# Bag Implementations that Use Arrays

## Chapter 2

*Data Structures and Abstractions with Java, 4e, Global Edition*
Frank Carrano

# Fixed-Size Array to Implement the ADT Bag

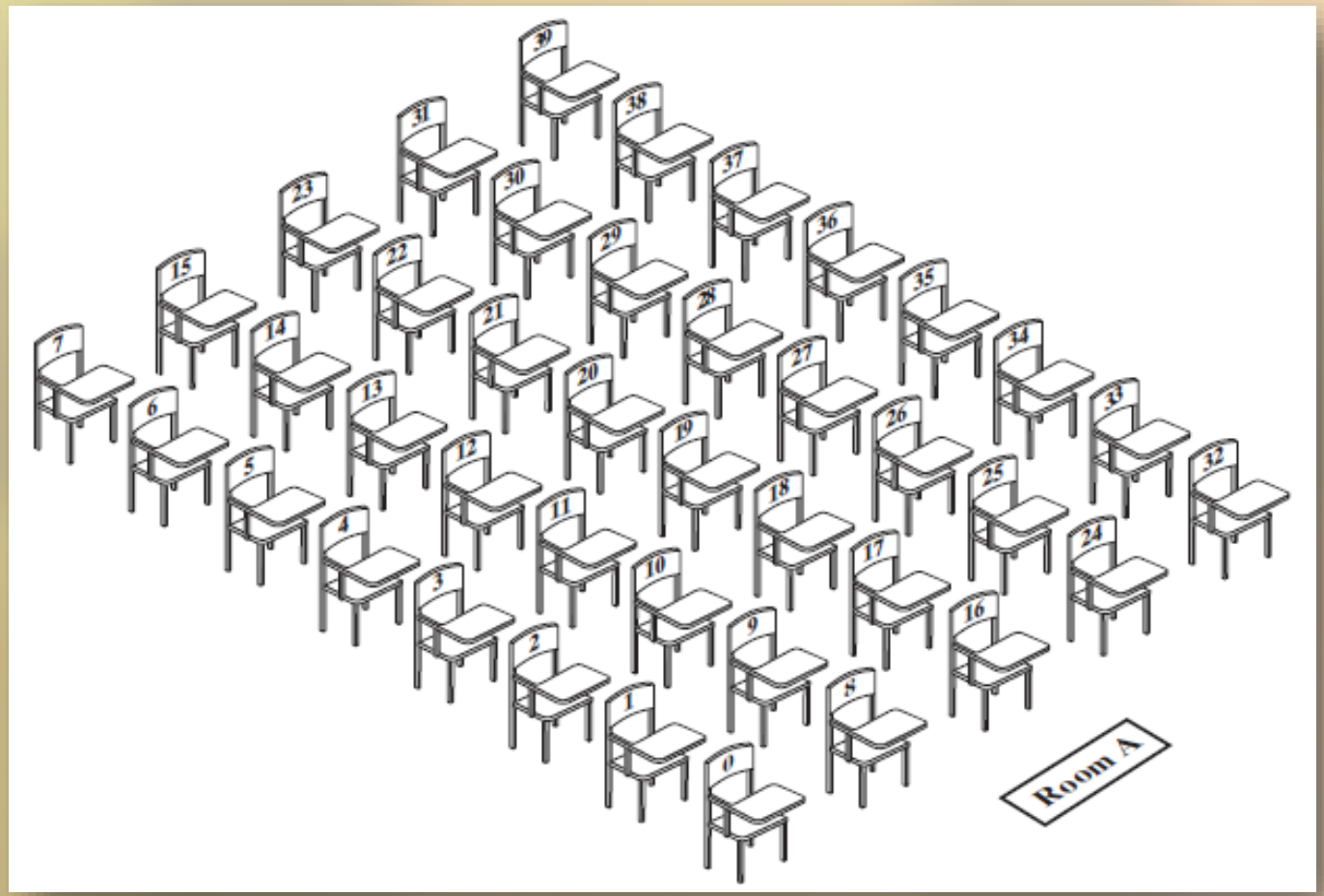


FIGURE 2-1 A classroom that contains desks in fixed positions

# Fixed-Size Array

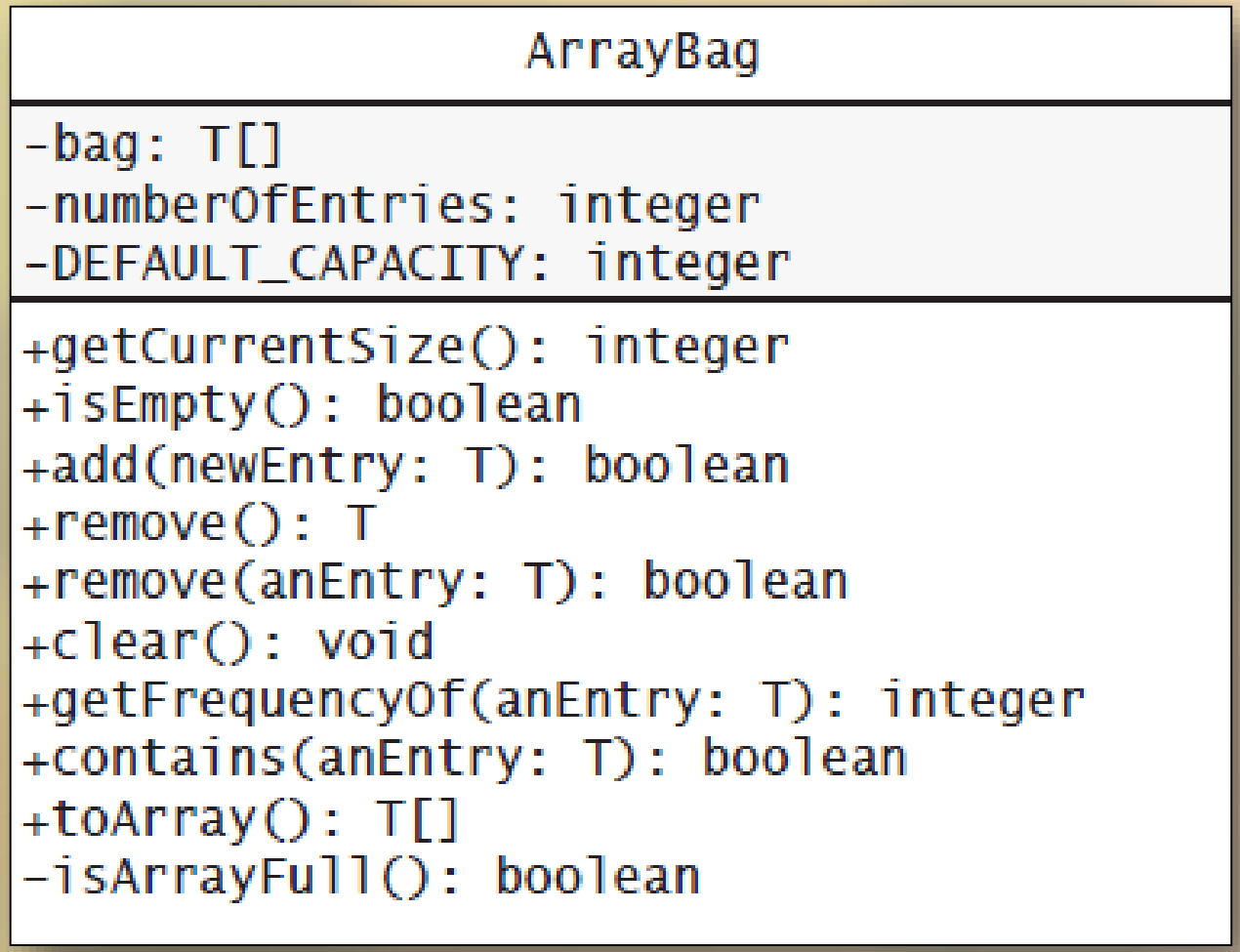


FIGURE 2-2 UML notation for the class **`ArrayBag`**, including the class's data fields

# Fixed-Size Array

```
/**
   A class of bags whose entries are stored in a fixed-size array.
   @author Frank M. Carrano
*/
public final class ArrayBag<T> implements BagInterface<T>
{
   private final T[] bag;
   private int numberOfEntries;
   private static final int DEFAULT_CAPACITY = 25;

   /** Creates an empty bag whose initial capacity is 25. */
   public ArrayBag()
   {
       this(DEFAULT_CAPACITY);
   } // end default constructor

   /** Creates an empty bag having a given initial capacity.
       @param capacity  The integer capacity desired. */
   public ArrayBag(int capacity)
```

# Fixed-Size Array

```
/** Creates an empty bag having a given initial capacity.
    @param capacity  The integer capacity desired. */
public ArrayBag(int capacity)
{
    // The cast is safe because the new array contains null entries.
    @SuppressWarnings("unchecked")
    T[] tempBag = (T[])new Object[capacity]; // Unchecked cast
    bag = tempBag;
    numberOfEntries = 0;
} // end constructor

/** 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)
{
    < Body to be defined >
} // end add

/** Retrieves all entries that are in this bag.
```

# Fixed-Size Array

```
    /** Retrieves all entries that are in this bag.
        @return  A newly allocated array of all the entries in the bag. */
    public T[] toArray()
    {
        < Body to be defined >
    } // end toArray

    // Returns true if the arraybag is full, or false if not.
    private boolean isArrayFull()
    {
        < Body to be defined >
    } // end isArrayFull

    < Similar partial definitions are here for the remaining methods
      declared in BagInterface. >

    . . .
} // end ArrayBag
```

# Fixed-Size Array

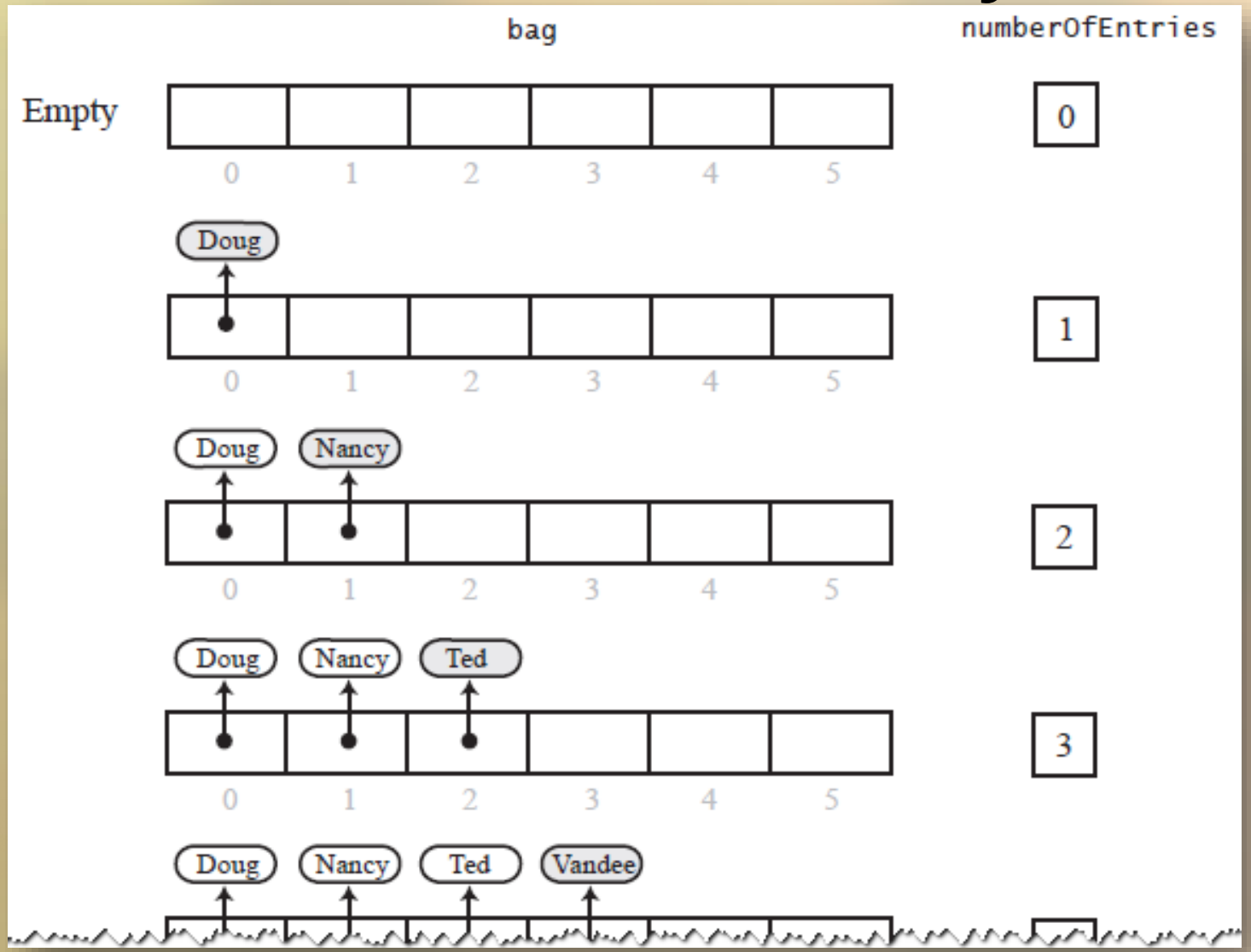


FIGURE 2-3 Adding entries to an array that represents a bag, whose capacity is six, until it becomes full

# Fixed-Size Array

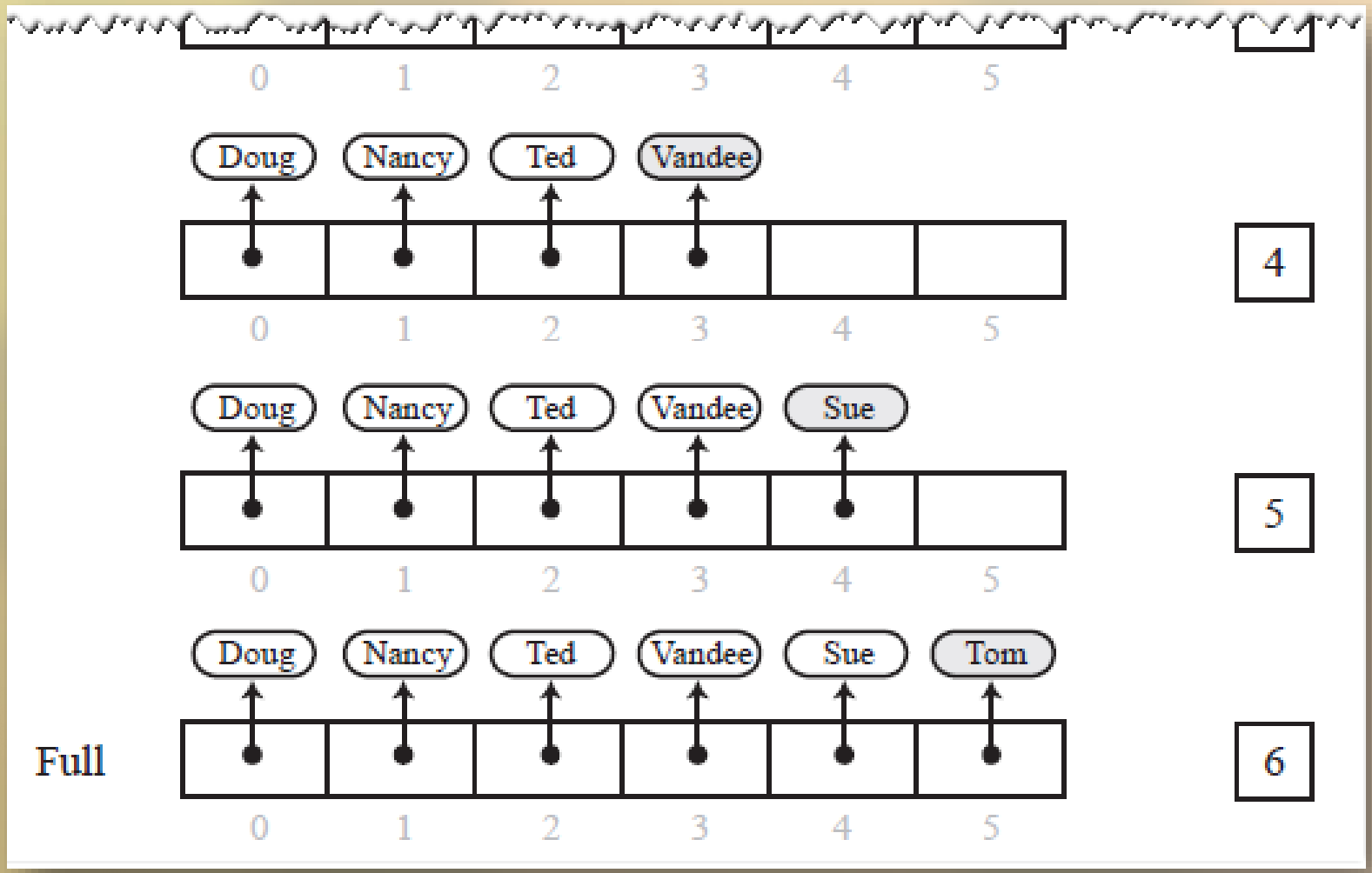


FIGURE 2-3 Adding entries to an array that represents a bag, whose capacity is six, until it becomes full

# Fixed-Size Array

```
/** 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

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

Method **`isFull`**

# Fixed-Size Array

```
/** 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
```

Method **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

```
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
```

Revised constructor

# Making the Implementation Secure

```
// 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
```

Method to check initialization

# Making the Implementation Secure

```
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

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

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

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

# Testing the Core Methods

```
/**
   A test of the constructors and the methods add and toArray,
   as defined in the first draft of the class ArrayBag.
   @author Frank M. Carrano
*/
public class ArrayBagDemo1
{
   public static void main(String[] args)
   {
      // Adding to an initially empty bag with sufficient capacity
      System.out.println("Testing an initially empty bag with" +
                         " the capacity to hold at least 6 strings:");
      BagInterface<String> aBag = new ArrayBag<> ();
      String[] contentsOfBag1 = {"A", "A", "B", "A", "C", "A"};
      testAdd(aBag, contentsOfBag1);

      // Filling an initially empty bag to capacity
      System.out.println("\nTesting an initially empty bag that " +
                         " will be filled to capacity:");
      aBag = new ArrayBag<>(7);
      String[] contentsOfBag2 = {"A", "B", "A", "C", "B", "C", "D",
                                 "another string"};
      testAdd(aBag, contentsOfBag2);
   } // end main
```

# Testing the Core Methods

```
                                                another String };
      testAdd(aBag, contentsOfBag2);
   } // end main

   // Tests the method add.
   private static void testAdd(BagInterface<String> aBag,
                               String[] content)
   {
      System.out.print("Adding the following " + content.length +
                       " strings to the bag: ");
      for (int index = 0; index < content.length; index++)
      {
         if (aBag.add(content[index]))
            System.out.print(content[index] + " ");
         else
```

LISTING 2-2 A program that tests core methods of the class **ArrayBag**

# Testing the Core Methods

```
            System.out.print("\nUnable to add " + content[index] +
                             " to the bag.");
         } // end for
         System.out.println();

         displayBag(aBag);
      } // end testAdd

      // Tests the method toArray while displaying the bag.
      private static void displayBag(BagInterface<String> aBag)
      {
         System.out.println("The bag contains the following string(s):");
         Object[] bagArray = aBag.toArray();
         for (int index = 0; index < bagArray.length; index++)
         {
            System.out.print(bagArray[index] + " ");
         } // end for

         System.out.println();
      } // end displayBag
} // end ArrayBagDemo1
```

# Testing the Core Methods

**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
```

LISTING 2-2 A program that tests core methods of the class **`ArrayBag`**

# Implementing More Methods

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

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

Methods **`isEmpty`** and **`getCurrentSize`**

# Implementing More Methods

```
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
```

Method **`getFrequencyOf`**

# Implementing More Methods

```
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
```

Method **contains**

# Methods That Remove Entries

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

The method **clear**

# Methods That Remove Entries

```
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
```

The method **remove**

# Methods That Remove Entries

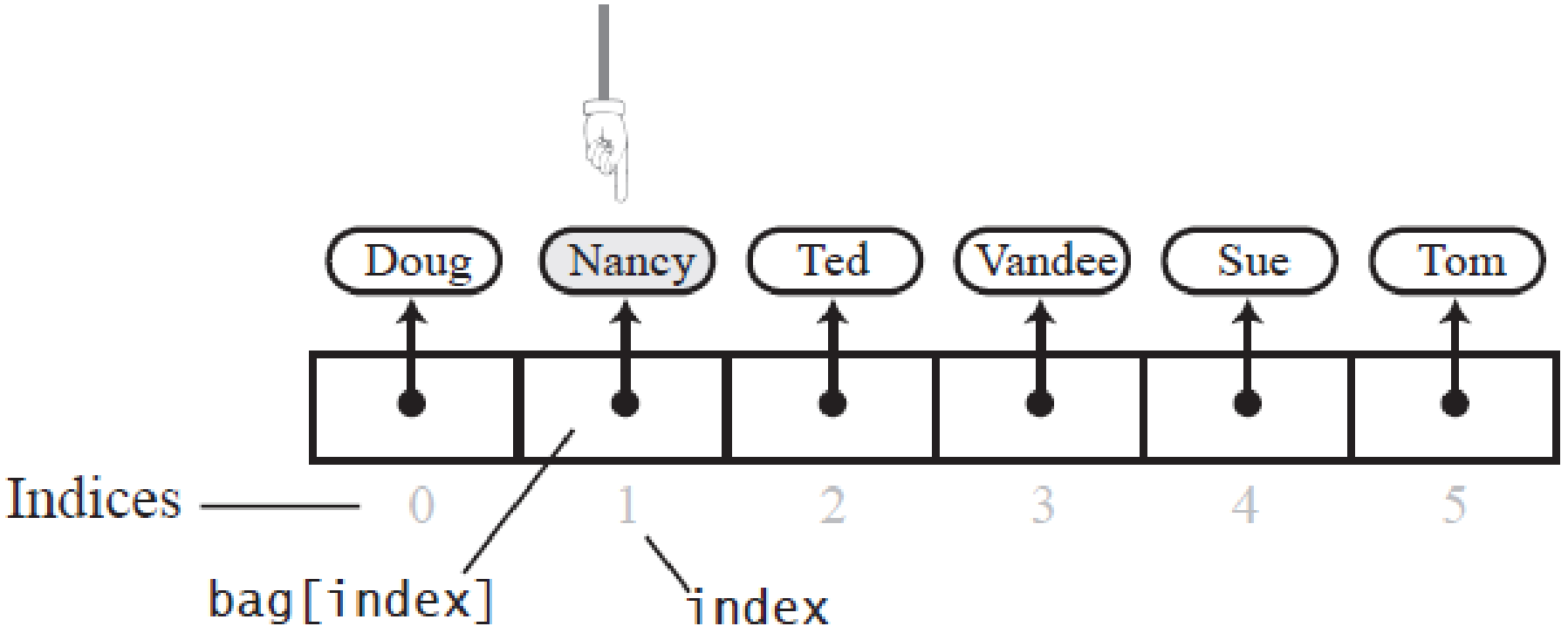


FIGURE 2-4 The array bag after a successful search for the string “Nancy"

# Methods That Remove Entries

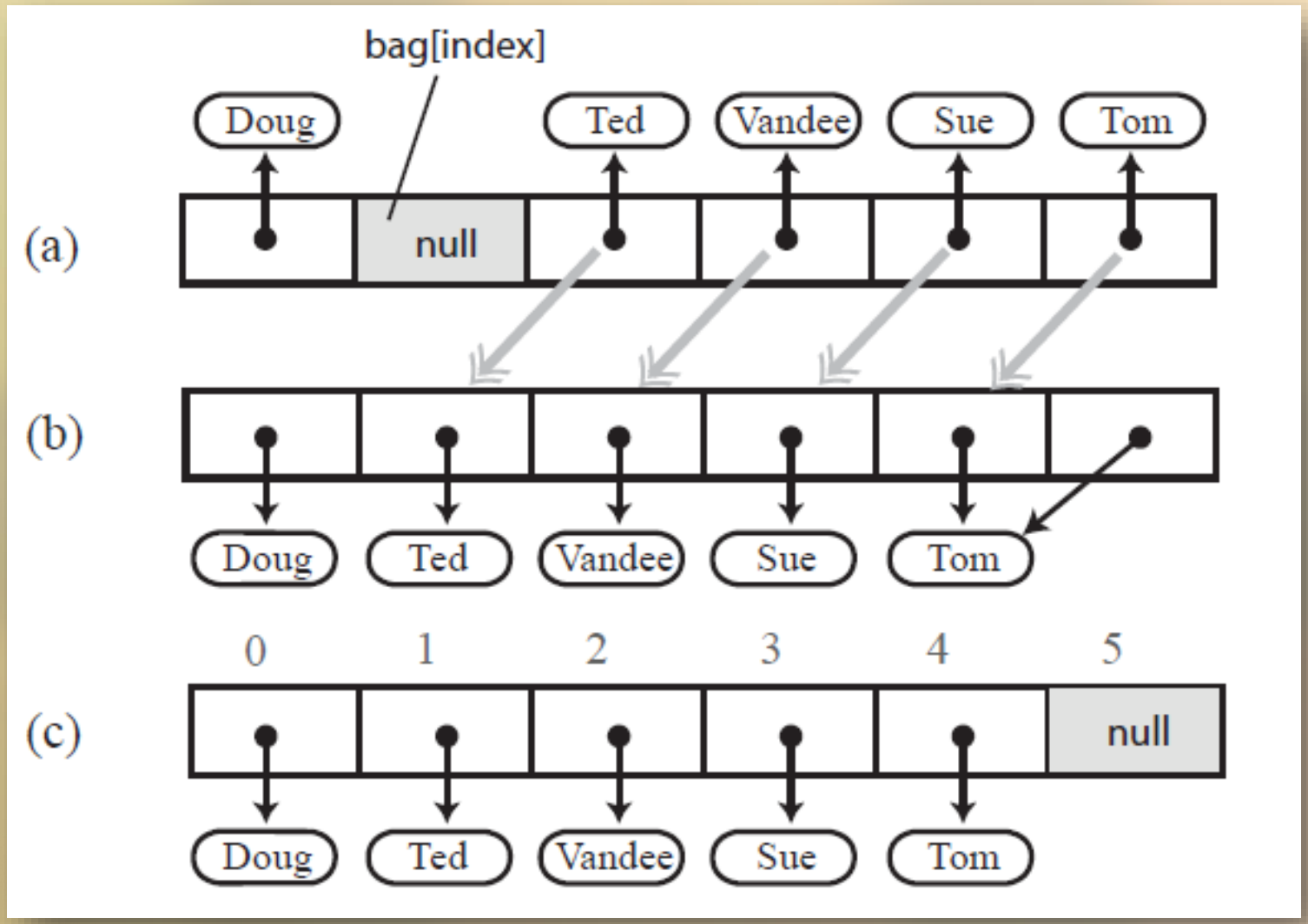


FIGURE 2-5 (a) A gap in the array bag after setting the entry in `bag[index]` to `null`; (b) the array after shifting subsequent entries to avoid a gap

# Methods That Remove Entries

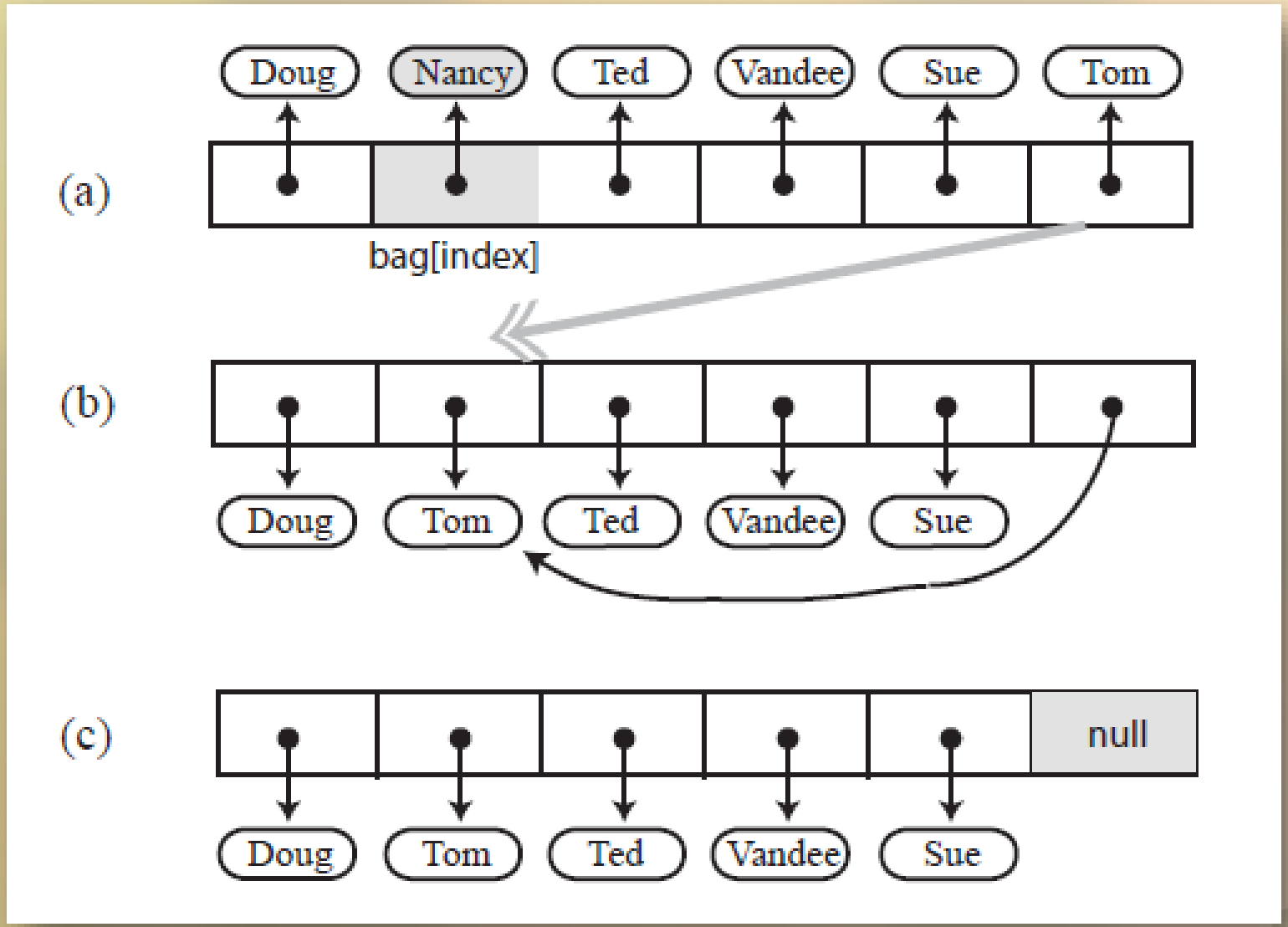


FIGURE 2-6 Avoiding a gap in the array while removing an entry

# Methods That Remove Entries

```
/** 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
```

The second **`remove`** method

# Methods That Remove Entries

```
// 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
```

The **`removeEntry`** method

# Methods That Remove Entries

```
// 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

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

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

# Using Array Resizing

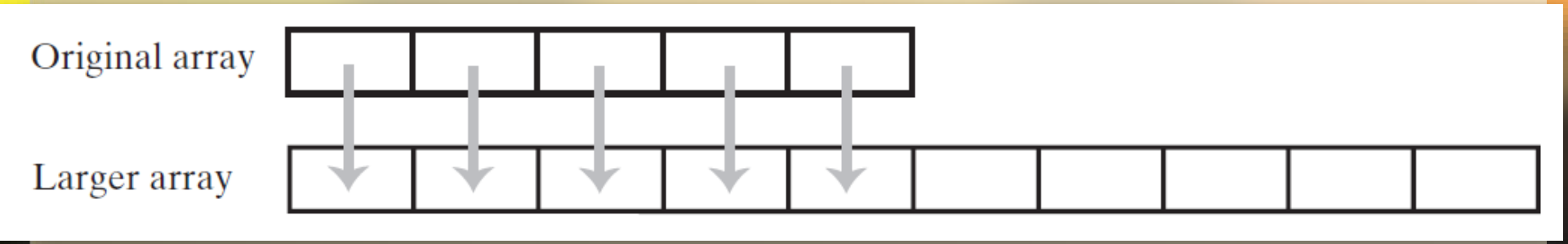


FIGURE 2-7 Resizing an array copies its contents to a larger second array

# Using Array Resizing

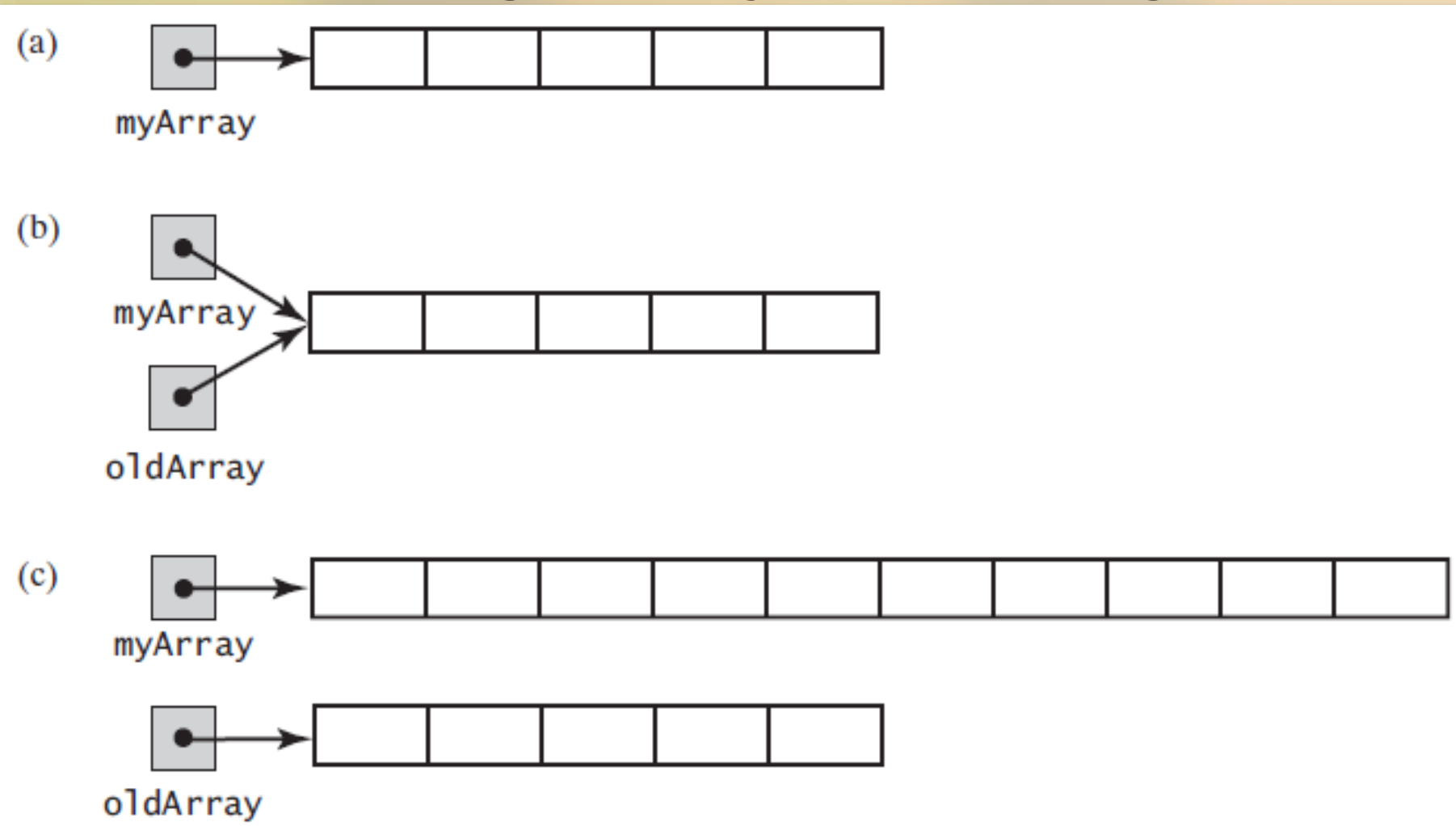


FIGURE 2-8 (a) An array; (b) two references to the same array; (c) the original array variable now references a new, larger array;

# Using Array Resizing

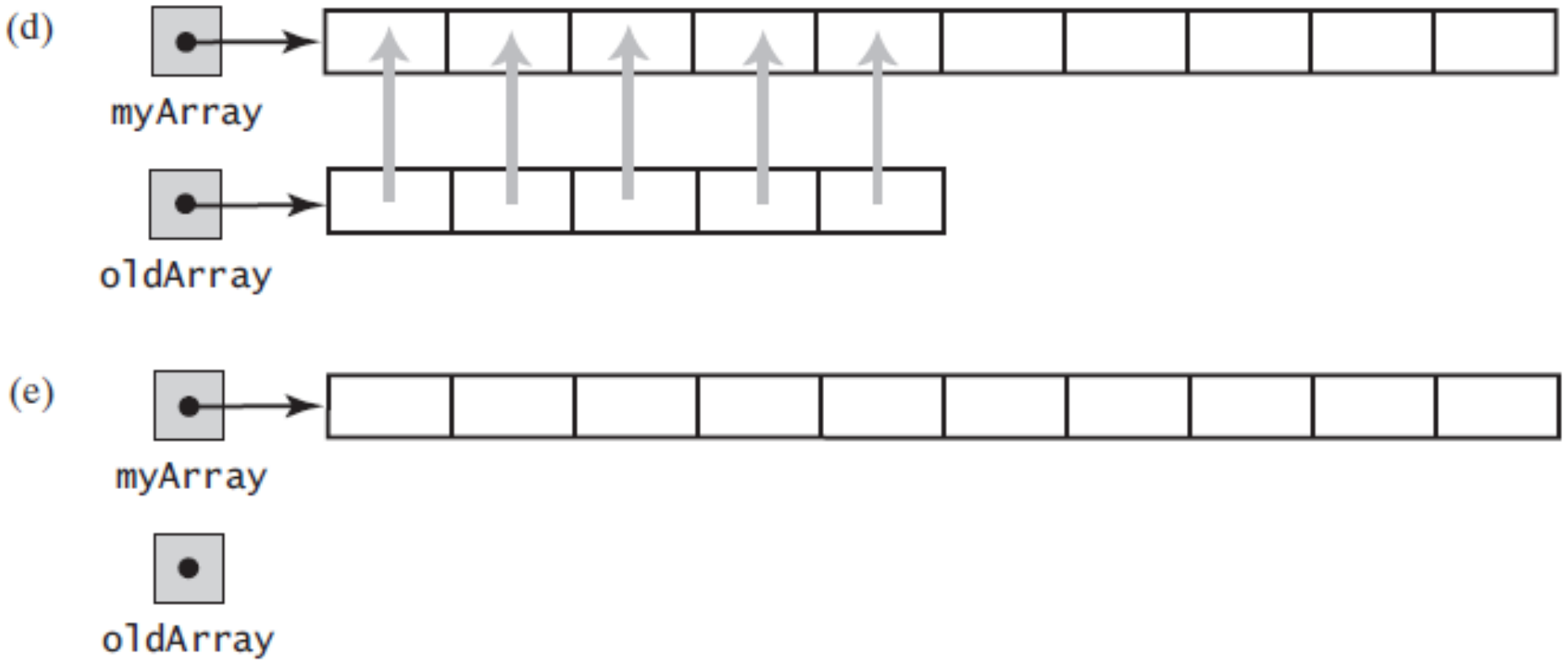


FIGURE 2-8 (d) the entries in the original array are copied to the new array; (e) the original array is discarded

# Using Array Resizing

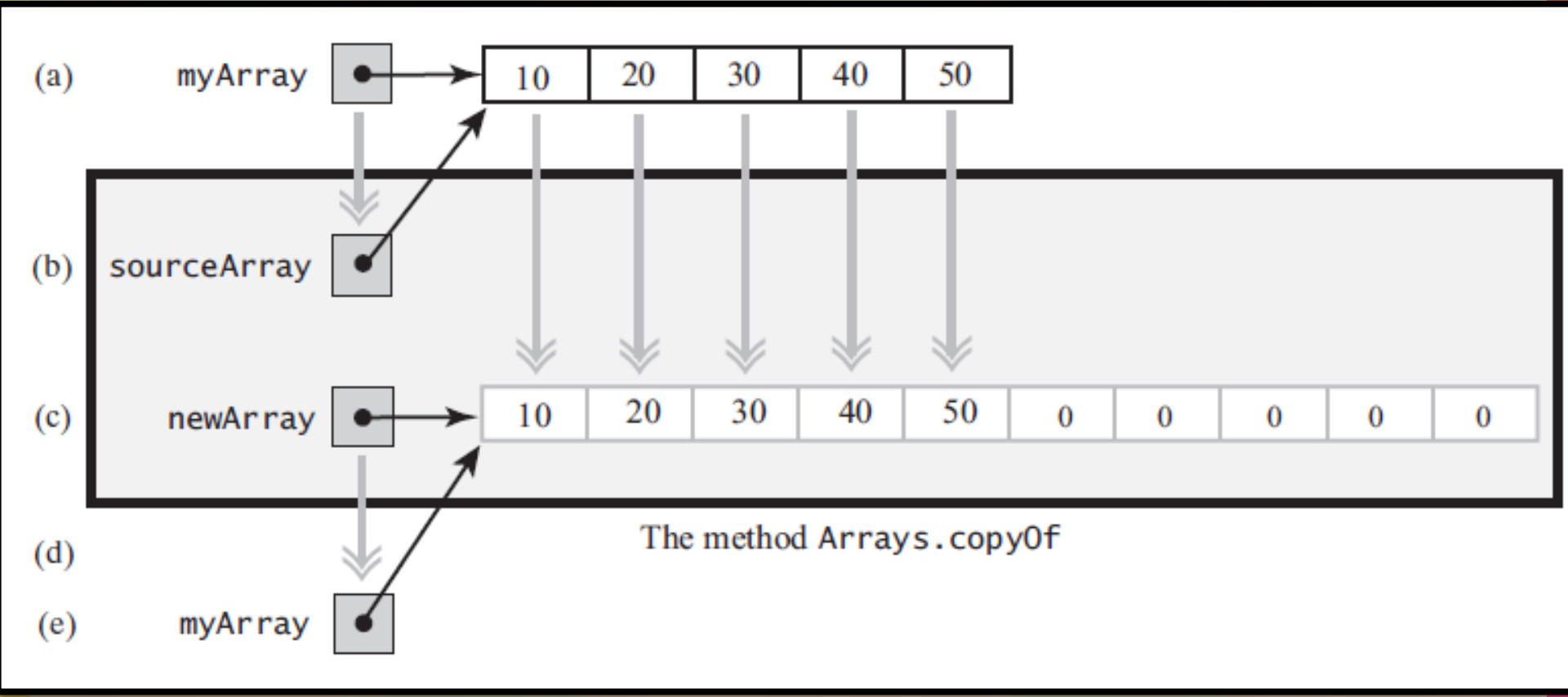


FIGURE 2-9 The effect of the statement

```
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

```
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
```

Previous definition of method **add**

# New Implementation of a Bag

```
public boolean add(T newEntry)
{
    checkInitialization();
  if (isArrayFull())
  {
     doubleCapacity();
  } // end if

  bag[numberOfEntries] = newEntry;
  numberOfEntries++;

  return true;
} // end add
```

New definition of method **add**

# New Implementation of a Bag

```
// 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
```

Revision of method **doubleCapacity**

# 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

# End

Chapter 2