked or CR entered in `elementText` field.
  - `CountOccurrencesListener`—`countOccurrencesButton` is clicked or CR entered in `targetText` field.

Applets for Testing…

- **Implementing Action Listeners**
  - Define *inner* class.
    - Visible only within another class definition.
    - Has access to all instance variables.
  - Attach instance of inner class as listener.
- **Pattern**

```
class _____ implements ActionListener
{
  void actionPerformed(ActionEvent event)
  {
    ...
  }
}
```


Applets for Testing…

- **Action Listeners—"`size()`" Button**

```
class SizeListener implements ActionListener
{
  void actionPerformed(ActionEvent event)
  {
    feedback.append("The bag has size " +
                    b.size + "\n");
  }
}
```

- **Attaching Listeners**
  - `sizeButton(addActionListener(new SizeListener()));`

Applets for Testing…

- **Action Listeners—"`add()`" Button & Text Field**
  - Must handle *empty* or *mistyped* number.
    - `NumberFormatException` when interpreted as integer.
  - Use try-catch block.
- **Attaching Listener**
  - `AddListener addListener = new AddListener();`
  - `addButton.addActionListener(addListener);`
  - `elementText.addActionListener(addListener);`

Applets for Testing…

- **Action Listeners—"`add()`" Button & Text Field**

```
class AddListener implements ActionListener {
  void actionPerformed(ActionEvent event) {
    try {
      String userInput = elementText.getText();
      int element = Integer.parseInteger(userInput);
      b.add(element);
      feedback.append
        (element +" has been added to the bag");
    }
    catch (NumberFormatException e) {
      feedback.append
        ("Type an integer before clicking button");
      elementText.requestFocus();
      elementText.selectAll();
    }
  }}
```


Applets for Testing

- **See `BagApplet.java` listing**

Applets for Testing…

- **HTML File**
  - `<APPLET CODE="BagApplet.class" WIDTH=480 HEIGHT=340>`
    `</APPLET>`
- **Starting Applet**
  - Browser
  - `appletviewer`

Sequence ADT

- **Similar to a bag.**
  - Collection of items.
- **Defines *current* item.**
- **Has methods that step through elements.**
  - Useable in a loop.
- **Contains *doubles*.**

Class `DoubleArraySeq`—Specification

- **Constructors**
  - `public void DoubleArraySeq()`
    - Empty sequence with initial capacity of 10.
  - `public void DoubleArraySeq(int capacity)`
    - Empty sequence with *specified* initial capacity.
- **`size` Method**
  - `public int size()`
    - Returns number of elements in the sequence.

Class `DoubleArraySeq`—Specification...

- **Examining a Sequence**
  - `public void start()`
    - Initialize *current* element to point to sequence's first element.
  - `public double getCurrent()`
    - Return *current* element of sequence.
  - `public void advance()`
    - Update `current` to point to sequence's next element.
  - `public boolean isCurrent()`
    - Does *current* point to an actual element of sequence?
    - Is *current* pointing past sequence's edge?

Class `DoubleArraySeq`—Specification...

- **Examining a Sequence—Example**

```
for (numbers.start(); numbers.isCurrent();
     numbers.advance())
{
    System.out.println(numbers.getCurrent());
}
```

```
10.1
40.2
 1.1
```

  - **Output**
    - `10.1`
    - `40.2`
    - `1.1`

Class `DoubleArraySeq`—Specification...

- **`addBefore` & `addAfter` Methods**
  - `public void addBefore(double element)`
    - Place new element before current.
    - Make new element current.
    - Sequence with no *current* adds at front.
    - Empty sequence adds first element.
    - May need to increase capacity.
    - `seq.addBefore(10.0);`

Before

```
42.1
 8.8
99.0
```

After

```
42.1
10.0
 8.8
99.0
```

  - `public void addAfter(double element)`
    - Similar to `addBefore` method.
    - Place new element *after* current.

Class `DoubleArraySeq`—Specification...

- **`removeCurrent` Method**
  - **`public`** *`boolean`* `removeCurrent()`
    - Removes *current* element from sequence.
    - Current must point to actual element.
    - New current element is one after old current element.
    - `seq.removeCurrent(10.0);`

Before

42.1
10.0
8.8
99.0

After

42.1
8.8
99.0

Class `DoubleArraySeq`—Specification...

- **`addAll` Method**
  - **`public`** *`void`* `addAll(DoubleArraySeq sequence)`
    - Adds all elements of parameter sequence to this sequence.
    - This sequence's *current* does not change.
    - `s1.addAll(s2);`

Before

`s1` 3.7 8.3 4.1 3.1

`s2` 4.9 9.3 2.5

After

`s1`

3.7 8.3 4.1 3.1 4.9 9.3 2.5

Class `DoubleArraySeq`—Specification...

- **`concatenation` Method**
  - **`public static`** `concatenate (DoubleArraySeq s1, DoubleArraySeq s2)`
    - Creates new sequence with contents of first sequence followed by contents of second sequence.
    - New sequence has no *current* element.
    - `DoubleArraySeq total = DoubleArraySeq.concatenate(s1, s2);`

Before

`s1` 3.7 8.3 4.1 3.1

`s2` 4.9 9.3 2.5

After

`total`

3.7 8.3 4.1 3.1 4.9 9.3 2.5

Class `DoubleArraySeq`—Specification...

- **`clone` Method**
  - **`public`** *`Object`* `clone()`
    - Creates a new object that is a copy of this sequence.
- **Capacity Methods**
  - **`public`** *`int`* `getCapacity()`
    - Returns capacity of this sequence.
  - **`public`** *`void`* `ensureCapacity(`*`int`* `minimumCapacity)`
    - Makes sure sequence's capacity is at least as big as specified.
  - **`public`** *`void`* `trimToSize()`
    - Reduces sequence's capacity to sequence's number of items.

Class `DoubleArraySeq`—Design

- **Basic Structure**
  - Store numbers in array.
  - Keep track of number of items in array.
  - Keep track of current item.