Bag Implementations that Use Arrays

Chapter 2

Contents

- Using a Fixed-Size Array to Implement the ADT Bag
  - An Analogy
  - A Group of Core Methods
  - Implementing the Core Methods
  - Testing the Core Methods
  - Implementing More Methods
  - Methods That Remove Entries

Contents

- Using Array Resizing to Implement the ADT Bag
  - Resizing an Array
  - A New Implementation of a Bag
- The Pros and Cons of Using an Array to Implement the ADT Bag

Objectives

- Implement ADT bag using
  - a fixed-size array or
  - an array that expanded dynamically
- Discuss advantages and disadvantages of implementations presented

Implement with Fixed Size Array

- Define methods specified by previous interface **`BagInterface`**
- Consider use of fixed size array
  - Not unlike a classroom with exactly 40 desks
  - Numbered from 0 to 39

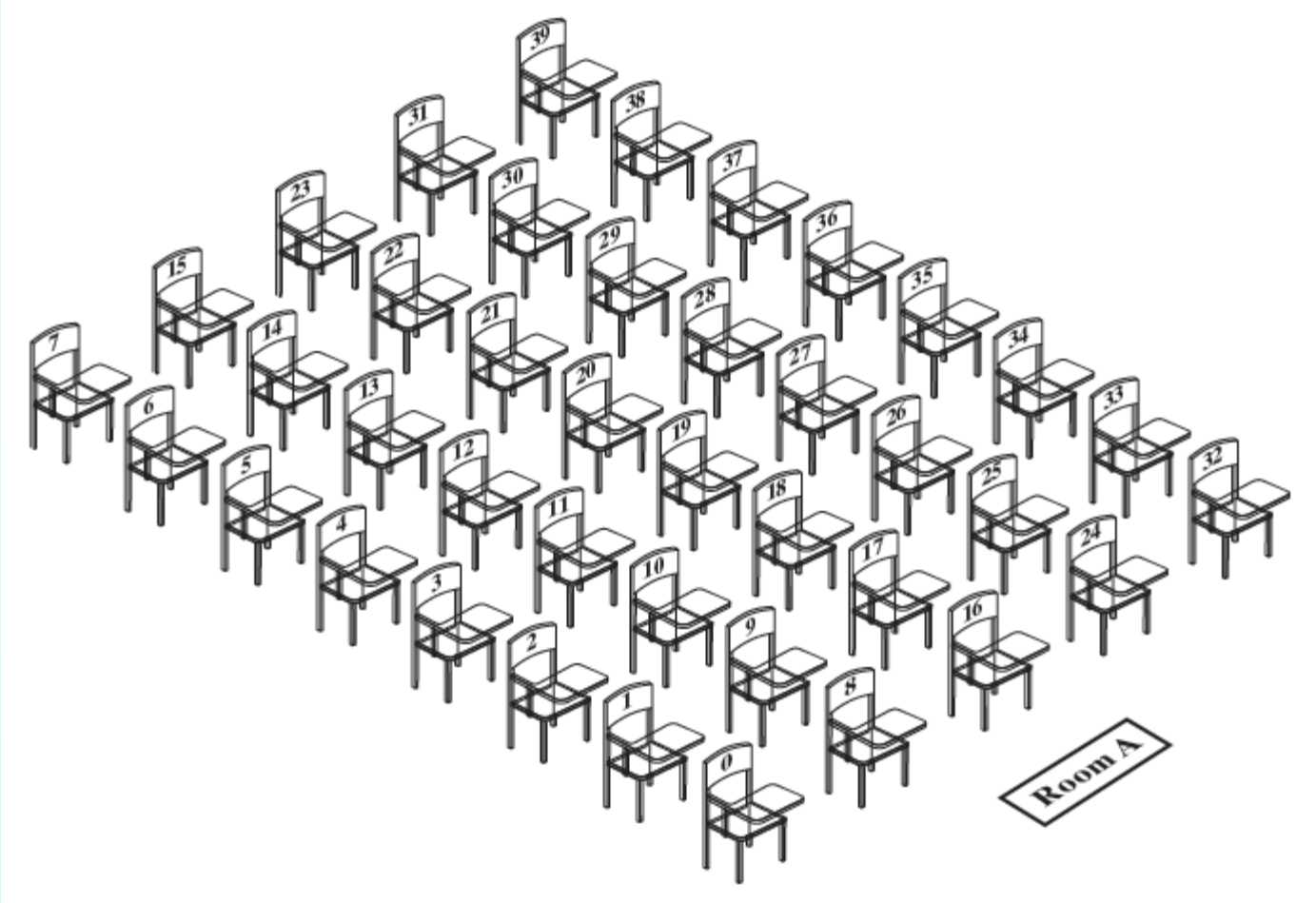
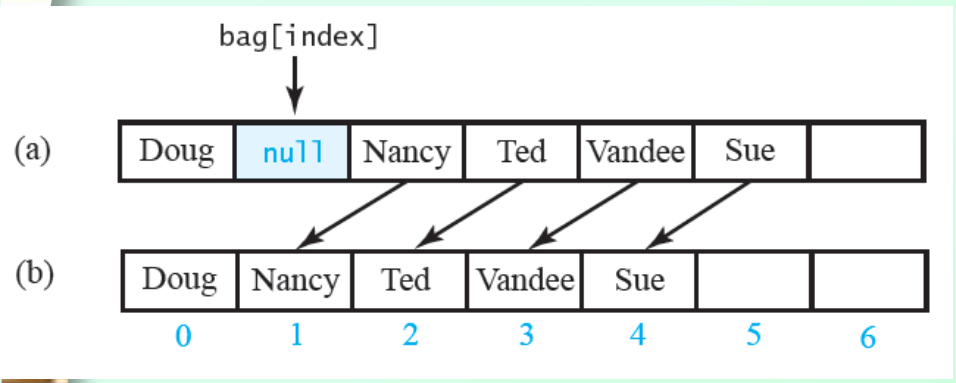


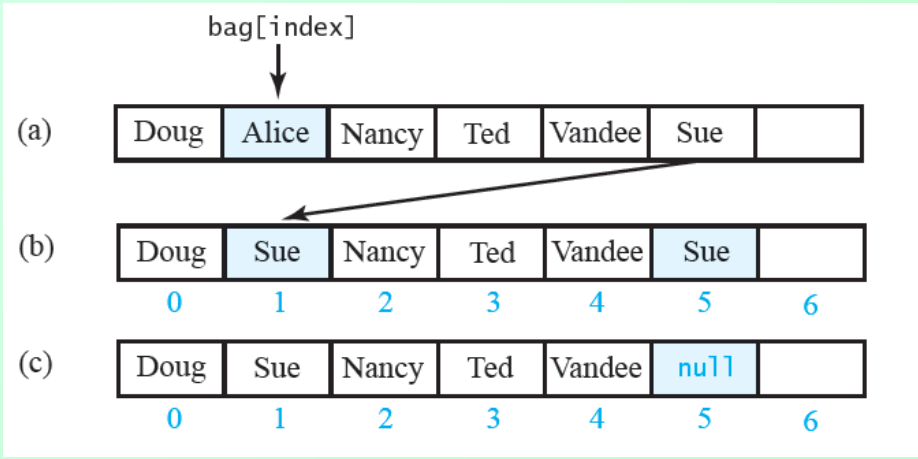
Figure 2-1 A classroom that contains desks in fixed positions

Adding, Removing Students

- Adding
  - Arbitrarily specify consecutively numbered desks be occupied
  - When desk #39 occupied, room is full
- Removing
  - What to do when someone in middle of sequence is removed?
  - Move last person there or shift everyone?

Options for removing a Student





Question 1 What is an advantage of moving a student as just described so that the vacated desk does not remain vacant?

Question 2 What is an advantage of leaving the vacated desk vacant?

Question 3 If a student were to drop the course, which one could do so without forcing another to change desks?

1. The students remain in consecutively numbered desks. You do not have to keep track of the locations of the empty desks.
2. Time is saved by not moving a student.
3. The student in the highest-numbered desk.

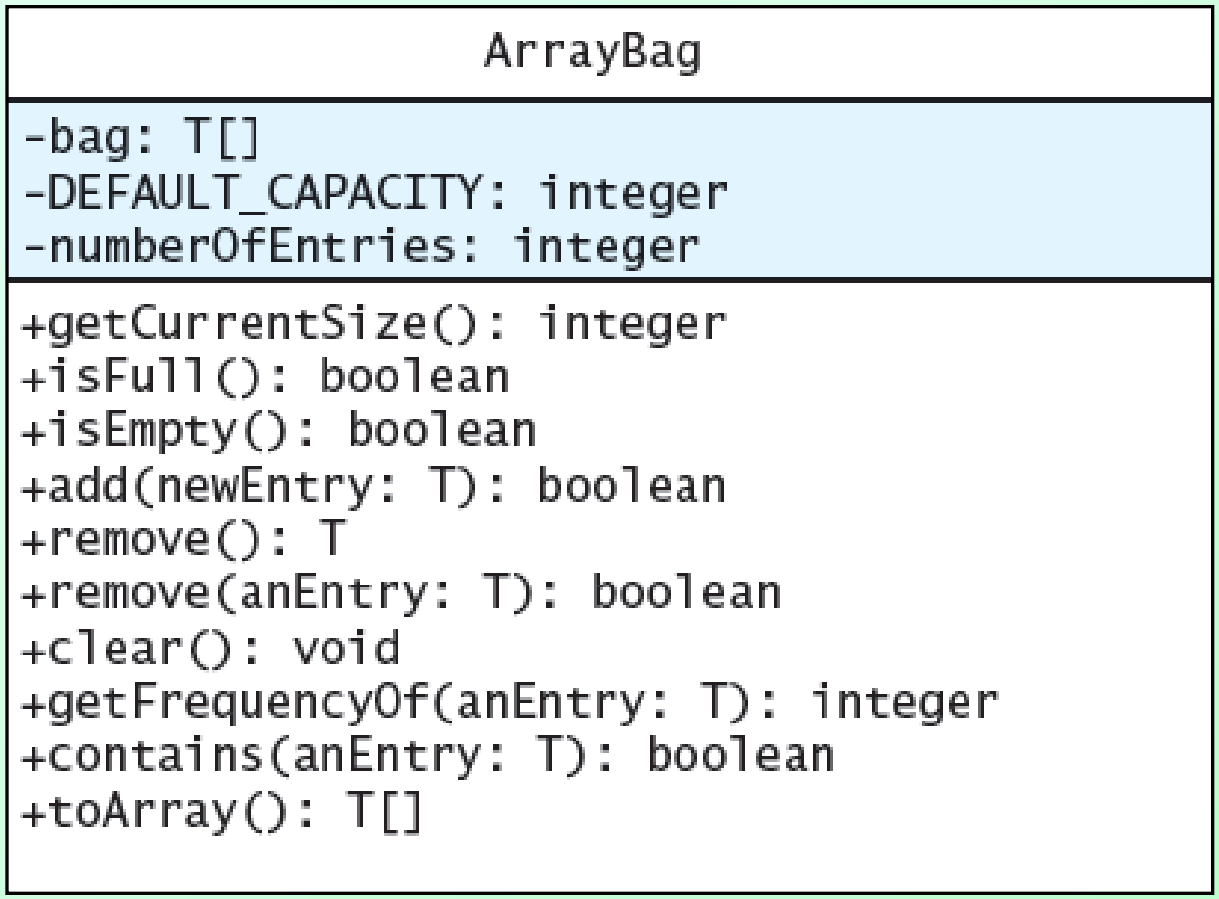


Figure 2-2 UML notation for the class **`ArrayBag`**, including the class's data fields

Group of Core Methods

- Do not attempt to define entire class, then test
- Instead, identify group of core methods
  - Define
  - Test
  - Then finish rest of class
- Note outline, Listing 2-1

Note: Code listing files must be in same folder as PowerPoint files for links to work

Design Decisions

- When array bag is partially full
  - Which array elements should contain entries?
- Options
  - Start at element 0 or element 1 ?
  - Require elements to be sequential?

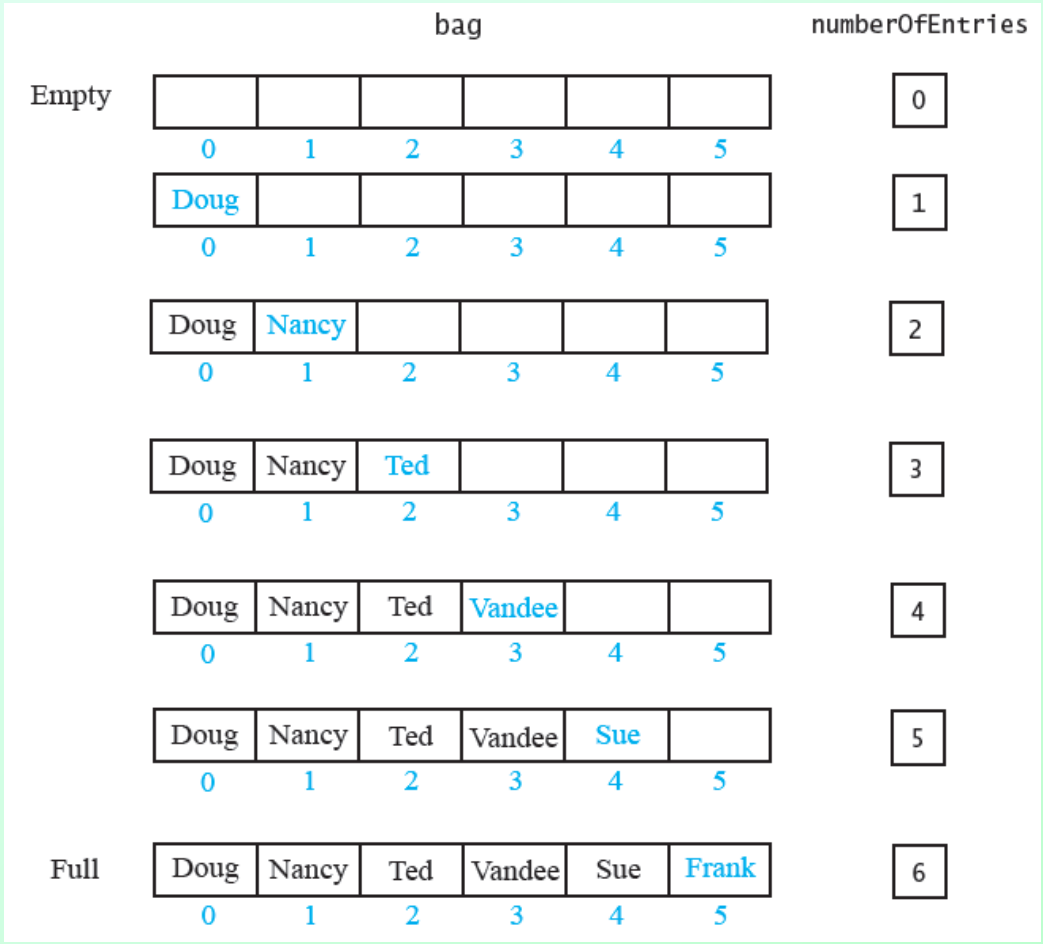


Figure 2-3 Adding entries to an array that represents a bag, whose capacity is six, until it becomes full

`Add` Method

```
/** 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 (isFull())
   {
      result = false;
   }
   else
   {  // assertion: result is true here
      bag[numberOfEntries] = newEntry;
      numberOfEntries++;
   } // end if

   return result;
} // end add
```

- Note: entries in no order

The entries in a bag have no particular order. Thus, the method add can place a new entry into a convenient element of the array bag. In the above definition of add, that element is the one immediately after the last element used.

A convenient way to think about an Array of objects

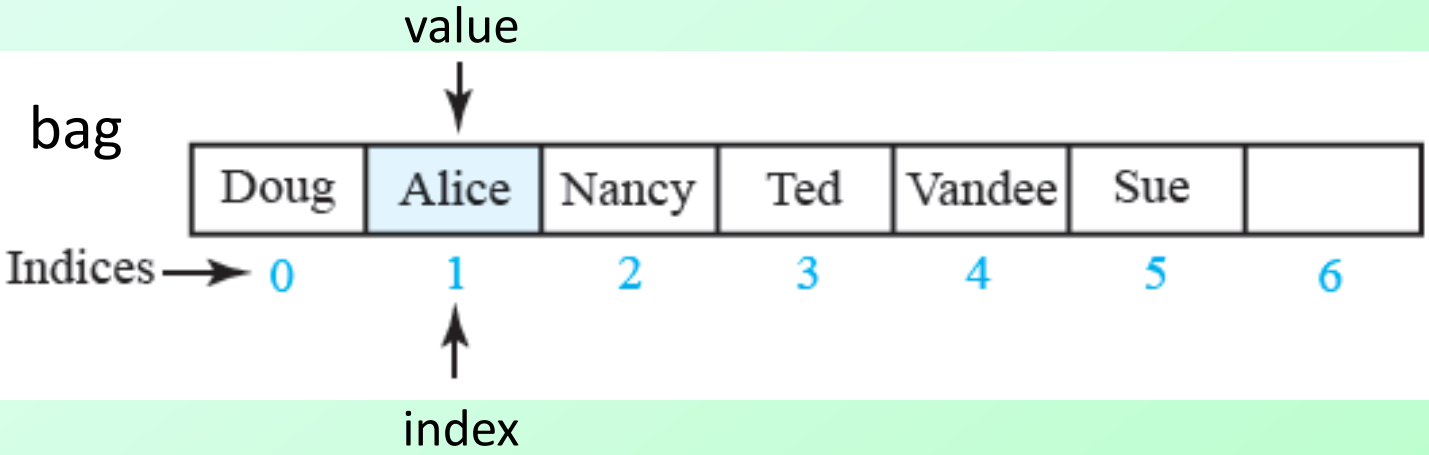


Figure 2-4A An array of objects the way we like to think of them, where each cell "holds" the object (not true though!)

How Java actually represents an array of objects

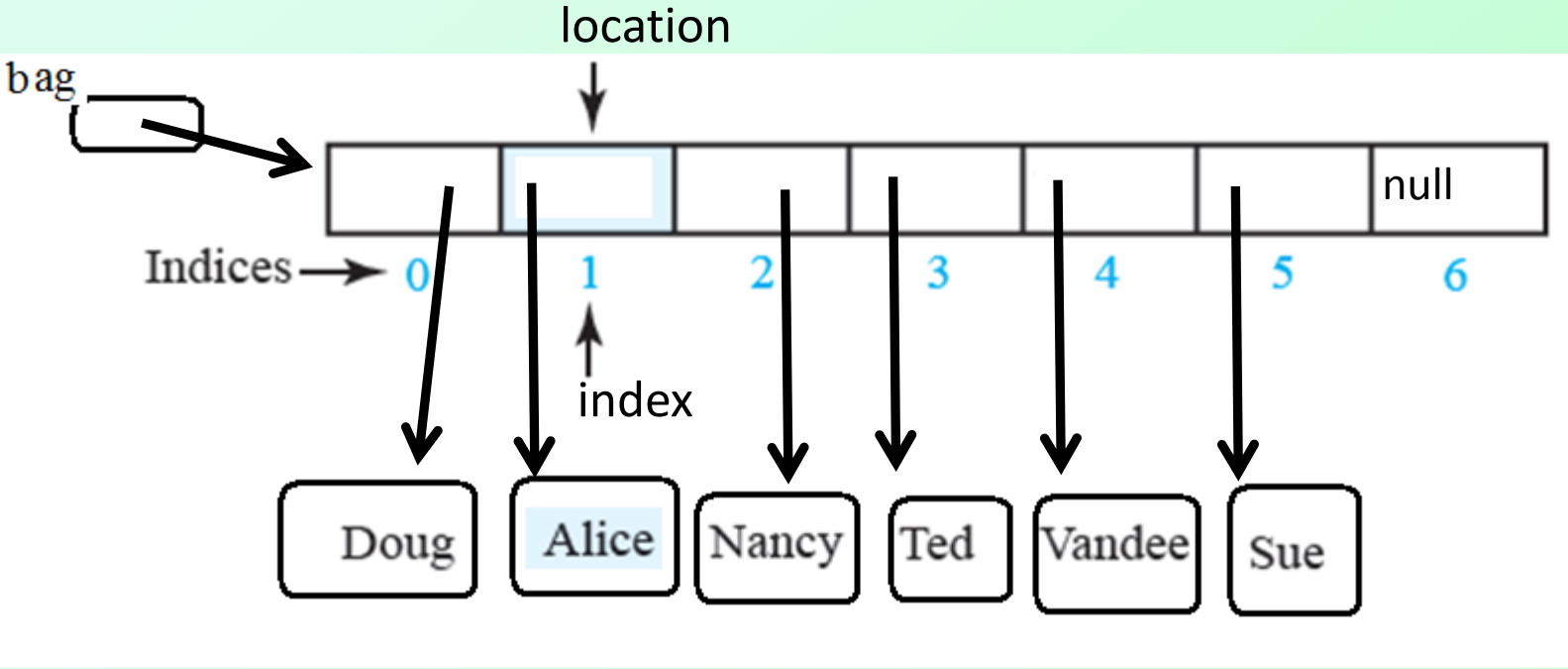


Figure 2-4B An array of objects ACTUALLY contains references to those objects not the objects themselves. Each cell holds a location in memory where the object can be found.

Draw a memory map of the following ArrayBag object

```
Bag<String> aBag = new ArrayBag<String>();


aBag.add("peas");
aBag.add("tofu");
aBag.add("carrots");
aBag.add("celery");
aBag.remove("tofu");
```

```java
/** Sees whether this bag is full.
    @return true if the bag is full, or false if not */
public boolean isFull()
{
   return numberOfEntries == bag.length;
} // end isFull
```

Method **`isFull`**

```java
/** 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**

Question 4 In the previous method `toArray`, does the value of `numberOfEntries` equal `bag.length` in general?

Question 5 Suppose that the previous method `toArray` gave the new array `result` the same length as the array `bag`. How would a client get the number of entries in the returned array?

Question 6 Suppose that the previous method `toArray` returned the array `bag` instead of returning a new array such as `result`. If `myBag` is a bag of five entries, what effect would the following statements have on the array `bag` and the field `numberOfEntries`?

```
Object[] bagArray = myBag.toArray();
bagArray[0] = null;
```

Question 7 The body of the method `toArray` could consist of one `return` statement if you call the method `Arrays.copyOf`. Make this change to `toArray`.

4. No. The two values are equal only when a bag is full.

5. If the client contained a statement such as

```
Object[] bagContents = myBag.toArray();
```

`myBag.getCurrentSize()` would be the number of entries in the array `bagContents`. With the proposed design, `bagContents.length` could be larger than the number of entries in the bag.

6. The statements set the first element of bag to `null`. The value of `numberOfEntries` does not change, so it is 5.

7.
```
public T[] toArray()
{
   return Arrays.copyOf(bag, bag.length);
} // end toArray
```

Design Decisions

- Should `toArray` return the array `bag` or a copy?
  - Best to return a copy … think about why.
- Temporarily make stub methods for testing at this stage
- View test program, Listing 2-2
  - Output

More Methods

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

```
/** Sees whether this bag is empty.
    @return true if the bag is empty, or false if not */
public boolean isEmpty()
{
   return numberOfEntries == 0;
} // end isEmpty

/** Gets the current number of entries in this bag.
    @return the integer number of entries currently in the bag */
public int getCurrentSize()
{
   return numberOfEntries;
} // end getCurrentSize
```

More Methods

- Method **`getFrequencyOf`**

```
/** Counts the number of times a given entry appears in this bag.
    @param anEntry  the entry to be counted
    @return the number of times anEntry appears in the bag */
public int getFrequencyOf(T anEntry)
{
   int counter = 0;

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

   return counter;
} // end getFrequencyOf
```

More Methods

- Method **`contains`**

```
/** Tests whether this bag contains a given entry.
    @param anEntry  the entry to locate
    @return true if the bag contains anEntry, or false otherwise */
public boolean contains(T anEntry)
{
   boolean found = false;

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

   return found;
} // end contains
```

Question 8 What is the result of executing the following statements within the `main` method of `BagDemo1`?

```
ArrayBag<String> aBag = new ArrayBag<String>();
displayBag(aBag);
```

Question 9 The method `contains` could call `getFrequencyOf` instead of executing a loop. That is, you could define the method as follows:

```
public boolean contains(T anEntry)
{
   return getFrequencyOf(anEntry) > 0;
} // end contains
```

What is an advantage and a disadvantage of this definition as compared to the one given in the previous segment?

8. The bag `aBag` is empty. When `displayBag` is called, the statement

```
Object[] bagArray = aBag.toArray();
```

executes. When `toArray` is called, the statement

```
T[] result = (T[])new Object[numberOfEntries];
```

executes. Since `aBag` is empty, `numberOfEntries` is zero. Thus, the new array, `result`, is empty. The loop in `toArray` is skipped and the empty array is returned and assigned to `bagArray`. Since `bagArray.length` is zero, the loop in `displayBag` is skipped. The result of the call `displayBag(aBag)` is simply the line

```
The bag contains
```

9. Advantage: This definition is easier to write, so you are less likely to make a mistake.
Disadvantage: This definition takes more time to execute, if the bag contains more than one occurrence of `anEntry`. Note that the loop in the method `getFrequencyOf` cycles through all of the entries in the bag, whereas the loop in the method `contains`, as given in Segment 2.18, ends as soon as the desired entry is found.

Methods That Remove Entries

- Method clear
  - Remove all entries

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

- Remove last entry in bag
  - How could this be more efficient?

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

   return result;
} // end remove
```

Question 10 Revise the definition of the method `clear` so that it does not call `isEmpty`.
Hint: The `while` statement should have an empty body.

Question 11 Consider the following definition of `clear`:

```
public void clear()
{
   numberOfEntries = 0;
} // end clear
```

What is a disadvantage of this definition as compared to the one shown in Segment 2.20?

Question 12 Why does the method `remove` set `bag[numberOfEntries]` to `null`?

Question 13 The previous `remove` method removes the last entry in the array `bag`. Why might removing a different entry be more difficult to accomplish?

10.

```
public void clear()
{
   while (remove() != null)
   {
   } // end while
} // end clear
```

11. Although the bag will appear empty to both the client and the other methods in `ArrayBag`, the references to the removed objects will remain in the array `bag`. Thus, the memory associated with these objects will not be deallocated.

12. By setting `bag[numberOfEntries]` to `null`, the method causes the memory assigned to the deleted entry to be recycled, unless another reference to that entry exists in the client.

13. An entry in the array `bag`, other than the last one, would be set to `null`. The remaining entries would no longer be in consecutive elements of the array. We could either rearrange the entries to get rid of the `null` entry or modify other methods to skip any `null` entry.

Method to remove a specified entry

First, have to locate the index of the desired entry

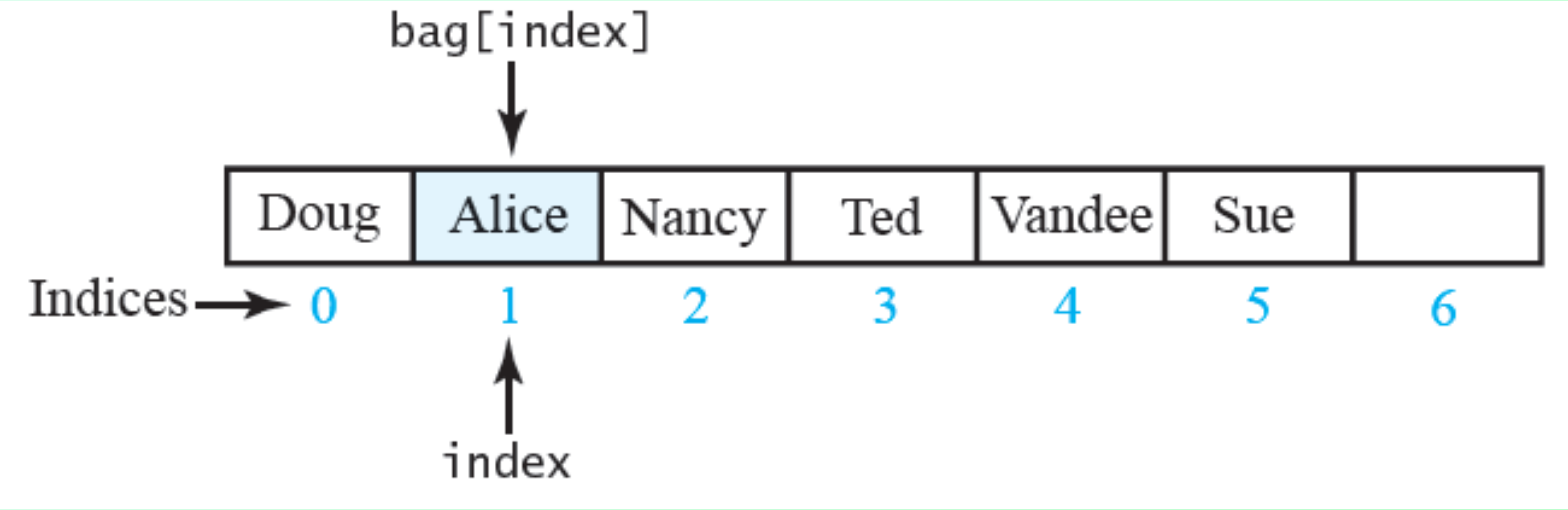


Figure 2-5 The array bag after a successful search for the string "Alice"

Once index is found, remove and shift values above to fill gap

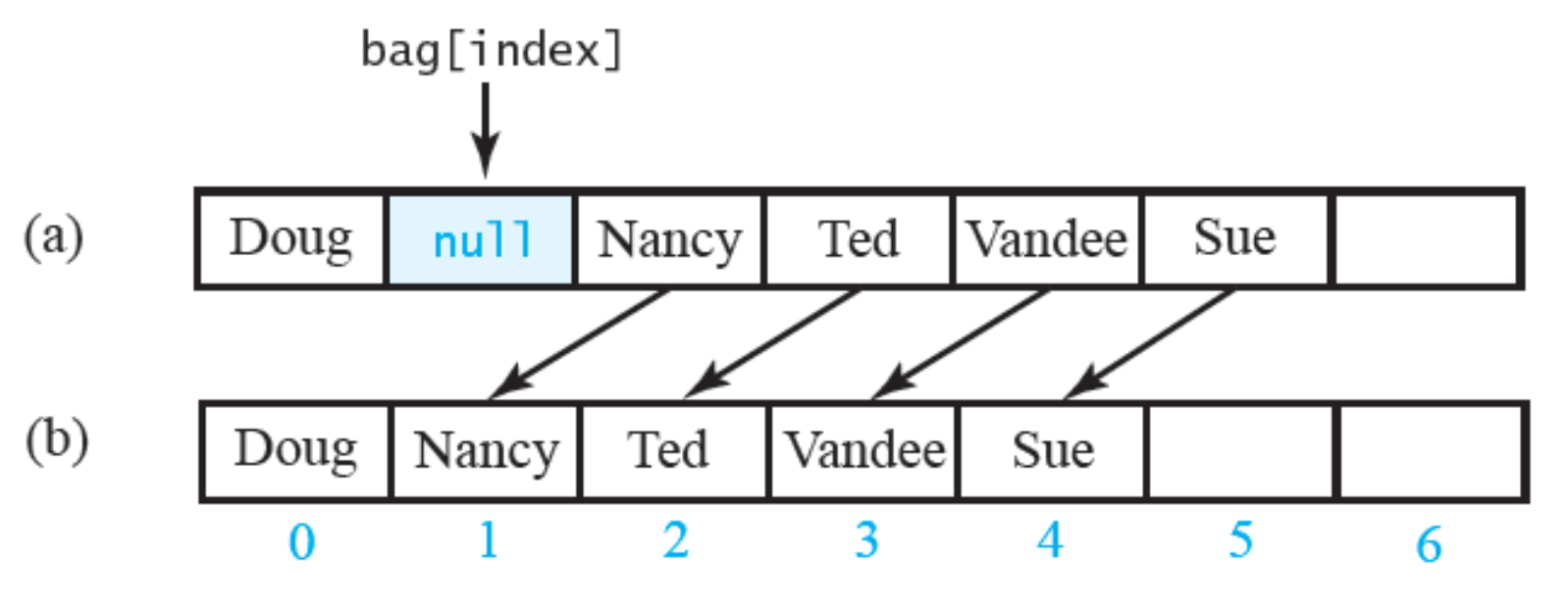


Figure 2-6 (a) A gap in the array bag after setting the entry in bag[index] to null; (b) the array after shifting subsequent entries to avoid a gap

Or, move the last value into the gap

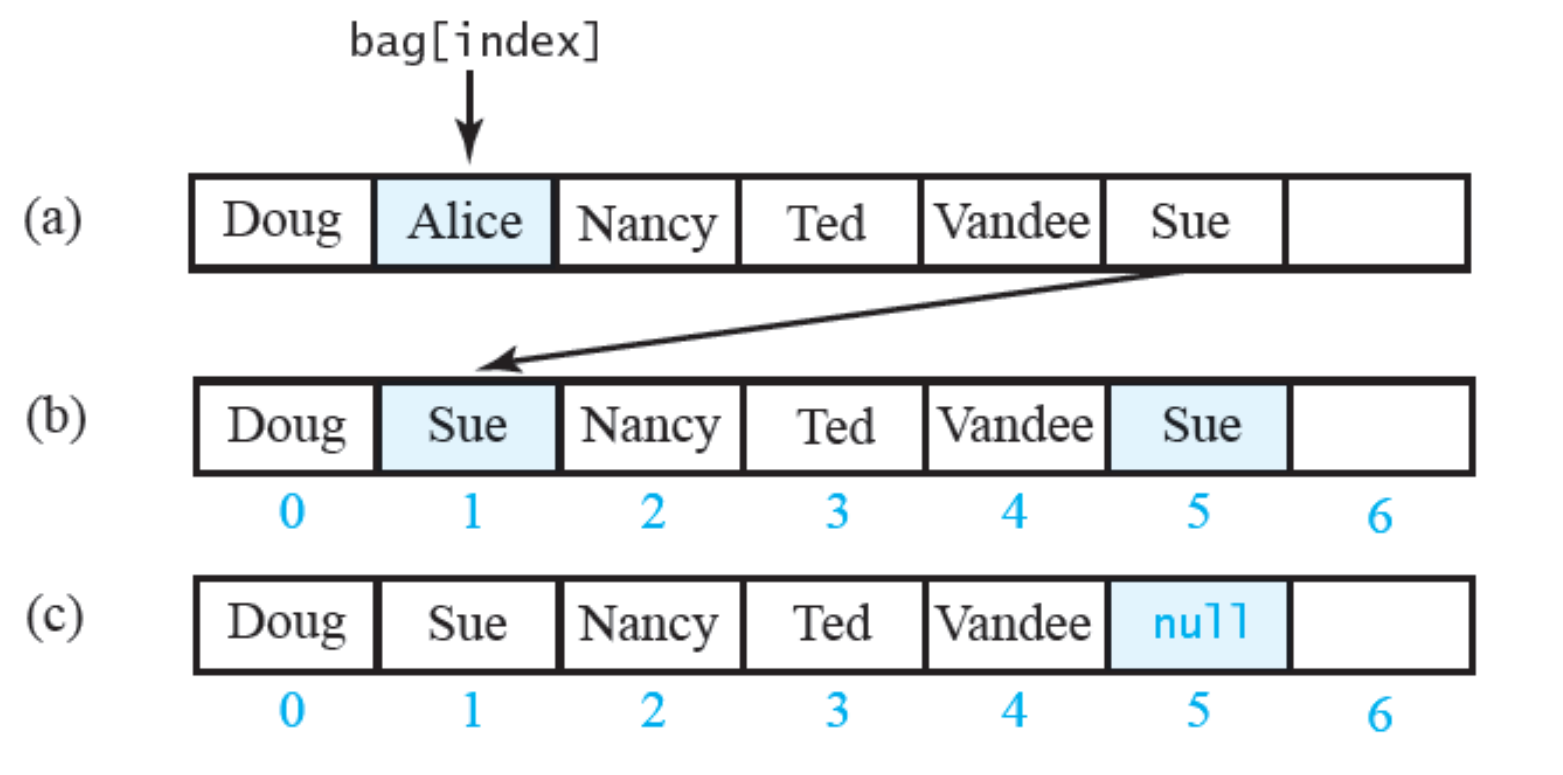


Figure 2-7 Avoiding a gap in the array while removing an entry

Methods That Remove Entries

- Method to remove a specified entry

```
/** 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)
{
   int index = getIndexOf(anEntry);
   T result = removeEntry(index);
   return anEntry.equals(result);
} // end remove
```

  - Assumes presence of private **`getIndexOf`** method and private **`removeEntry`** method

Finding the index of anEntry

```
// Locates a given entry within the array bag.
// Returns the index of the entry, if located, or -1 otherwise.
private int getIndexOf(T anEntry)
{
   int where = -1;
   boolean found = false;

   for (int index = 0; !found && (index < numberOfEntries); index++)
   {
      if (anEntry.equals(bag[index]))
      {
         found = true;
         where = index;
      } // end if
   } // end for

   // 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
```

Removing Entry at givenIndex

```
// Removes and returns the entry at a given index within the arraybag.
// If no such entry exists, returns null.
private T removeEntry(int givenIndex)
{
   T result = null;
   if (!isEmpty() && (givenIndex >= 0))
   {
      result = bag[givenIndex];                  // entry to remove
      numberOfEntries--;
      bag[givenIndex] = bag[numberOfEntries];    // replace entry with last entry
      bag[numberOfEntries] = null;               // remove last entry
   } // end if

   return result;
} // end removeEntry
```

Question 14 Can the `return` statement in the previous definition of `remove` be written as follows?

a.
```
return result.equals(anEntry);
```
b.
```
return result != null;
```

Question 15 The array `bag` in `ArrayBag` contains the entries in the bag `aBag`. If `bag` contains the strings `"A"`, `"A"`, `"B"`, `"A"`, `"C"`, why does `aBag.remove("B")` change the contents of `bag` to `"A"`, `"A"`, `"C"`, `"A"`, `null` instead of either `"A"`, `"A"`, `"A"`, `"C"`, `null` or `"A"`, `"A"`, `null`, `"A"`, `"C"`?

14. **a.** No. If `result` were `null`—and that is quite possible—a `NullPointerException` would occur.
b. Yes.

15. After locating `"B"` in the bag, the `remove` method replaces it with the last relevant entry in the array `bag`, which is `"C"`. It then replaces that last entry with `null`. Although we could define `remove` to result in either of the two other possibilities given in the question, both choices are inferior. For example, to get `"A"`, `"A"`, `"A"`, `"C"`, `null`, `remove` would shift the array elements, requiring more execution time. Leaving a gap in the array, such as `"A"`, `"A"`, `null`, `"A"`, `"C"`, is easy for `remove` to do but complicates the logic of the remaining methods.

Resizing Array

- Need to accommodate more elements than originally specified for bag

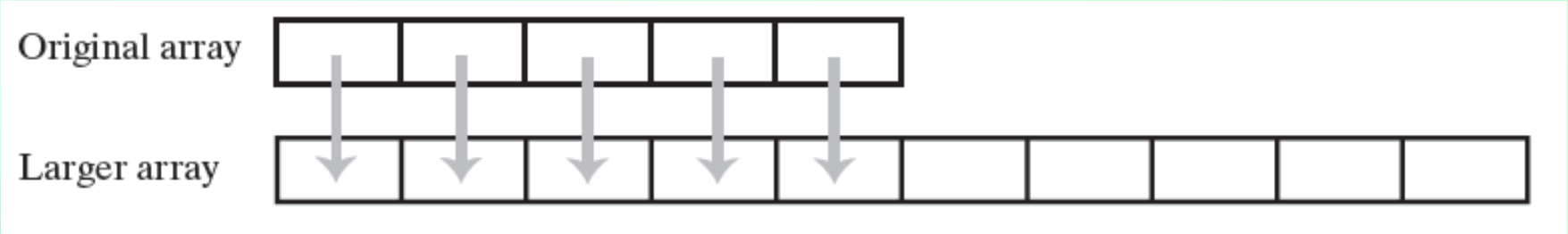


Figure 2-8 Resizing an array copies its contents to a larger second array

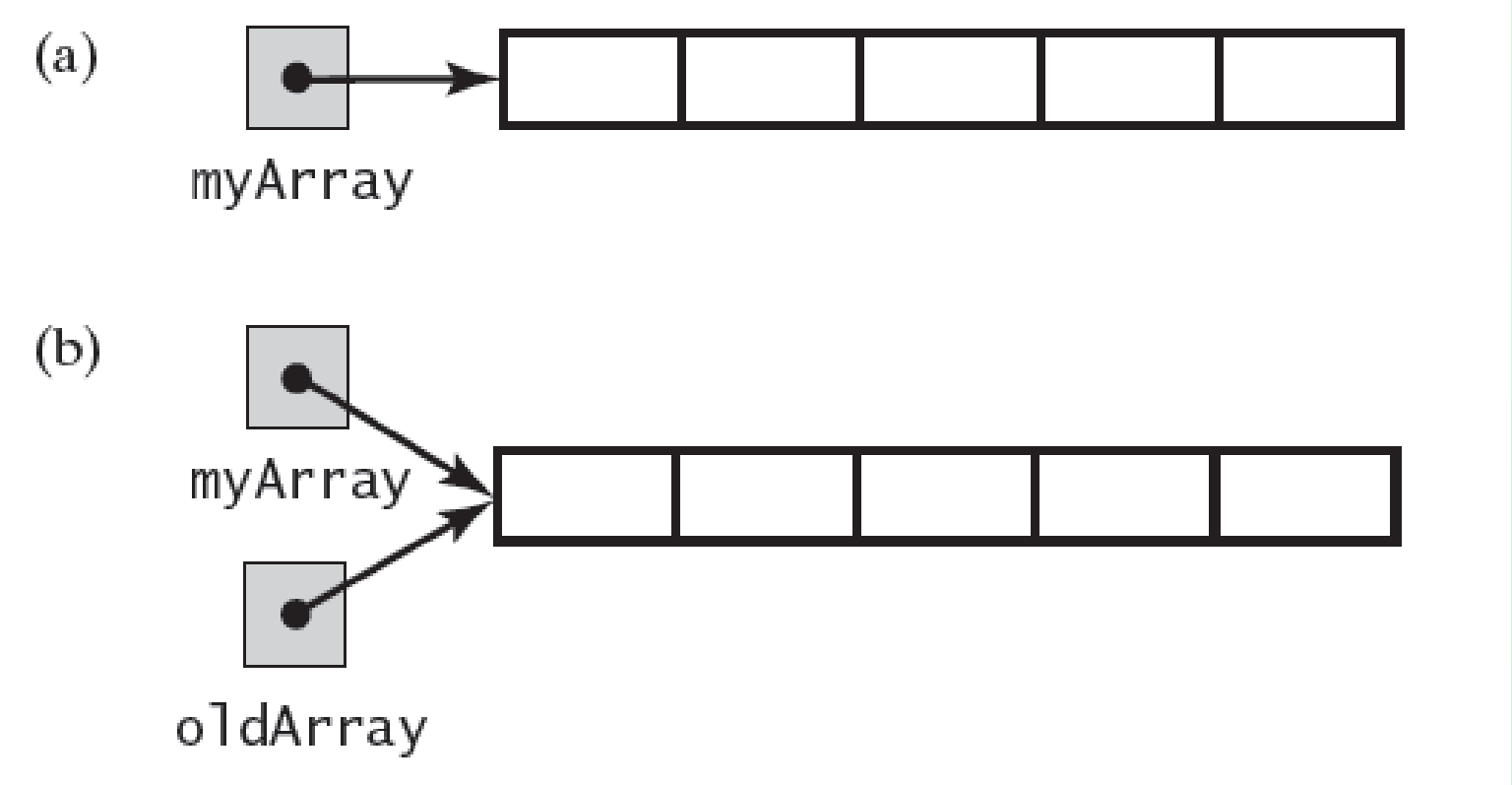


Figure 2-9 (a) An array; (b) two references to the same array;

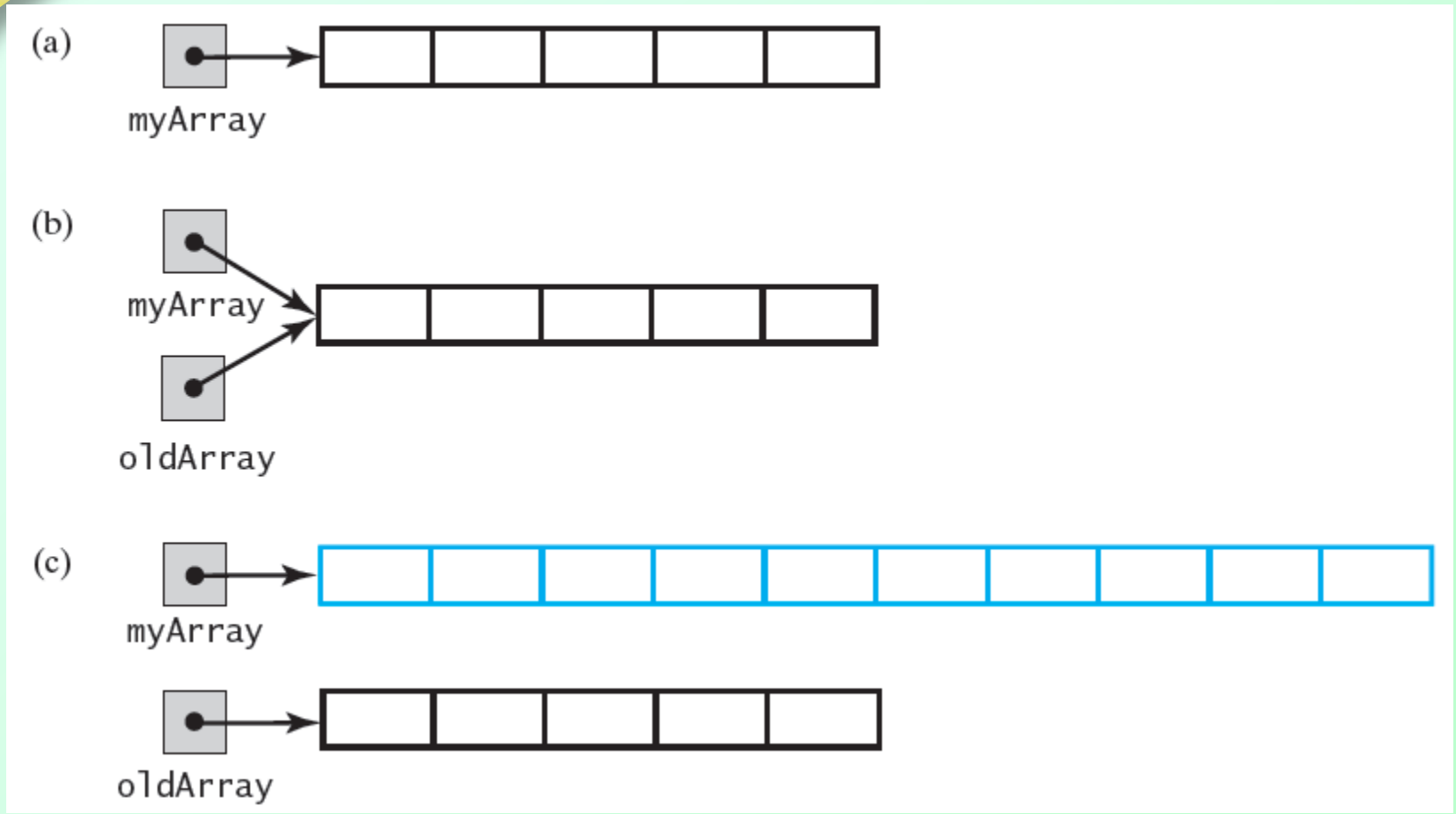


Figure 2-9 (a) An array; (b) two references to the same array; (c) the original array variable now references a new, larger array;

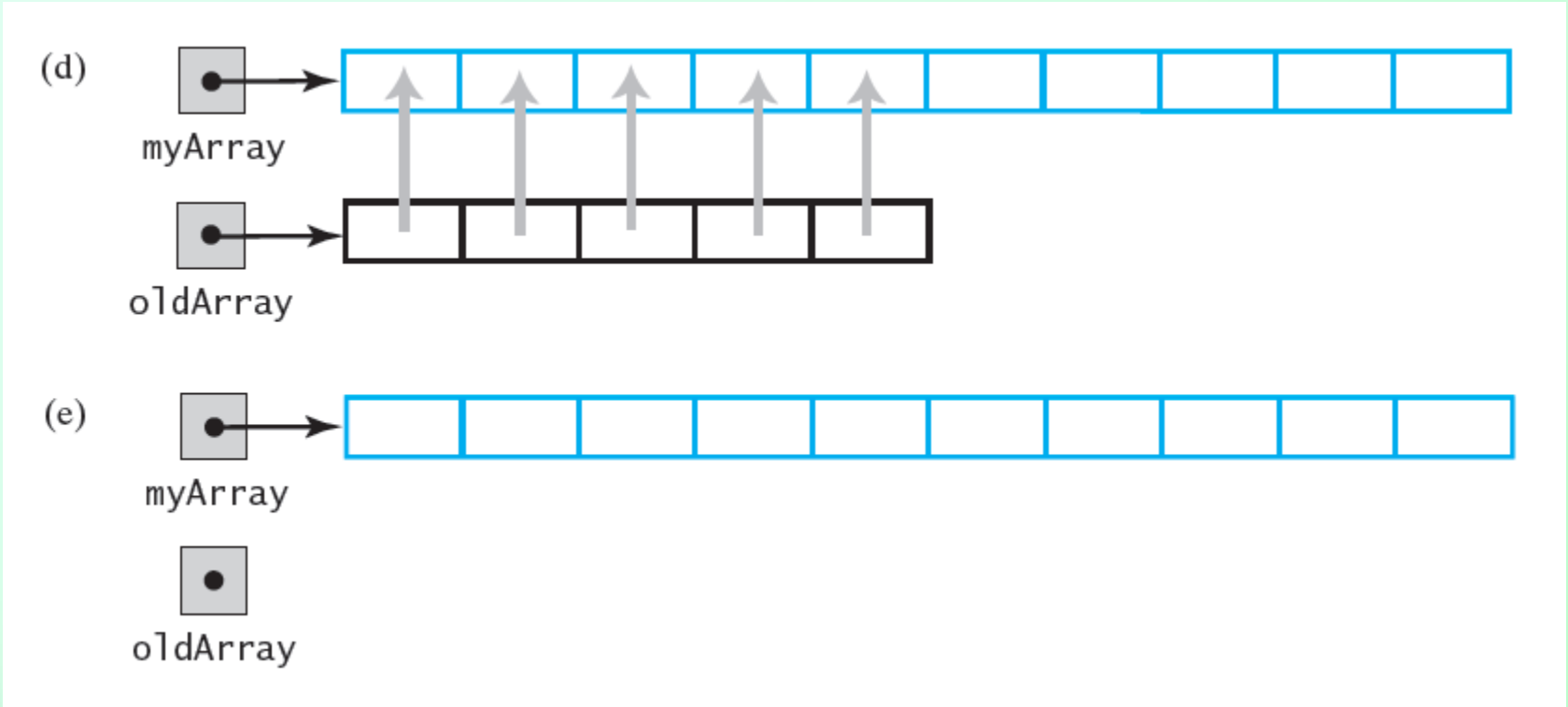


Figure 2-9 (d) the entries in the original array are copied to the new array; (e) the original array is discarded

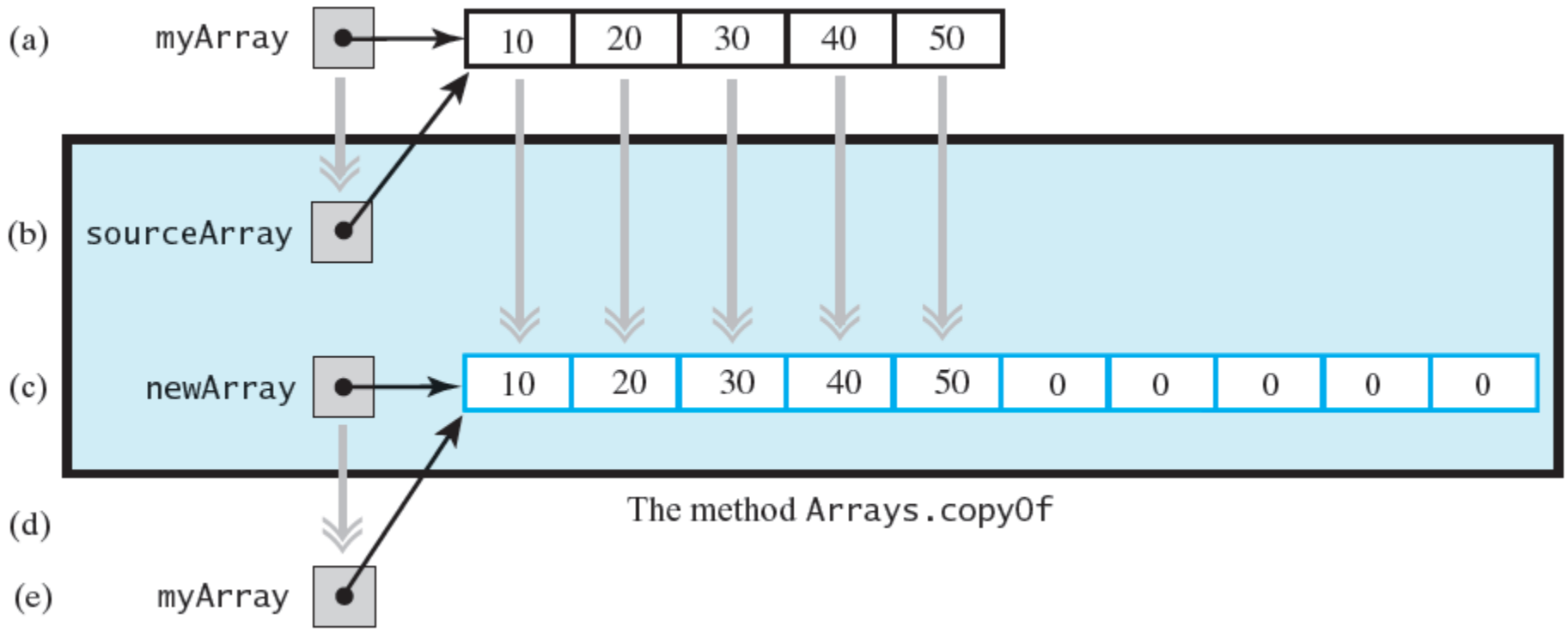


FIGURE 2-10 The effect of the statement **`myArray = Arrays.copyOf (myArray, 2 * myArray.length);`**
(a) The argument array; (b) the parameter that references the argument array; (c) a new, larger array that gets the contents of the argument array; (d) the return value that references the new array; (e) the argument variable is assigned the return value

Question 18 Consider the array of strings that the following statement defines:

```
String[] text = {"cat", "dog", "bird", "snake"};
```

What Java statements will increase the capacity of the array `text` by five elements without altering its current contents?

Question 19 Consider an array `text` of strings. If the number of strings placed into this array is less than its length (capacity), how could you decrease the array's length without altering its current contents? Assume that the number of strings is in the variable `size`.

18.
```
text = Arrays.copyOf(text, text.length + 5);
```

or

```
String[] origText = text;
text = new String[text.length + 5];
System.arraycopy(origText, 0, text, 0, origText.length);
```

19.
```
text = Arrays.copyOf(text, size);
```

A New Implementation of a Bag

- Change name of class to **`ResizableArrayBag`**, distinguish between implementations.
- Remove modifier final from declaration of array bag to enable resizing.
- Change the name of constant **`DEFAULT_CAPACITY`** to **`DEFAULT_INITIAL_CAPACITY`**.

A New Implementation of a Bag

- Although unnecessary, change clarifies new purpose of constant,
  - Bag's capacity will increase as necessary.
  - Make same change in default constructor
- Change names of constructors to match new class name.

```java
public class ResizableArrayBag<T> implements BagInterface<T>
{
    private T[] bag; // cannot be final due to doubling
    private static final int DEFAULT_INITIAL_CAPACITY = 25; // in
    private int numberOfEntries;

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

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

Question 20 What is the definition of a constructor that you could add to the class `ResizableArrayBag` to initialize the bag to the contents of a given array?

Question 21 In the definition of the constructor described in the previous question, is it necessary to copy the entries from the argument array to the array `bag`, or would a simple assignment (`bag = contents`) be sufficient?

Question 22 What is an advantage of using an array to organize data? What is a disadvantage?

20.

```
/** Creates a bag containing the given array of entries.
    @param contents  an array of objects */
public ResizableArrayBag(T[] contents)
{
   bag = Arrays.copyOf(contents, contents.length);
   numberOfEntries = contents.length;
} // end constructor
```

21. A simple assignment statement would be a poor choice, since then the client could corrupt the bag's data by using the reference to the array that it passes to the constructor as an argument. Copying the argument array to the array `bag` is necessary to protect the integrity of the bag's data.

22. Advantage: You can access any array location directly if you know its index.
Disadvantages: The array has a fixed size, so you will either waste space or run out of room. Resizing the array avoids the latter disadvantage, but requires you to copy the contents of the original array to a larger array.

A New Implementation of a Bag

- Revise definition of method `add` to always accommodate new entry.
  - Method will never return `false`.
- Revise definition of method `isFull` to always return `false`.
  - A bag will never become full.

A New Implementation of a Bag

- New **`add`** method

```
public boolean add(T newEntry)
{
   ensureCapacity();
   bag[numberOfEntries] = newEntry;
   numberOfEntries++;

   return true;
} // end add
```

  - Assumes method **`ensureCapacity`**

```
// Doubles the size of the array bag if it is full.
private void ensureCapacity()
{
   if (numberOfEntries == bag.length)
      bag = Arrays.copyOf(bag, 2 * bag.length);
} // end ensureCapacity
```

Some thoughts on ArrayBag's

When you use an array to implement the ADT bag,

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