University of Pittsburgh

Algorithms and Data Structures 1
CS 0445

Fall 2022

Sherif Khattab

ksm73@pitt.edu

(Slides are adapted from Dr. Ramirez's and Dr. Farnan's CS1501 slides.)

Announcements

- Recitations start this week
  - Lab 1 will be posted on Canvas
- Homework 1 due this Friday
- Draft slides and handouts available on Canvas
- Programming Assignment 1 will be posted this Friday
- TA Student Support hours now posted on the Syllabus page

Previous Lecture …

- ADT Bag
- Fixed-size array implementation

Muddiest Points (Java Generics)

- Reference types: Any type in Java except the primitive types (e.g., int, double, …). All reference types have Object as their ancestor
- Generic data type: public final class ArrayBag<T>
  - T is a formal type parameter
  - ArrayBag<Square> declares a reference variable of the generic type ArrayBag<T>
    - ArrayBag<Square> is a parameterized type
    - Square is the actual type parameter
- Generic data types, like any other class, may extend another class (generic or not) and/or implement interface(s) (generic or not)
- **Q: Is the wildcard type similar to a generic? Would you ever use it instead of a generic?**
- A: The wildcard type (denoted by a ?) means an unknown type. You can use instead of a type parameter if you don't want to specify any dependencies between method parameters and/or return type.
  - Also, note that ArrayBag<?>, not ArrayBag<Object>, is the superclass of all ArrayBag parameterized types.

- **Q: I found the muddiest points to be when to initialize a parameterized type array with an array of type object vs when to not initialize it with an array of type object. My understanding is, if the upper bound for the generic is NOT object, then we do not initialize it with type object but rather the upper bound, however I am not certain if this is correct.**
- A: ArrayBag<Integer>[] bags = new ArrayBag<?>[]; //no compilation errors
  - ArrayBag<Integer>[] bags = new Object[]; //compilation error
  - Right hand side has to be a super class (or the same class) as the left hand side
- **Q: the comparable interface.**

Muddiest Points

- **Q: I have no clue what PrintWriter and FileWriter do or how they work.**
- A: Both allow for writing into a text-output stream (e.g., a file). PrintWriter prints formatted representations of Objects whereas FileWriter prints streams of characters. You should always Check the JDK documentation (linked from Canvas) for more details.
- **Q: I was confused on how "@SuppressWarnings("unchecked")" is supposed to be used. What is "unchecked" used for here & how do you tell the difference between when to use "SupressWarnings" instead of fixing the issue?**
- A: Use it before element declaration to suppress the warnings(s) listed between parentheses. "unchecked" is the name for unchecked conversion warnings.
- **Q: using abstract class vs interface for bag adt**
- Q: I understand their relationship, but I do not understand when/why gadget and widget would be used.
- **Q: what defines the hierarchy of of array lists in regards to their bounds**
- A: ArrayList<?> is the supertype of all parametrized ArrayList types, such as ArrayList<Integer>, ArrayList<String>, etc.
- **Q: using the "this" keyword inside the default constructor**
- A: this(DEFAULT_CAPACITY) calls another constructor of the class

Today's Agenda

- ADT Bag Implementations
  - Fixed-size array: ArrayBag
  - Variable-size array: ResizableArrayBag

First Implementation of ADT Bag: Fixed-Size Array

An outline of the class **ArrayBag**

```
1  /**
2     A class of bags whose entries are stored in a fixed-size array.
3     @author Frank M. Carrano
4  */
5  public final class ArrayBag<T> implements BagInterface<T>
6  {
7     private final T[] bag;
8     private int numberOfEntries;
9     private static final int DEFAULT_CAPACITY = 25;
10
11    /** Creates an empty bag whose initial capacity is 25. */
12    public ArrayBag()
13    {
14        this(DEFAULT_CAPACITY);
15    } // end default constructor
16
17    /** Creates an empty bag having a given initial capacity.
18        @param capacity  The integer capacity desired. */
19    public ArrayBag(int capacity)
```

Fixed-Size Array

An outline of the class **`ArrayBag`**

```
17    /** Creates an empty bag having a given initial capacity.
18        @param capacity  The integer capacity desired. */
19    public ArrayBag(int capacity)
20    {
21        // The cast is safe because the new array contains null entries.
22        @SuppressWarnings("unchecked")
23        T[] tempBag = (T[])new Object[capacity]; // Unchecked cast
24        bag = tempBag;
25        numberOfEntries = 0;
26    } // end constructor
27
28    /** Adds a new entry to this bag.
29        @param newEntry  The object to be added as a new entry.
30        @return  True if the addition is successful, or false if not. */
31    public boolean add(T newEntry)
32    {
33        < Body to be defined >
34    } // end add
35
36    /** Retrieves all entries that are in this bag.
```

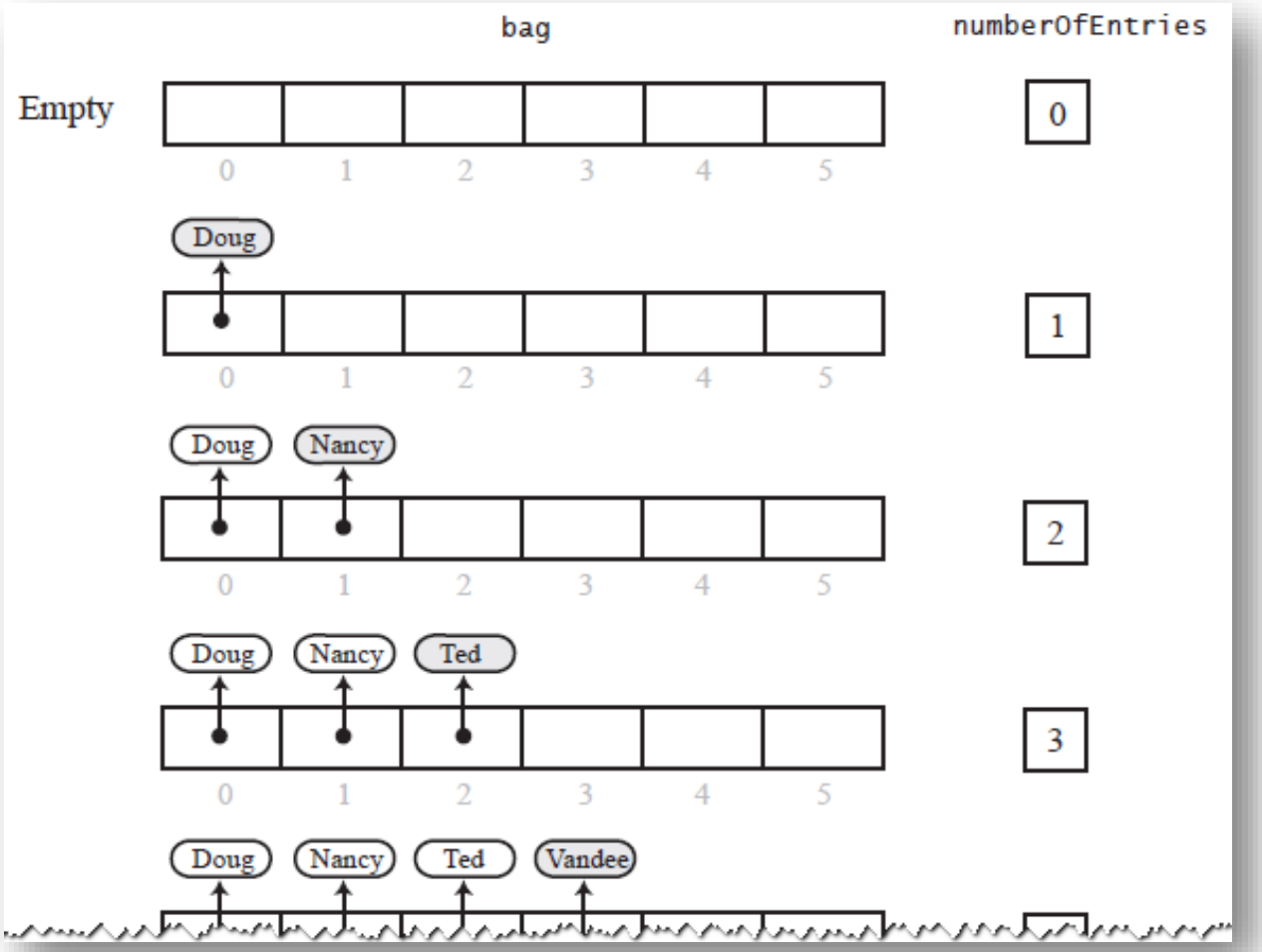
Fixed-Size Array

An outline of the class **`ArrayBag`**

```
36    /** Retrieves all entries that are in this bag.
37        @return  A newly allocated array of all the entries in the bag. */
38    public T[] toArray()
39    {
40        < Body to be defined >
41    } // end toArray
42
43    // Returns true if the arraybag is full, or false if not.
44    private boolean isArrayFull()
45    {
46        < Body to be defined >
47    } // end isArrayFull
48
49    < Similar partial definitions are here for the remaining methods
50      declared in BagInterface. >
51
52    . . .
53 } // end ArrayBag
```

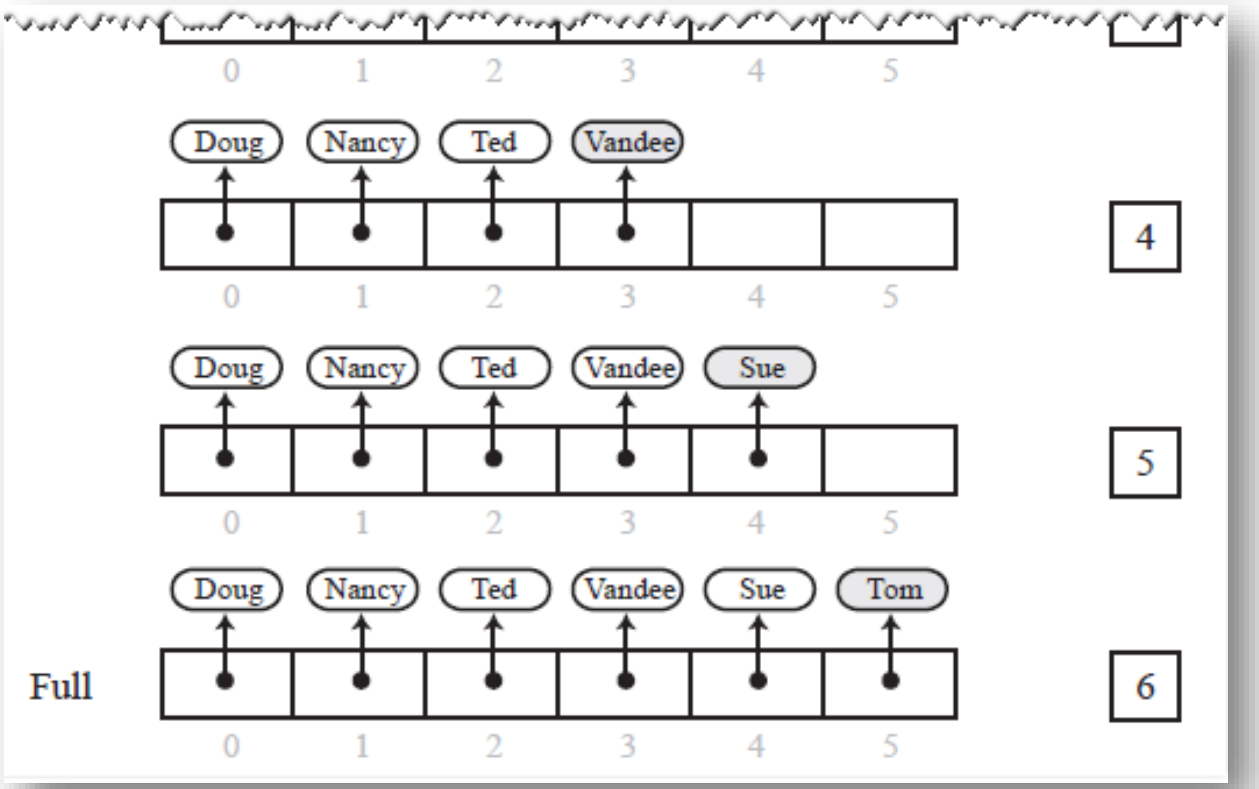
Fixed-Size Array

Adding entries to an array that represents a bag, whose capacity is six, until it becomes full



Fixed-Size Array

Adding entries to an array that represents a bag, whose capacity is six, until it becomes full



Fixed-Size Array

Method `add`

```
/** Adds a new entry to this bag.
    @param newEntry  The object to be added as a new entry.
    @return  True if the addition is successful, or false if not. */
public boolean add(T newEntry)
{
   boolean result = true;
   if (isArrayFull())
   {
      result = false;
   }
   else
   {  // Assertion: result is true here
      bag[numberOfEntries] = newEntry;
      numberOfEntries++;
   } // end if

   return result;
} // end add
```

Fixed-Size Array

Method **`isArrayFull`**

```
// Returns true if the bag is full, or false if not.
private boolean isArrayFull()
{
    return numberOfEntries >= bag.length;
} // end isArrayFull
```

Fixed-Size Array

Method `toArray`

```
/** Retrieves all entries that are in this bag.
    @return  A newly allocated array of all the entries in the bag. */
public T[] toArray()
{
   // The cast is safe because the new array contains null entries.
   @SuppressWarnings("unchecked")
   T[] result = (T[])new Object[numberOfEntries]; // Unchecked cast
   for (int index = 0; index < numberOfEntries; index++)
   {
       result[index] = bag[index];
   } // end for
   return result;
} // end toArray
```

Making the Implementation Secure

- Practice fail-safe programming by including checks for anticipated errors
- Validate input data and arguments to a method
- Refine incomplete implementation of **`ArrayBag`** to make code more secure by adding the following two data fields

```
private boolean initialized = false;
private static final int MAX_CAPACITY = 10000;
```

Making the Implementation Secure

Revised constructor

```
public ArrayBag(int desiredCapacity)
{
  if (desiredCapacity <= MAX_CAPACITY)
  {
     // The cast is safe because the new array contains null entries
     @SuppressWarnings("unchecked")
     T[] tempBag = (T[])new Object[desiredCapacity]; // Unchecked cast
     bag = tempBag;
     numberOfEntries = 0;
     initialized = true;                              // Last action
  }
  else
     throw new IllegalStateException("Attempt to create a bag " +
                                     "whose capacity exceeds " +
                                     "allowed maximum.");
} // end constructor
```

Making the Implementation Secure

Method to check initialization

```
// Throws an exception if this object is not initialized.
private void checkInitialization()
{
   if (!initialized)
      throw new SecurityException("ArrayBag object is not initialized " +
                                  "properly.");
} // end checkInitialization
```

Making the Implementation Secure

Revise the method **add**

```
public boolean add(T newEntry)
{
   checkInitialization();
   boolean result = true;
   if (isArrayFull())
   {
      result = false;
   }
   else
   { // Assertion: result is true here
     bag[numberOfEntries] = newEntry;
     numberOfEntries++;
   } // end if
   return result;
} // end add
```

Testing the Core Methods

Stubs for **`remove`** and **`clear`**

```
public T remove()
{
   return null; // STUB
} // end remove

...

public void clear()
{
   // STUB
} // end clear
```

Testing the Core Methods

A program that tests core methods of the class **ArrayBag**

```
1  /**
2     A test of the constructors and the methods add and toArray,
3     as defined in the first draft of the class ArrayBag.
4     @author Frank M. Carrano
5  */
6  public class ArrayBagDemo1
7  {
8     public static void main(String[] args)
9     {
10       // Adding to an initially empty bag with sufficient capacity
11       System.out.println("Testing an initially empty bag with" +
12                          " the capacity to hold at least 6 strings:");
13       BagInterface<String> aBag = new ArrayBag<> ();
14       String[] contentsOfBag1 = {"A", "A", "B", "A", "C", "A"};
15       testAdd(aBag, contentsOfBag1);
16
17       // Filling an initially empty bag to capacity
18       System.out.println("\nTesting an initially empty bag that " +
19                          " will be filled to capacity:");
20       aBag = new ArrayBag<>(7);
21       String[] contentsOfBag2 = {"A", "B", "A", "C", "B", "C", "D",
22                                  "another string"};
23       testAdd(aBag, contentsOfBag2);
24    } // end main
```

Testing the Core Methods

A program that tests core methods of the class **`ArrayBag`**

```
22                                    another String };
23          testAdd(aBag, contentsOfBag2);
24       } // end main
25
26       // Tests the method add.
27       private static void testAdd(BagInterface<String> aBag,
28                                   String[] content)
29       {
30          System.out.print("Adding the following " + content.length +
31                           " strings to the bag: ");
32          for (int index = 0; index < content.length; index++)
33          {
34             if (aBag.add(content[index]))
35                System.out.print(content[index] + " ");
36             else
```

Testing the Core Methods

A program that tests core methods of the class **ArrayBag**

```
37              System.out.print("\nUnable to add " + content[index] +
38                               " to the bag.");
39          } // end for
40          System.out.println();
41
42          displayBag(aBag);
43      } // end testAdd
44
45      // Tests the method toArray while displaying the bag.
46      private static void displayBag(BagInterface<String> aBag)
47      {
48          System.out.println("The bag contains the following string(s):");
49          Object[] bagArray = aBag.toArray();
50          for (int index = 0; index < bagArray.length; index++)
51          {
52              System.out.print(bagArray[index] + " ");
53          } // end for
54
55          System.out.println();
56      } // end displayBag
57  } // end ArrayBagDemo1
```

Testing the Core Methods

A program that tests core methods of the class **`ArrayBag`**

Output

```
Testing an initially empty bag with sufficient capacity:
Adding the following 6 strings to the bag: A A B A C A
The bag contains the following string(s):
A A B A C A

Testing an initially empty bag that will be filled to capacity:
Adding the following 8 strings to the bag: A B A C B C D
Unable to add another string to the bag.
The bag contains the following string(s):
A B A C B C D
```

Implementing More Methods

Methods `isEmpty` and `getCurrentSize`

```
public boolean isEmpty()
{
   return numberOfEntries == 0;
} // end isEmpty

public int getCurrentSize()
{
   return numberOfEntries;
} // end getCurrentSize
```

Implementing More Methods

Method `getFrequencyOf`

```
public int getFrequencyOf(T anEntry)
{
  checkInitialization();
  int counter = 0;

  for (int index = 0; index < numberOfEntries; index++)
  {
    if (anEntry.equals(bag[index]))
    {
       counter++;
    } // end if
  } // end for

  return counter;
} // end getFrequencyOf
```

Implementing More Methods

Method `contains`

```
public boolean contains(T anEntry)
{
   checkInitialization();
   boolean found = false;
   int index = 0;
   while (!found && (index < numberOfEntries))
   {
      if (anEntry.equals(bag[index]))
      {
         found = true;
      } // end if
      index++;
   } // end while

   return found;
} // end contains
```

Methods That Remove Entries

The method **clear**

```
/** Removes all entries from this bag. */
public void clear()
{
    while (!isEmpty())
        remove();
} // end clear
```

Methods That Remove Entries

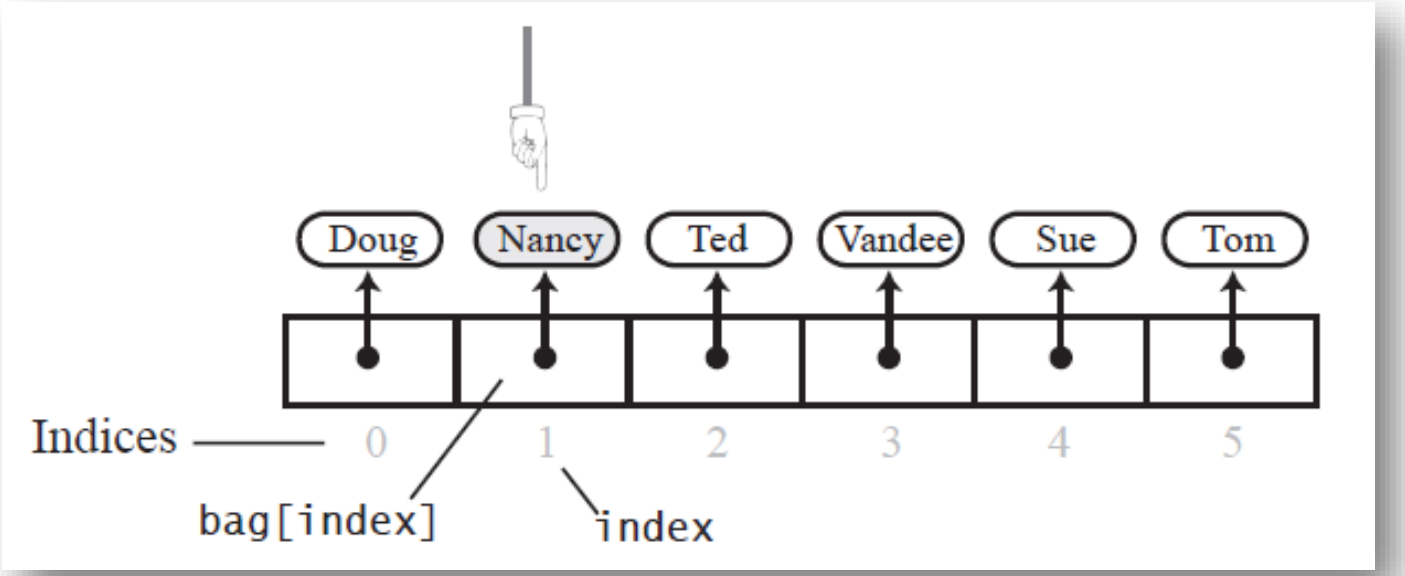
The method **`remove`**

```
public T remove()
{
    checkInitialization();
    T result = null;
    if (numberOfEntries > 0)
    {
        result = bag[numberOfEntries - 1];
        bag[numberOfEntries - 1] = null;
        numberOfEntries--;
    } // end if

    return result;
} // end remove
```

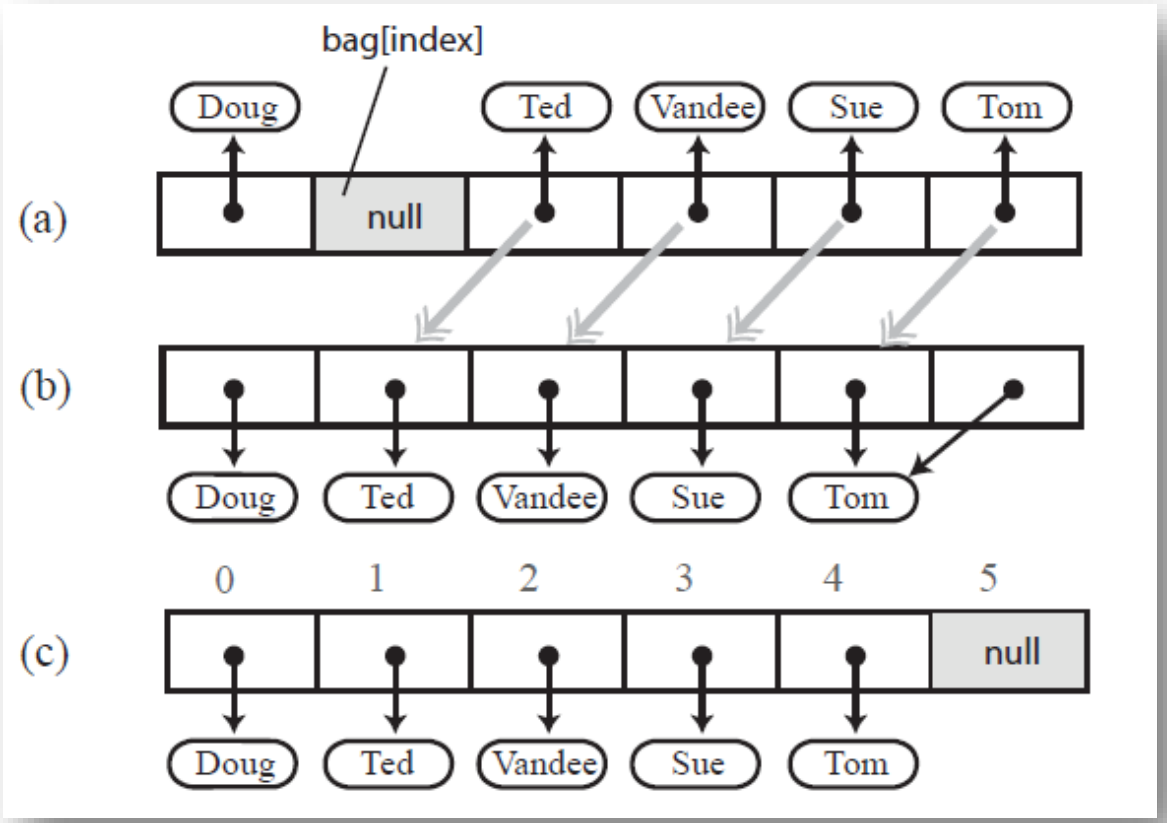
Methods That Remove Entries

The array bag after a successful search for the string “Nancy"



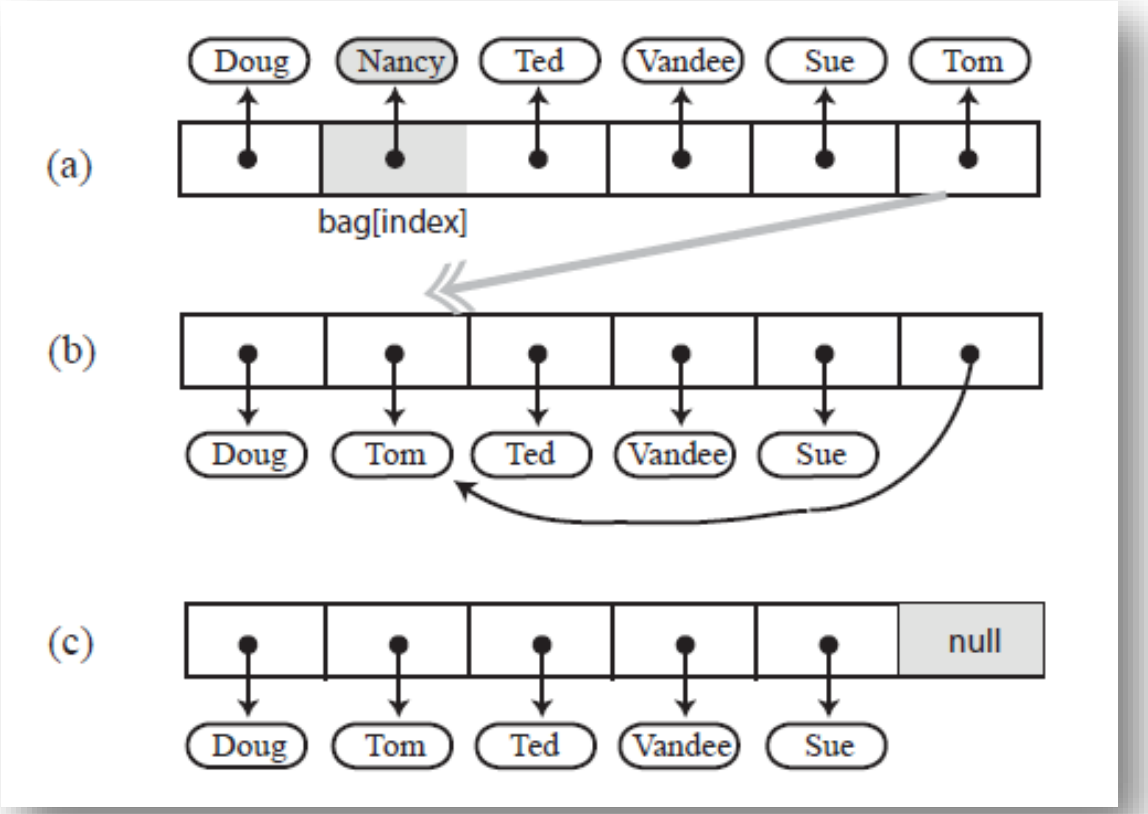
Methods That Remove Entries

- (a) A gap in the array bag after setting the entry in `bag[index]` to `null`
- (b-c) the array after shifting subsequent entries to avoid a gap



Methods That Remove Entries

Avoiding a gap in the array while removing an entry



Methods That Remove Entries

New definition of **`remove`**

```
public T remove()
{
    checkInitialization();
    T result = null;
    if (numberOfEntries > 0)
    {
        result = bag[numberOfEntries - 1];
        bag[numberOfEntries - 1] = null;
        numberOfEntries--;
    } // end if

    return result;
} // end remove
```

Methods That Remove Entries

The second **`remove`** method

```
/** Removes one occurrence of a given entry from this bag.
    @param anEntry  The entry to be removed.
    @return  True if the removal was successful, or false if not. */
public boolean remove(T anEntry)
{
   checkInitialization();
   int index = getIndexOf(anEntry);
   T result = removeEntry(index);
   return anEntry.equals(result);
} // end remove
```

Methods That Remove Entries

The **`removeEntry`** method

```
// Removes and returns the entry at a given index within the array bag.
// If no such entry exists, returns null.
// Preconditions: 0 <= givenIndex < numberOfEntries;
//                checkInitialization has been called.
private T removeEntry(int givenIndex)
{
    T result = null;
    if (!isEmpty() && (givenIndex >= 0))
    {
        result = bag[givenIndex];                    // Entry to remove
        bag[givenIndex] = bag[numberOfEntries - 1]; // Replace entry with last
                                                     // entry
        bag[numberOfEntries - 1] = null;             // Remove last entry
        numberOfEntries--;
    } // end if

    return result;
} // end removeEntry
```

Methods That Remove Entries

Definition for the method `getIndexOf`

```
// Locates a given entry within the array bag.
// Returns the index of the entry, if located, or -1 otherwise.
// Precondition: checkInitialization has been called.
private int getIndexOf(T anEntry)
{
   int where = -1;
   boolean found = false;
   int index = 0;
   while (!found && (index < numberOfEntries))
   {
      if (anEntry.equals(bag[index]))
      {
         found = true;
         where = index;
      } // end if
      index++;
   } // end while

   // Assertion: If where > -1, anEntry is in the array bag, and it
   // equals bag[where]; otherwise, anEntry is not in the array

   return where;
} // end getIndexOf
```

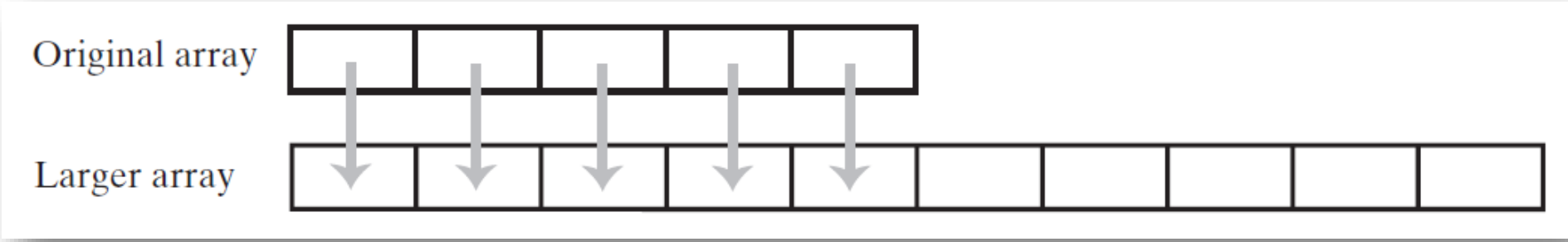
Methods That Remove Entries

Revised definition for the method **`contains`**

```
public boolean contains(T anEntry)
{
    checkInitialization();
    return getIndexOf(anEntry) > -1;
} // end contains
```

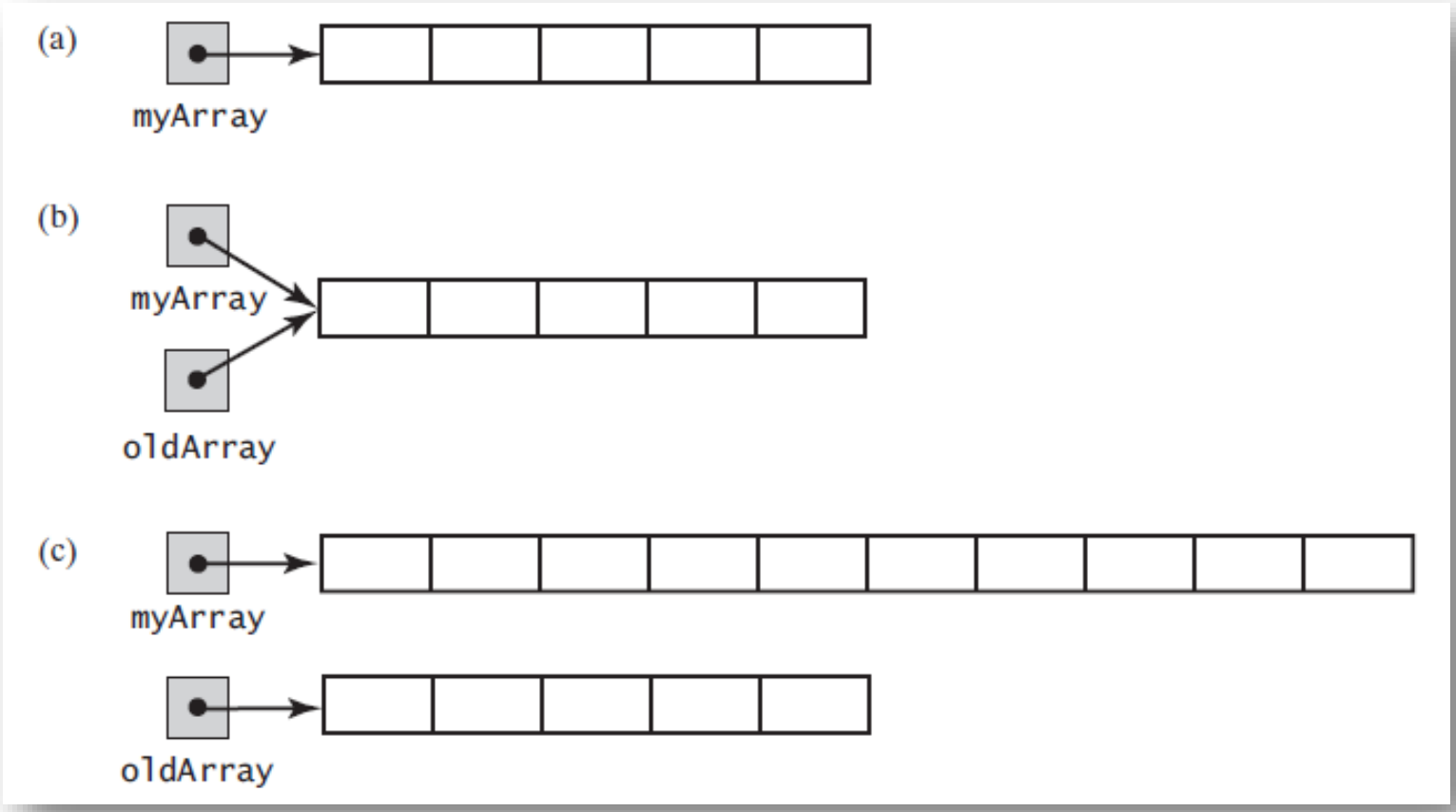
Using Array Resizing

Resizing an array copies its contents to a larger second array



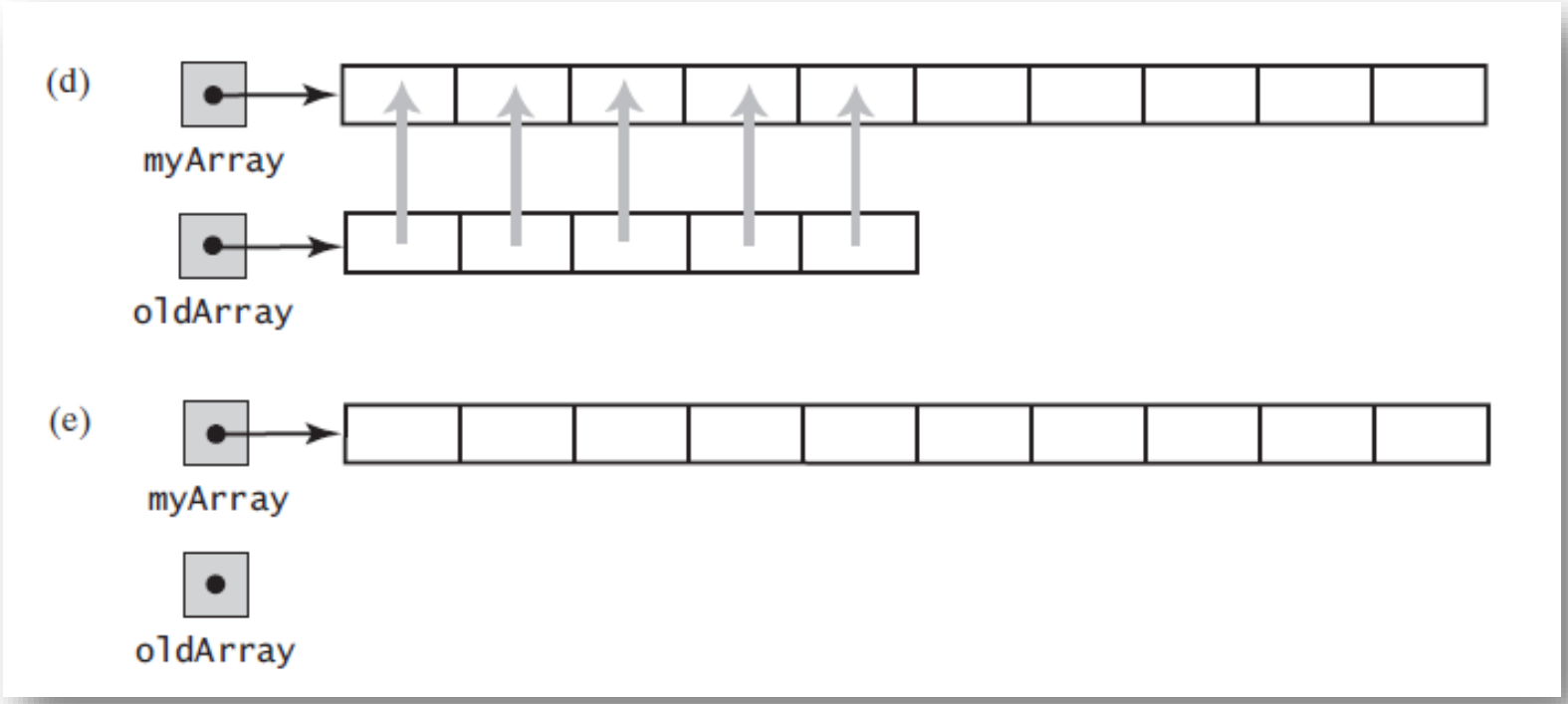
Using Array Resizing

(a) An array; (b) two references to the same array; (c) the original array variable now references a new, larger array;



Using Array Resizing

(d) the entries in the original array are copied to the new array; (e) the original array is discarded

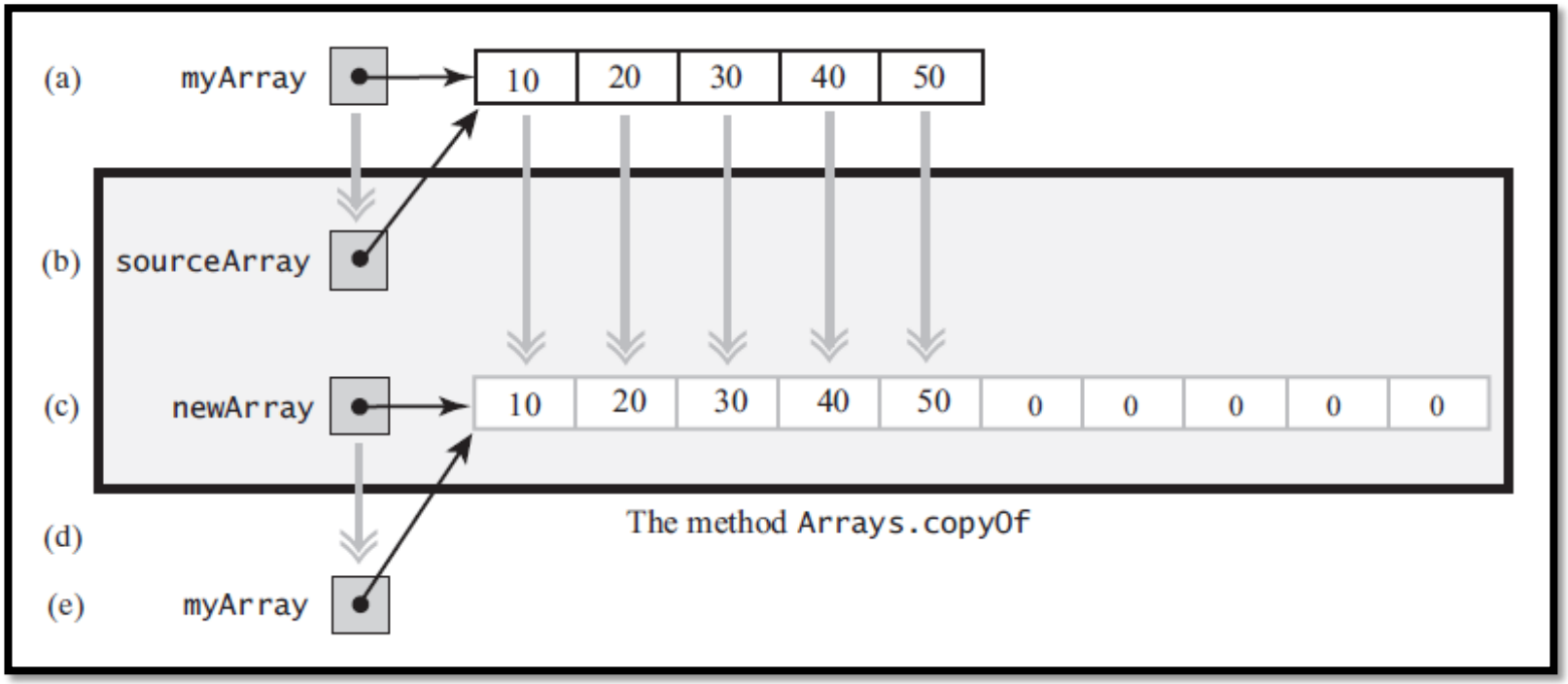


Using Array Resizing

The effect of the statement

```
myArray = Arrays.copyOf(myArray, 2 * myArray.length);
```

(a) The argument array; (b) the parameter that references the argument array; (c) a new, larger array that gets the contents of the argument array; (d) the return value that references the new array; (e) the argument variable is assigned the return value



New Implementation of a Bag

Previous definition of method **add**

```
public boolean add(T newEntry)
{
    checkInitialization();
    boolean result = true;
    if (isArrayFull())
    {
        result = false;
    }
    else
    { // Assertion: result is true here
      bag[numberOfEntries] = newEntry;
      numberOfEntries++;
    } // end if
    return result;
} // end add
```

New Implementation of a Bag

The method **doubleCapacity**

```
// Doubles the size of the array bag.
// Precondition: checkInitialization has been called.
private void doubleCapacity()
{
   int newLength = 2 * bag.length;
   checkCapacity(newLength);
   bag = Arrays.copyOf(bag, newLength);
} // end doubleCapacity
```

Using a Bag

A Bag is a simple ADT, but it can still be useful

- See examples in text
- Here is another simple one
  - A number of players “shout” Snap! each with a certain probability.
  - Add the player number to a Bag if she shouts.
  - Count the number of shouts in the Bag.

Pros and Cons of Using an Array

- Adding an entry to the bag is fast
- Removing an unspecified entry is fast
- Removing a particular entry requires time to locate the entry
- Increasing the size of the array requires time to copy its entries

Problems with Array Implementation

- Array has fixed size
- May become full
- Alternatively may have wasted space
- Resizing is possible but requires time overhead