CS2400 - Data Structures and Advanced Programming
Module 3: Bags

Hao Ji

Computer Science Department

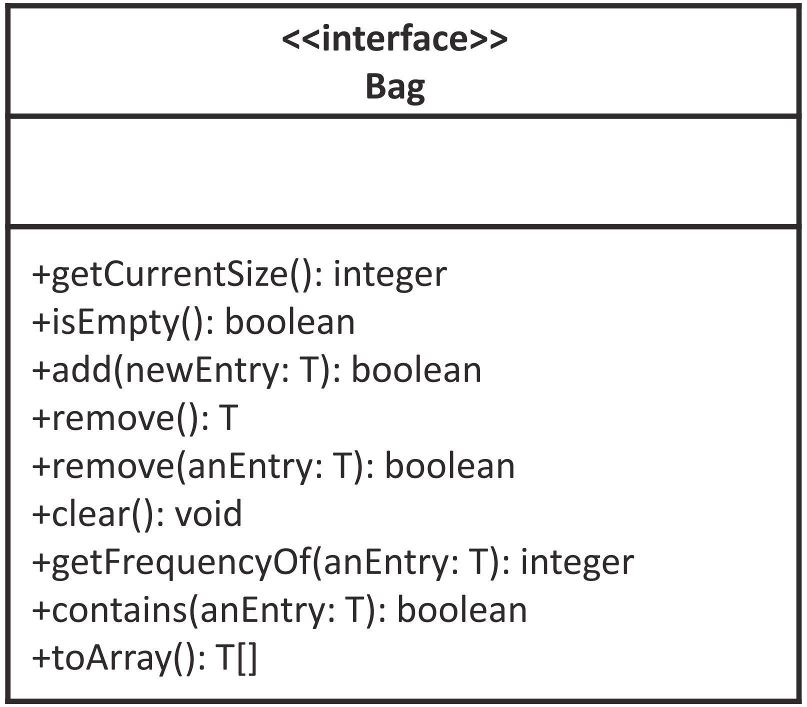
Cal Poly Pomona

The ADT Bag

- Definition
    - A finite collection of objects in no particular order
    - Can contain duplicate items

- Possible behaviors
    - Get number of items
    - Check for empty
    - Add and remove objects

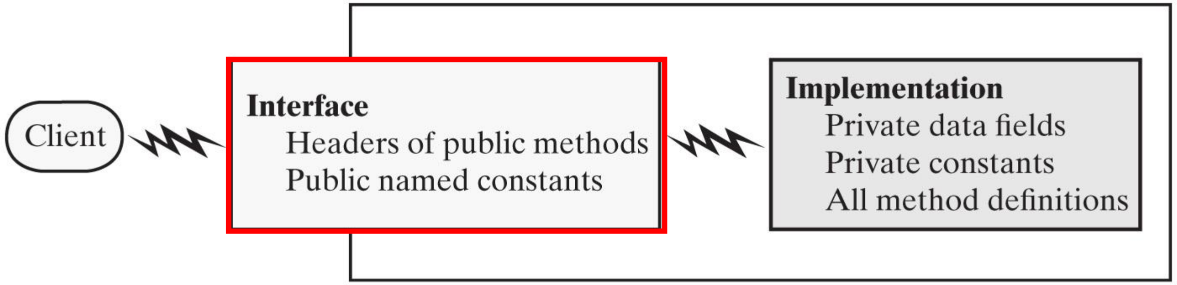
Using UML Notation to Specify a Class



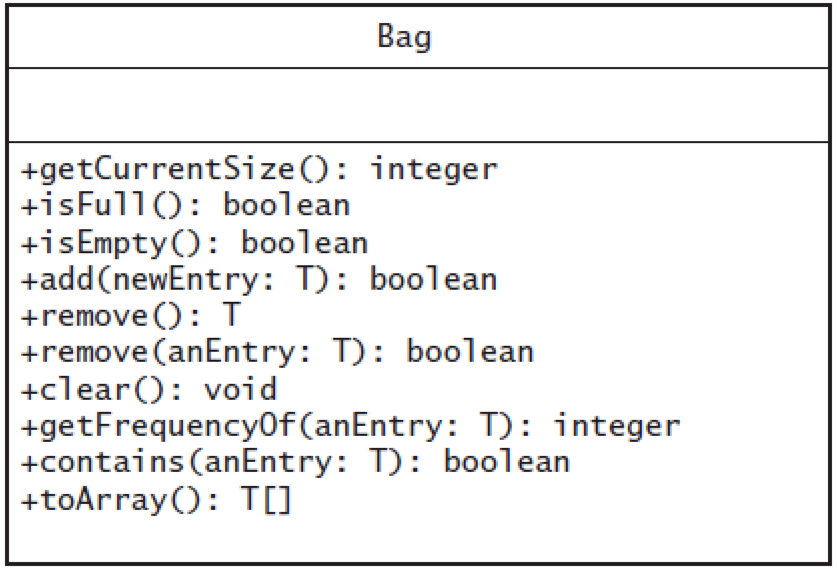
<<interface>>
Bag

+getCurrentSize(): integer	*// Get the number of items currently in the bag*
+isEmpty(): boolean	*// See whether the bag is empty*
+add(newEntry: T): boolean	*// Add a given object to the bag*
+remove(): T	*// Remove an unspecified object from the bag*
+remove(anEntry: T): boolean	*// Remove a particular object from the bag, if possible*
+clear(): void	*// Remove all objects from the bag*
+getFrequencyOf(anEntry: T): integer	*// Count the number of times a certain object occurs in the bag*
+contains(anEntry: T): boolean	*// Test whether the bag contains a particular object*
+toArray(): T[]	*// Look at all objects that are in the bag*

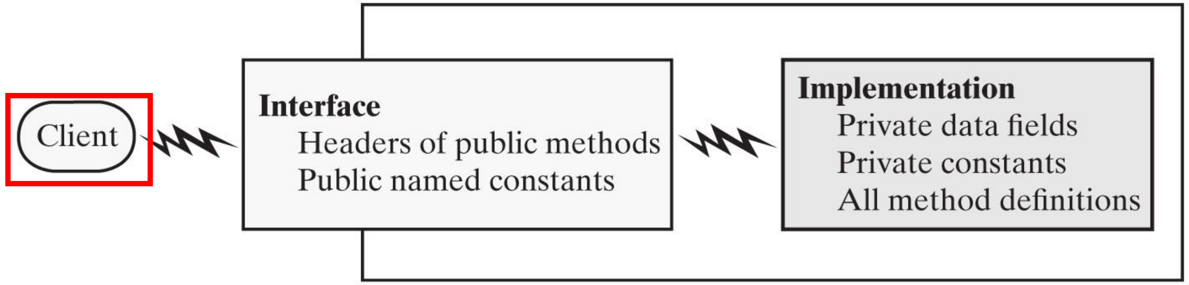
```
/** An interface that describes the operations of a bag of objects. */
public interface BagInterface<T>
{
      /** Gets the current number of entries in this bag.
         @return  The integer number of entries currently in the bag. */
      public int getCurrentSize();

      /** Sees whether this bag is empty.
         @return  True if the bag is empty, or false if not. */
      public boolean isEmpty();

      /** 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);

      /** Removes one unspecified entry from this bag, if possible.
         @return  Either the removed entry, if the removal was successful, or null. */
      public T remove();

      /** Removes one occurrence of a given entry from this bag, if possible.
         @param anEntry  The entry to be removed.
         @return  True if the removal was successful, or false if not. */
     public boolean remove(T anEntry);

      /** Removes all entries from this bag. */
      public void clear();

      /** 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);

      /** Tests whether this bag contains a given entry.
         @param anEntry  The entry to find.
         @return  True if the bag contains anEntry, or false if not. */
      public boolean contains(T anEntry);

      /** Retrieves all entries that are in this bag.
         @return  A newly allocated array of all the entries in the bag. Note: If the bag is empty, the returned array is empty. */
      public T[] toArray();

} // end BagInterface
```



© 2019 Pearson Education, Inc.



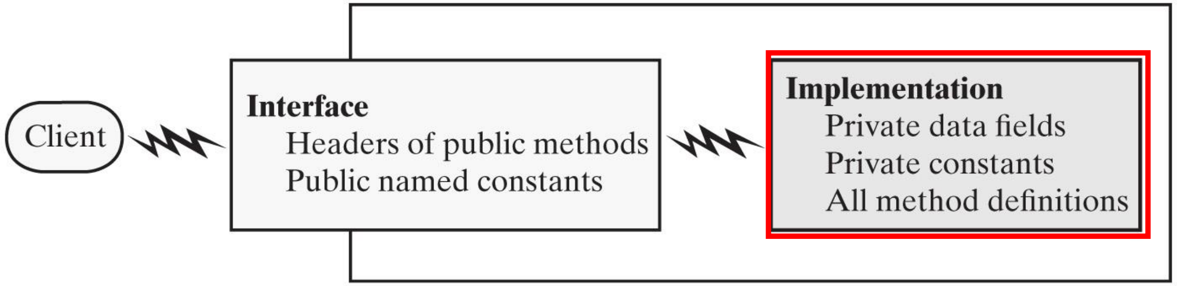
```
Bag

+getCurrentSize(): integer
+isFull(): boolean
+isEmpty(): boolean
+add(newEntry: T): boolean
+remove(): T
+remove(anEntry: T): boolean
+clear(): void
+getFrequencyOf(anEntry: T): integer
+contains(anEntry: T): boolean
+toArray(): T[]
```



```java
/** A class that maintains a shopping cart for an online store. */
public class OnlineShopper
{
      public static void main(String[] args)
      {
            Item[] items = {new Item("Bird feeder", 2050),
                     new Item("Squirrel guard", 1547),
                     new Item("Bird bath", 4499),
                     new Item("Sunflower seeds", 1295)};

            BagInterface<Item> shoppingCart = new ArrayBag<>();
            int totalCost = 0;

             // Statements that add selected items to the shopping cart:
            for (int index = 0; index < items.length; index++)
            {
                      Item nextItem = items[index]; // Simulate getting item from shopper
                      shoppingCart.add(nextItem);
                      totalCost = totalCost + nextItem.getPrice();
            } // end for

            // Simulate checkout
            while (!shoppingCart.isEmpty())
                  System.out.println(shoppingCart.remove());

             System.out.println("Total cost: " + "\t$" + totalCost / 100 + "." + totalCost % 100);
      } // end main
} // end OnlineShopper
```

```java
public class Item
{
  private String description;
  private int    price;

public Item(String productDescription, int productPrice)
{
    description = productDescription;
    price = productPrice;
} // end constructor

public String getDescription()
{
    return description;
} // end getDescription

public int getPrice()
{
    return price;
} // end getPrice

public String toString()
{
    return description + "\t$" + price / 100 + "." + price % 100;
} // end toString
} // end Item
```

Implementations of a Bag

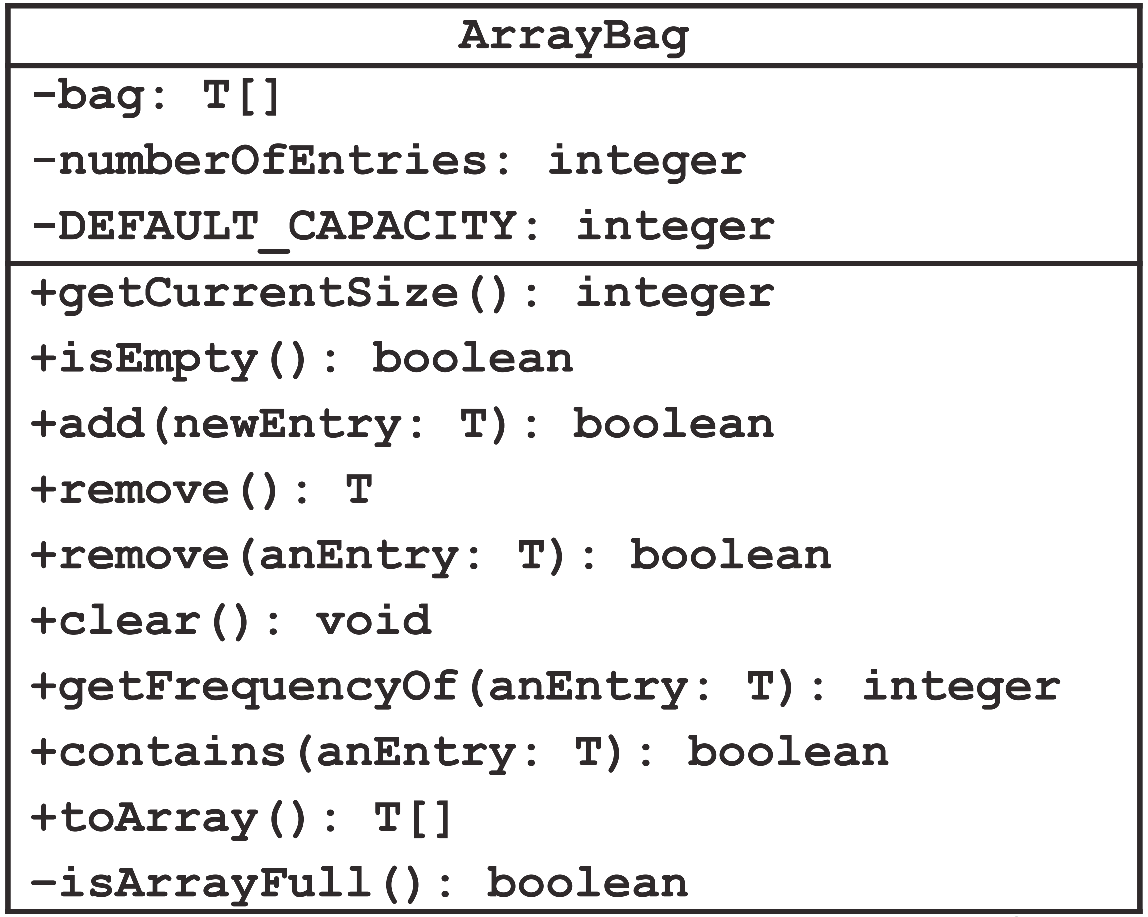


Implementations of a Bag

- Using Fixed-Size Arrays
- Using Array Resizing
- Using Linked Data

Implementations of a Bag

- **Using Fixed-Size Arrays**
- Using Array Resizing
- Using Linked Data



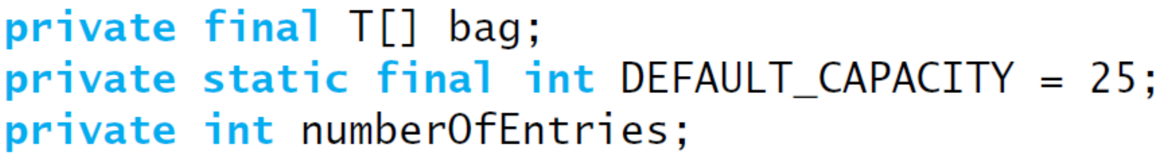
ArrayBag

-bag: T[]
-numberOfEntries: integer
-DEFAULT_CAPACITY: integer

+getCurrentSize(): integer
+isEmpty(): boolean
+add(newEntry: T): boolean
+remove(): T
+remove(anEntry: T): boolean
+clear(): void
+getFrequencyOf(anEntry: T): integer
+contains(anEntry: T): boolean
+toArray(): T[]
-isArrayFull(): boolean

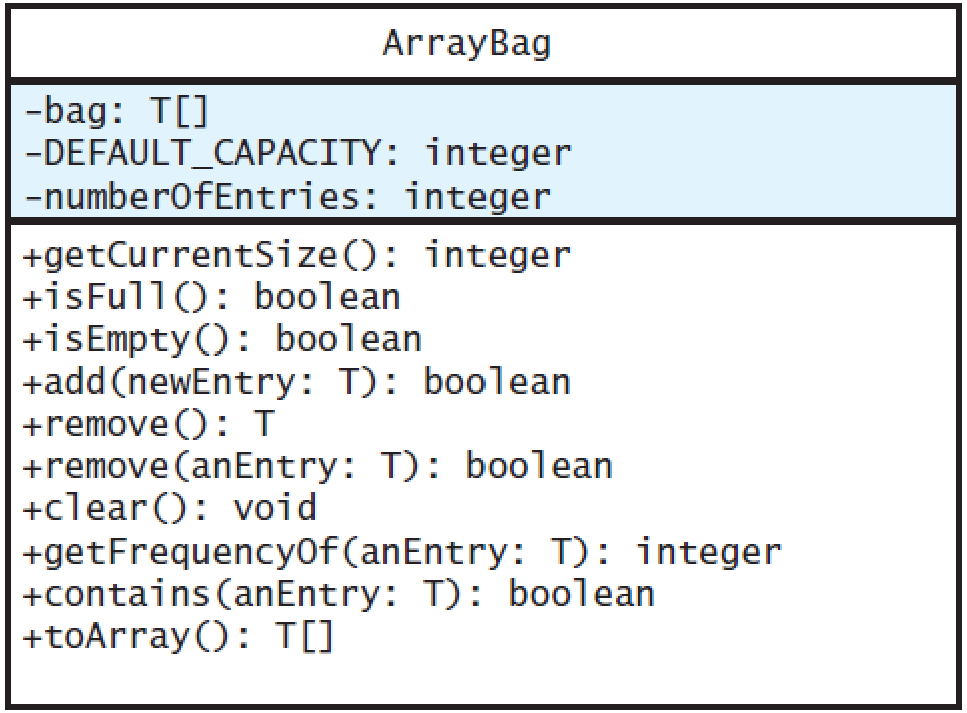
Bag Implementations That Use Arrays

- Private Data Fields



```
private final T[] bag;
private static final int DEFAULT_CAPACITY = 25;
private int numberOfEntries;
```

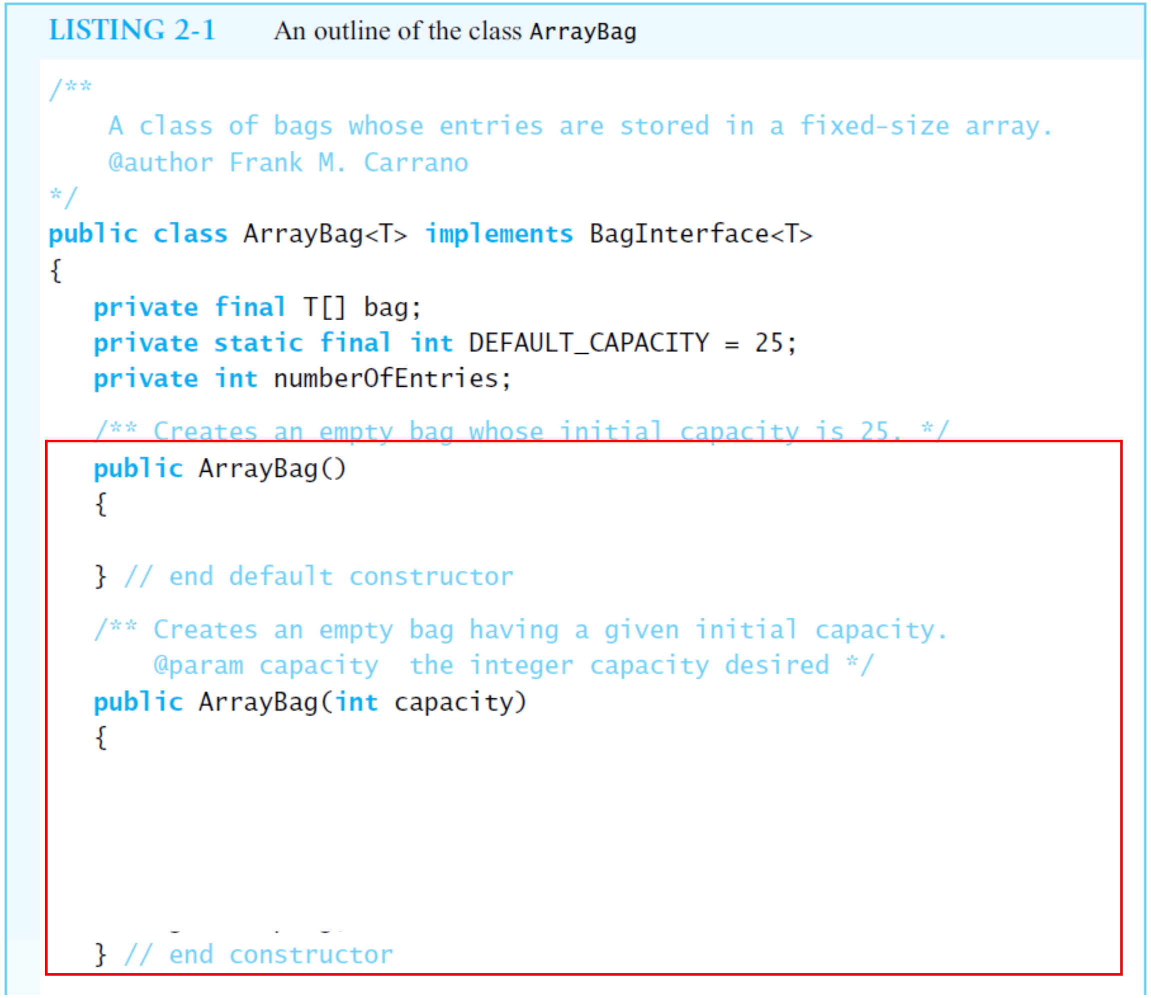
- By declaring the array bag as a final data members of the class ArrayBag, we know that the reference to the array in the variable *bag* cannot change.



```
ArrayBag
-bag: T[]
-DEFAULT_CAPACITY: integer
-numberOfEntries: integer
+getCurrentSize(): integer
+isFull(): boolean
+isEmpty(): boolean
+add(newEntry: T): boolean
+remove(): T
+remove(anEntry: T): boolean
+clear(): void
+getFrequencyOf(anEntry: T): integer
+contains(anEntry: T): boolean
+toArray(): T[]
```

Bag Implementations Th

- **Constructors**
  - Initialize the field *numberOfEntries* to 0.
  - Create the array bag.
    - To create the array, the constructor must specify the array’s length, which is the bag’s capacity.

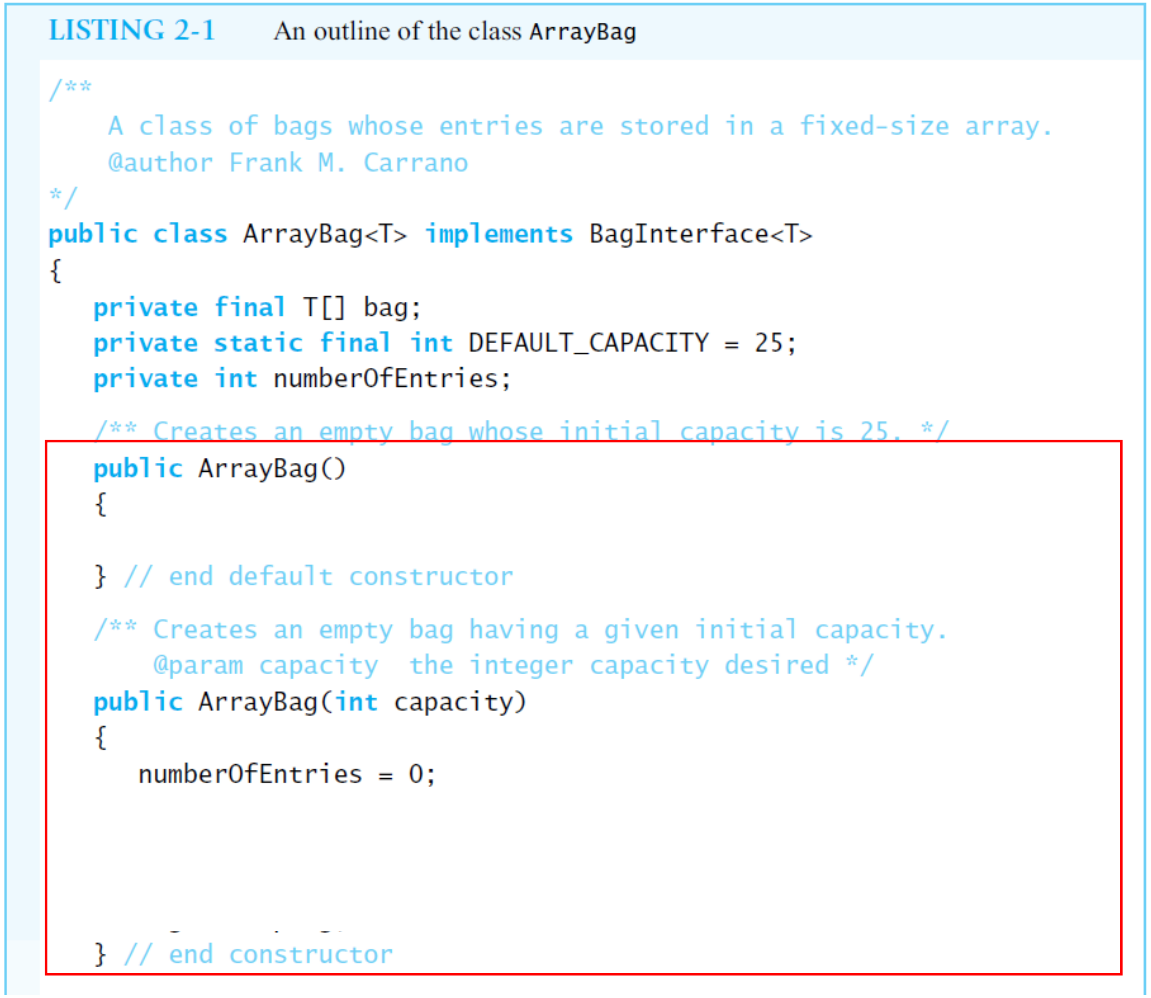


LISTING 2-1 An outline of the class ArrayBag

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

   /** Creates an empty bag whose initial capacity is 25. */
   public ArrayBag()
   {

   } // end default constructor

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

   } // end constructor
```

Bag Implementations Th

- **Constructors**
  - Initialize the field *numberOfEntries* to 0.
  - Create the array bag.
    - To create the array, the constructor must specify the array's length, which is the bag's capacity.



LISTING 2-1 An outline of the class ArrayBag

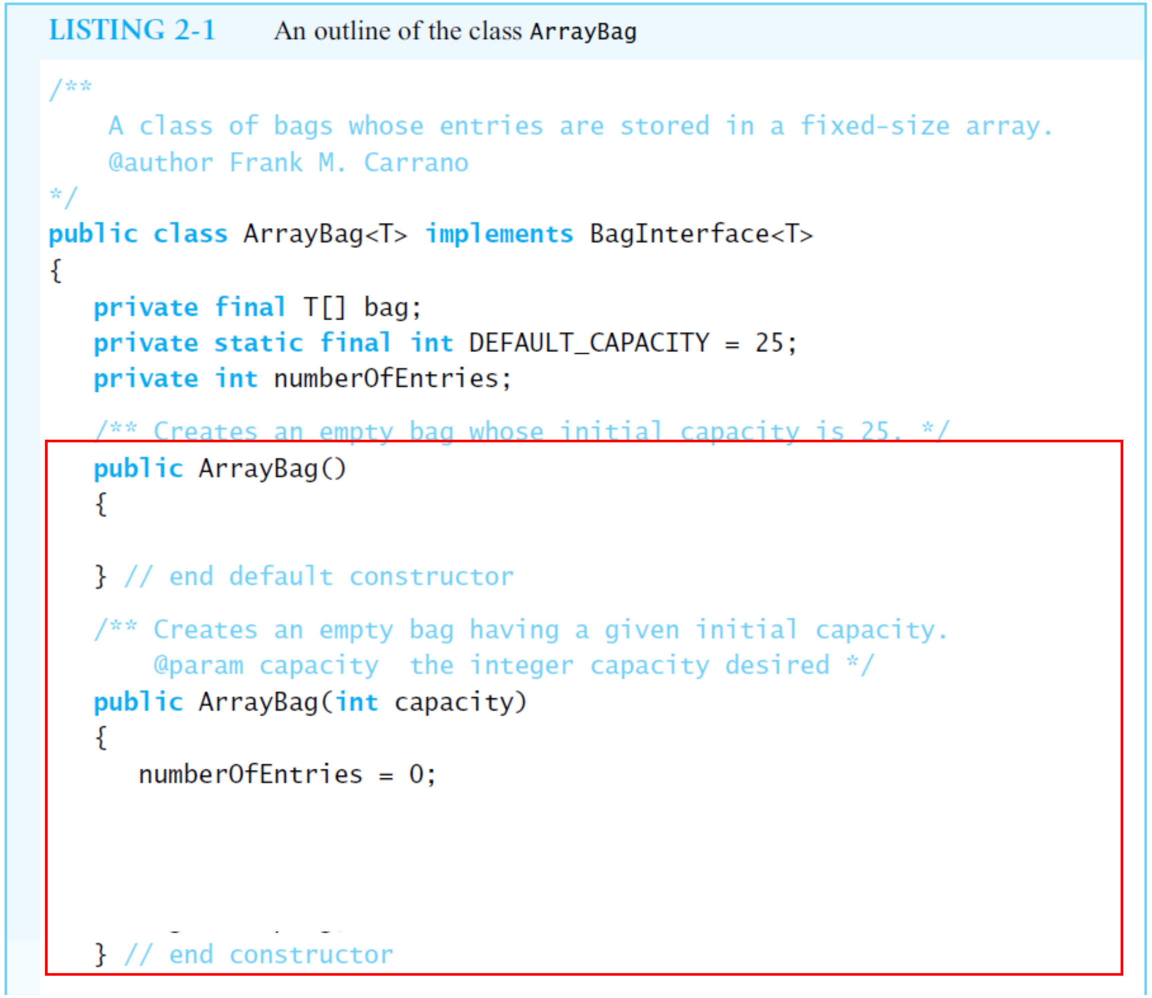
```
/**
    A class of bags whose entries are stored in a fixed-size array.
    @author Frank M. Carrano
*/
public class ArrayBag<T> implements BagInterface<T>
{
   private final T[] bag;
   private static final int DEFAULT_CAPACITY = 25;
   private int numberOfEntries;

   /** Creates an empty bag whose initial capacity is 25. */
   public ArrayBag()
   {

   } // end default constructor

   /** Creates an empty bag having a given initial capacity.
       @param capacity  the integer capacity desired */
   public ArrayBag(int capacity)
   {
      numberOfEntries = 0;




   } // end constructor
```

- Option 1: `bag = new T[capacity];`
- Option 2: `bag = new Object[capacity];`
- Option 3: `bag = (T[])new Object[capacity];`
- Option 4:

```
@SuppressWarnings("unchecked")
T[] tempBag = (T[])new Object[capacity];
bag = tempBag;
```

Bag Implementations Th

- **Constructors**
  - Initialize the field *numberOfEntries* to 0.
  - Create the array bag.
    - To create the array, the constructor must specify the array's length, which is the bag's capacity.



LISTING 2-1 An outline of the class ArrayBag

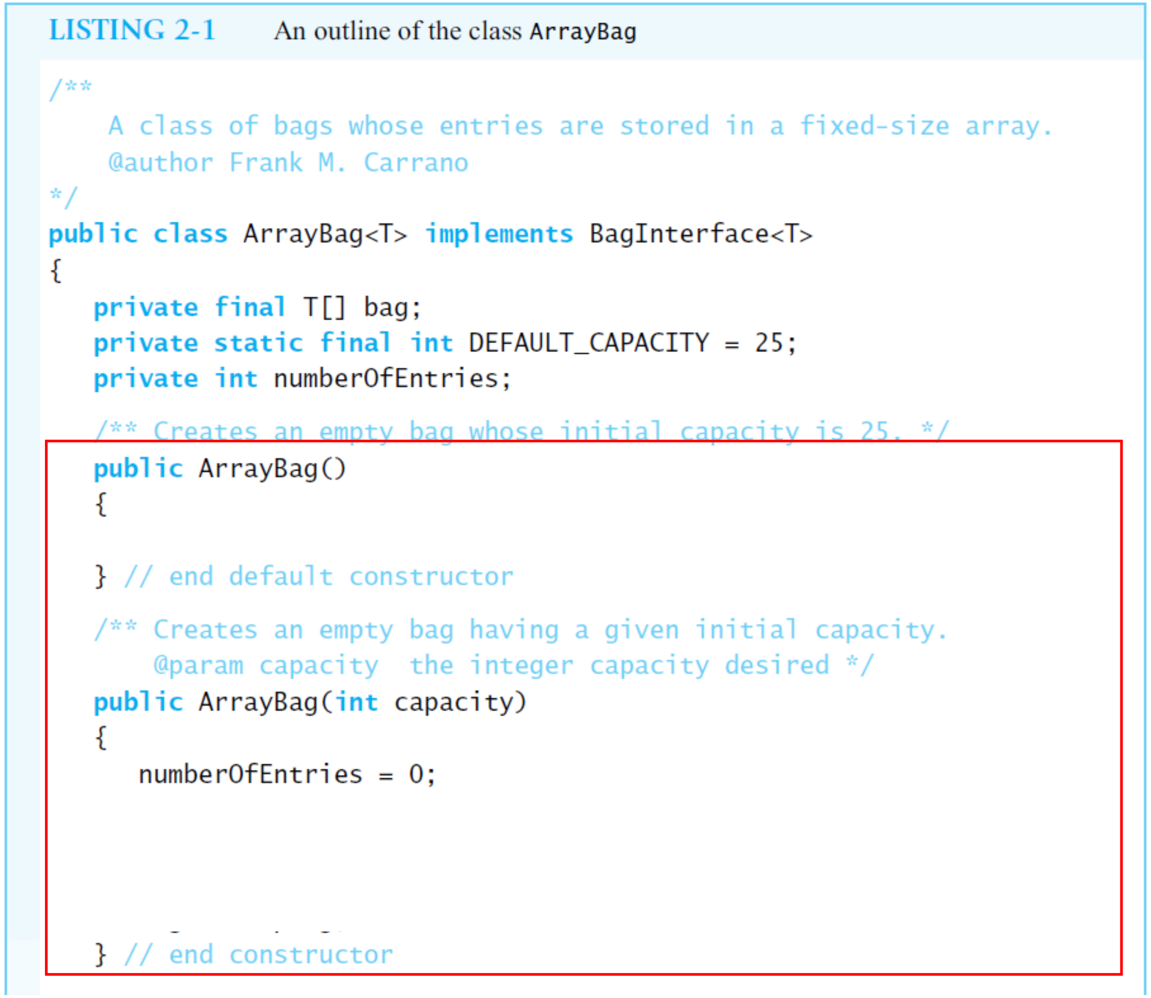
```
/**
   A class of bags whose entries are stored in a fixed-size array.
   @author Frank M. Carrano
*/
public class ArrayBag<T> implements BagInterface<T>
{
   private final T[] bag;
   private static final int DEFAULT_CAPACITY = 25;
   private int numberOfEntries;

   /** Creates an empty bag whose initial capacity is 25. */
   public ArrayBag()
   {

   } // end default constructor

   /** Creates an empty bag having a given initial capacity.
       @param capacity  the integer capacity desired */
   public ArrayBag(int capacity)
   {
      numberOfEntries = 0;




   } // end constructor
```

- Option 1: `bag = new T[capacity]; // SYNTAX ERROR` ✗

You cannot use a generic type when allocating an array

Bag Implementations Th

- **Constructors**
  - Initialize the field *numberOfEntries* to 0.
  - Create the array bag.
    - To create the array, the constructor must specify the array’s length, which is the bag’s capacity.



LISTING 2-1 An outline of the class ArrayBag

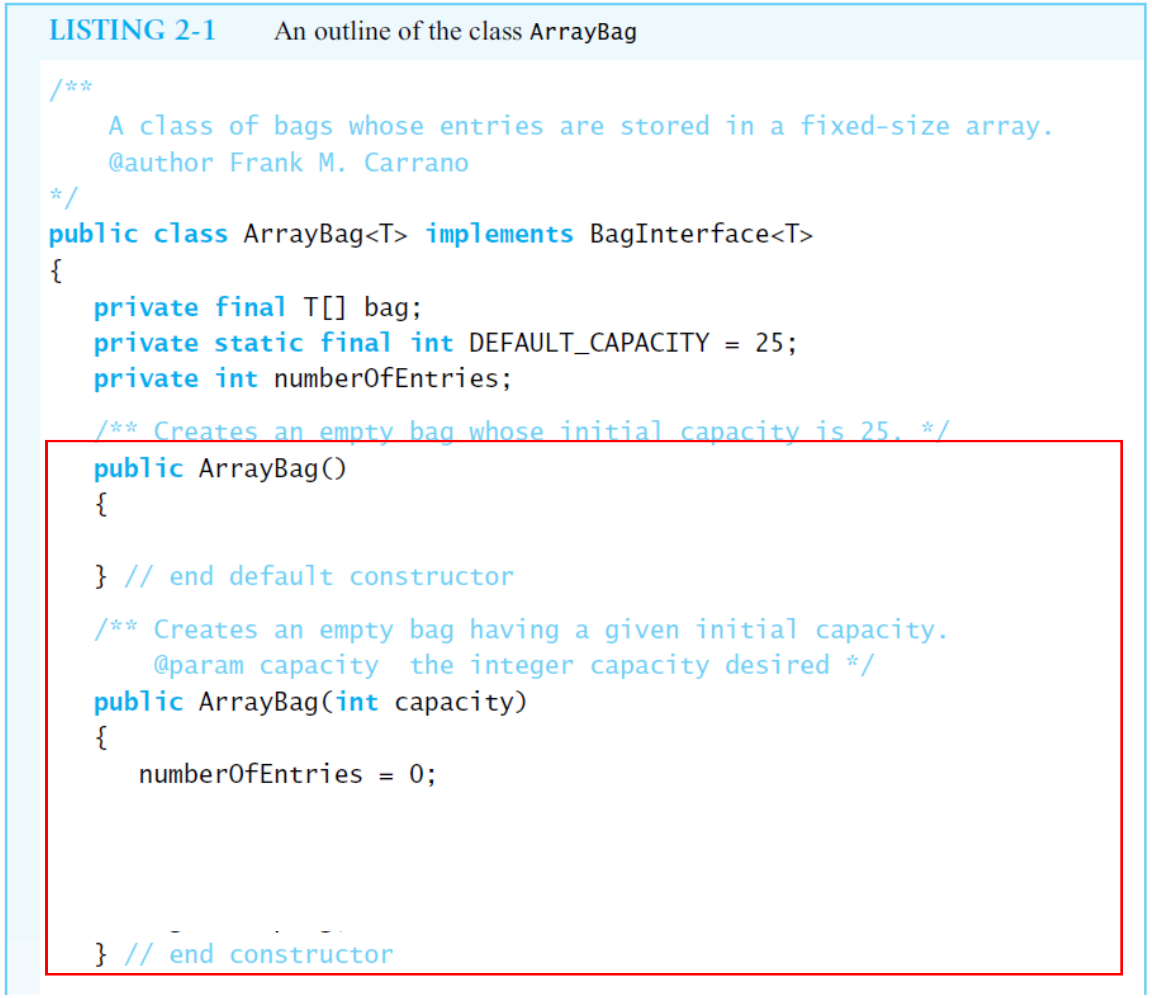
```
/**
   A class of bags whose entries are stored in a fixed-size array.
   @author Frank M. Carrano
*/
public class ArrayBag<T> implements BagInterface<T>
{
   private final T[] bag;
   private static final int DEFAULT_CAPACITY = 25;
   private int numberOfEntries;

   /** Creates an empty bag whose initial capacity is 25. */
   public ArrayBag()
   {

   } // end default constructor

   /** Creates an empty bag having a given initial capacity.
       @param capacity  the integer capacity desired */
   public ArrayBag(int capacity)
   {
      numberOfEntries = 0;



   } // end constructor
```

- Option 1: `bag = new T[capacity]; // SYNTAX ERROR`
- Option 2: `bag = new Object[capacity]; // SYNTAX ERROR: incompatible types`

You cannot assign an array of type Object[] to an array of type T[].

Bag Implementations Th

- **Constructors**
  - Initialize the field *numberOfEntries* to 0.
  - Create the array bag.
    - To create the array, the constructor must specify the array's length, which is the bag's capacity.



LISTING 2-1 An outline of the class ArrayBag

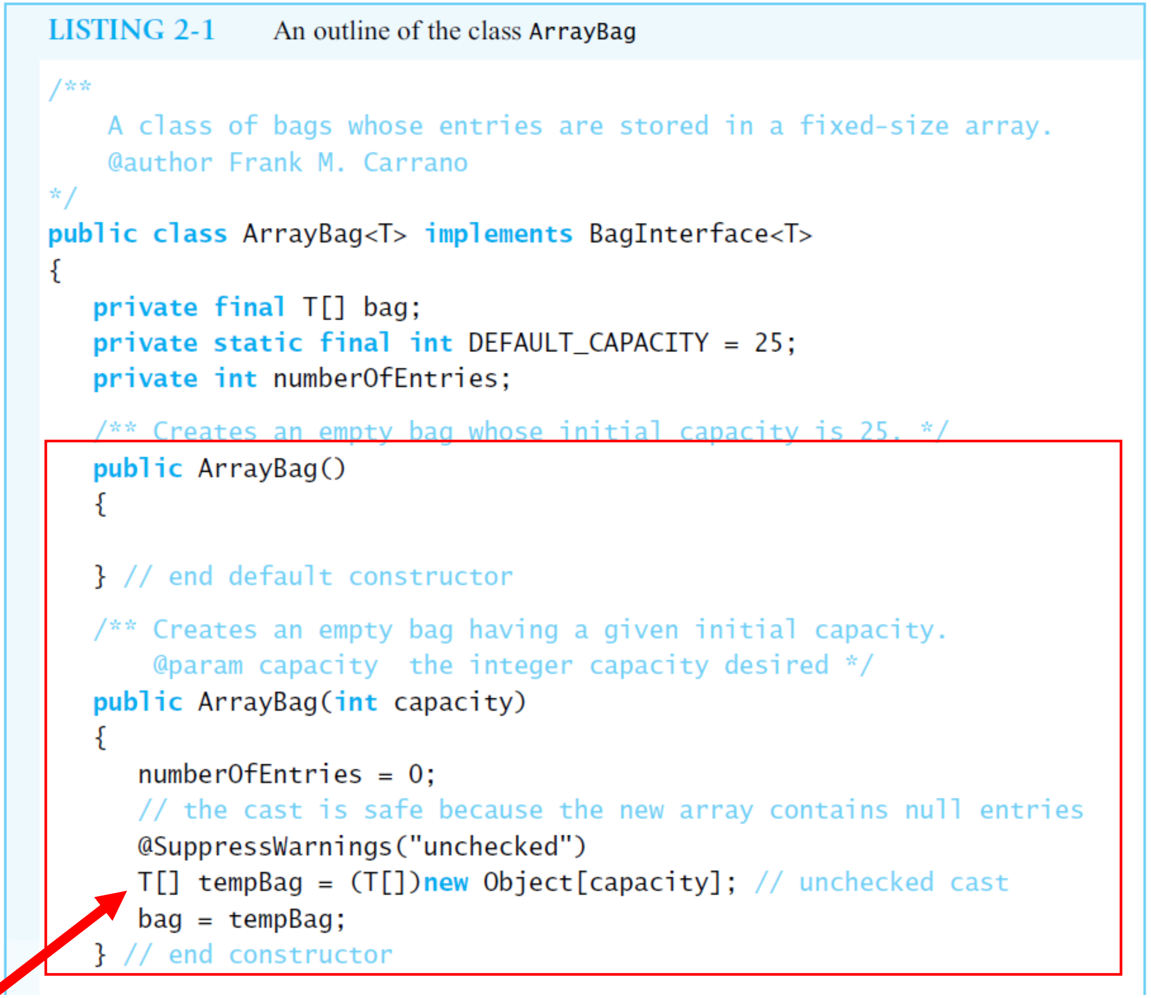
```
/**
    A class of bags whose entries are stored in a fixed-size array.
    @author Frank M. Carrano
*/
public class ArrayBag<T> implements BagInterface<T>
{
   private final T[] bag;
   private static final int DEFAULT_CAPACITY = 25;
   private int numberOfEntries;

   /** Creates an empty bag whose initial capacity is 25. */
   public ArrayBag()
   {

   } // end default constructor

   /** Creates an empty bag having a given initial capacity.
       @param capacity  the integer capacity desired */
   public ArrayBag(int capacity)
   {
      numberOfEntries = 0;




   } // end constructor
```

- Option 1: `bag = new T[capacity]; // SYNTAX ERROR` ✗
- Option 2: `bag = new Object[capacity]; // SYNTAX ERROR: incompatible types` ✗
- Option 3: `bag = (T[])new Object[capacity];` // warning: ArrayBag.java uses unchecked or unsafe operations. ✓✗

You can instruct the compiler to ignore the warning by writing the annotation @SuppressWarnings("unchecked") before the offending statement

Bag Implementations Th

- **Constructors**
  - Initialize the field *numberOfEntries* to 0.
  - Create the array bag.
    - To create the array, the constructor must specify the array's length, which is the bag's capacity.



LISTING 2-1 An outline of the class ArrayBag

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

   /** Creates an empty bag whose initial capacity is 25. */
   public ArrayBag()
   {

   } // end default constructor

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

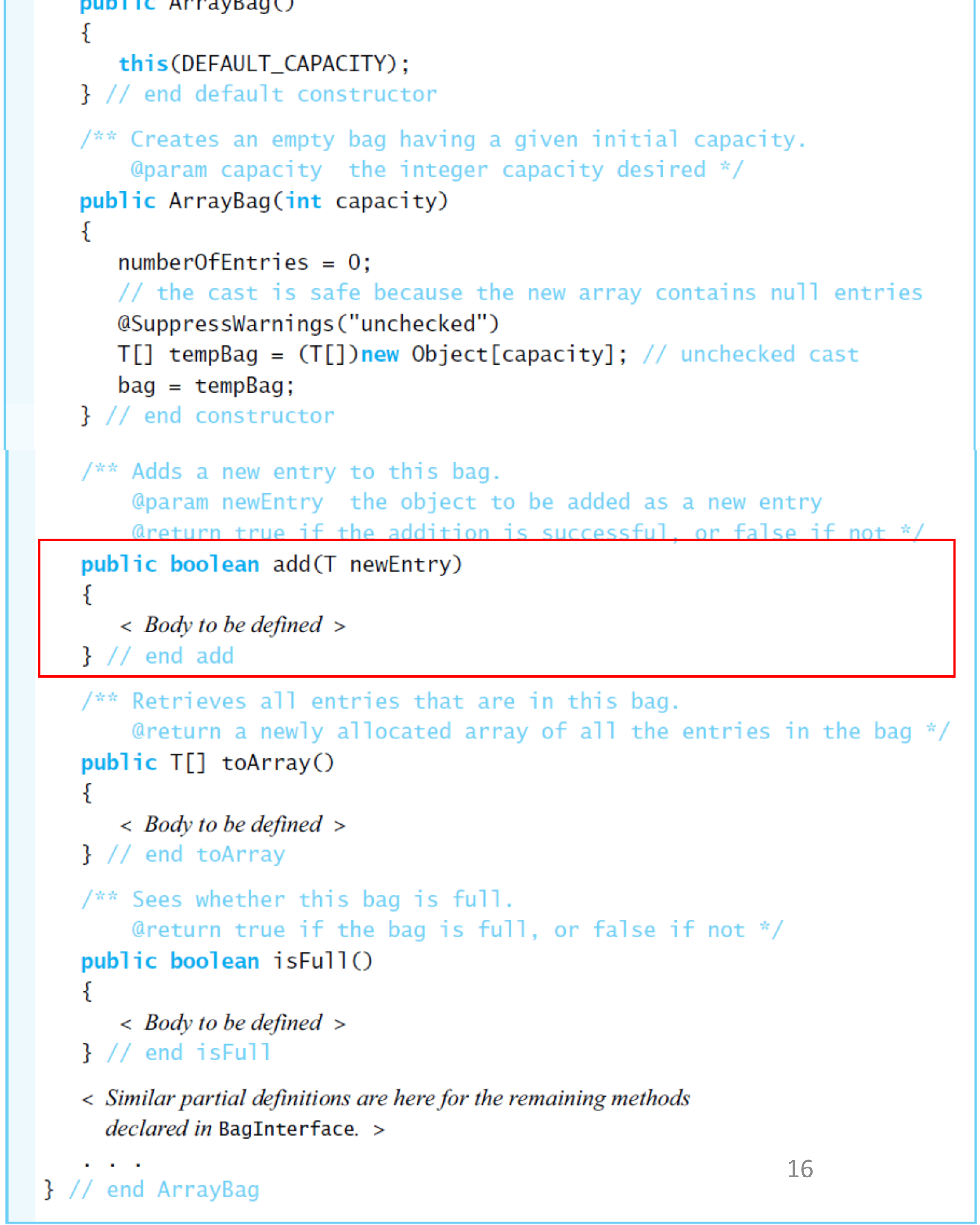
- Option 1: `bag = new T[capacity]; // SYNTAX ERROR` ✗
- Option 2: `bag = new Object[capacity]; // SYNTAX ERROR: incompatible types` ✗
- Option 3: `bag = (T[])new Object[capacity];` // warning: ArrayBag.java uses unchecked or unsafe operations. ✓✗
- Option 4:

```
@SuppressWarnings("unchecked")
T[] tempBag = (T[])new Object[capacity];
bag = tempBag;
```

// This instruction to the compiler can only precede a method definition or a variable declaration.

Bag Implementations Th

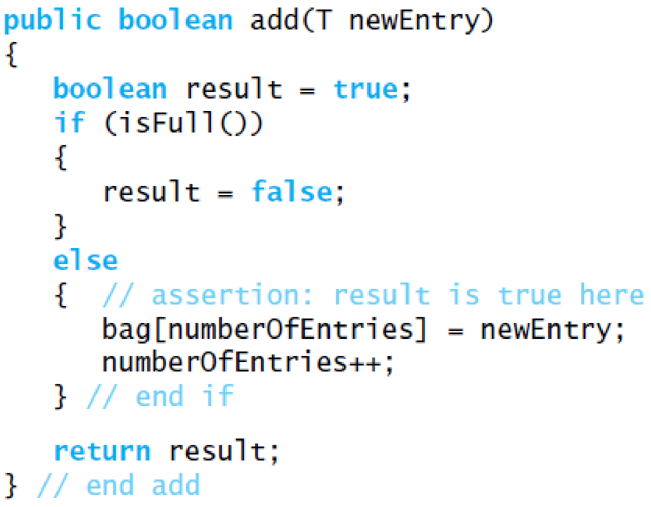
- The method **add**
  - If the bag is full, we cannot add anything to it. In that case, the method **add** should return false
  - Otherwise, we simply add newEntry immediately after the last entry in the array bag



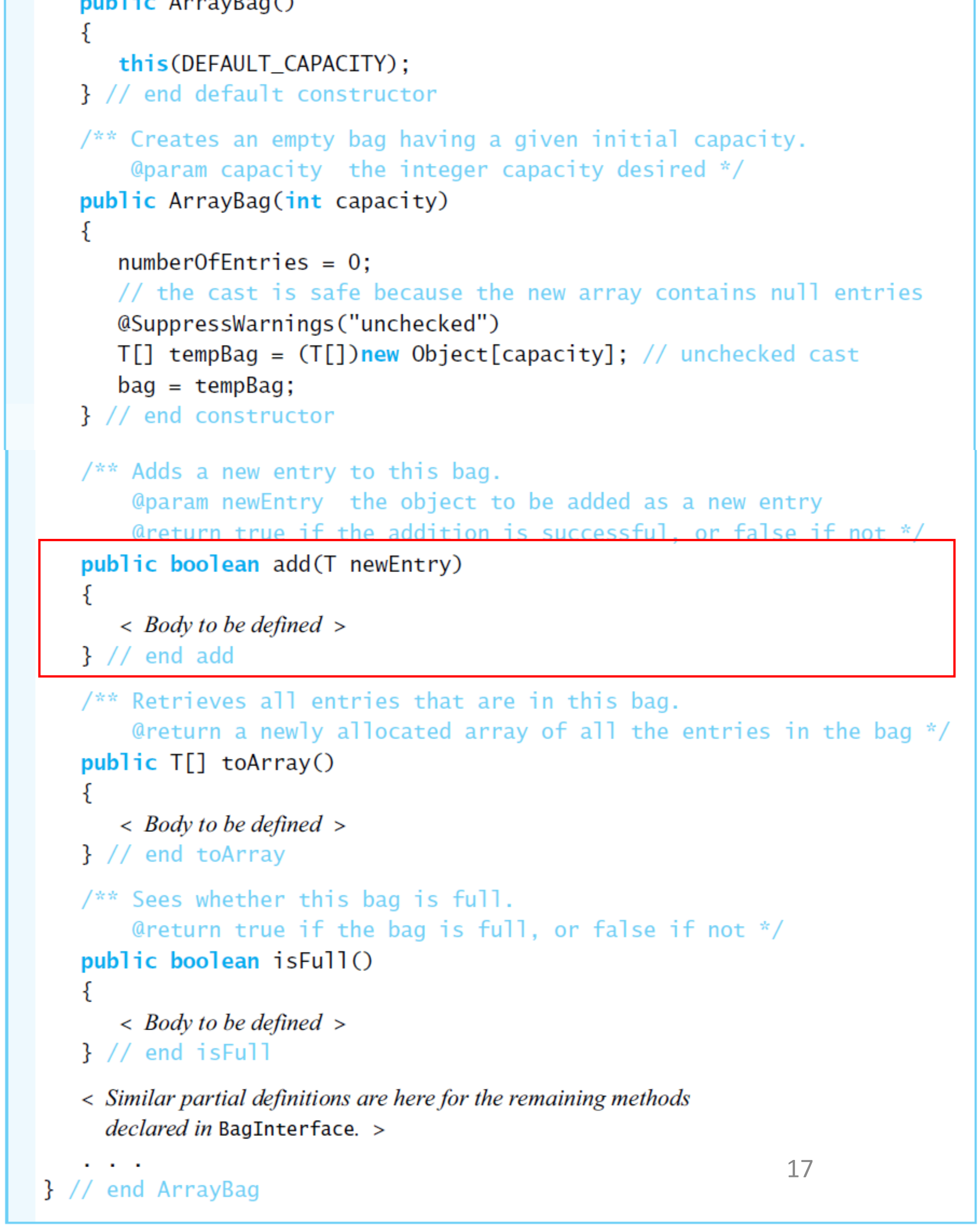
```
   public ArrayBag()
   {
      this(DEFAULT_CAPACITY);
   } // end default constructor

   /** Creates an empty bag having a given initial capacity.
       @param capacity  the integer capacity desired */
   public ArrayBag(int capacity)
   {
      numberOfEntries = 0;
      // the cast is safe because the new array contains null entries
      @SuppressWarnings("unchecked")
      T[] tempBag = (T[])new Object[capacity]; // unchecked cast
      bag = tempBag;
   } // 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.
       @return a newly allocated array of all the entries in the bag */
   public T[] toArray()
   {
      < Body to be defined >
   } // end toArray

   /** Sees whether this bag is full.
       @return true if the bag is full, or false if not */
   public boolean isFull()
   {
      < Body to be defined >
   } // end isFull

   < Similar partial definitions are here for the remaining methods
     declared in BagInterface. >
   . . .
} // end ArrayBag
```

Bag Implementations Th

- The method **add**
  - If the bag is full, we cannot add anything to it. In that case, the method add should return false
  - Otherwise, we simply add newEntry immediately after the last entry in the array bag



```
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
```

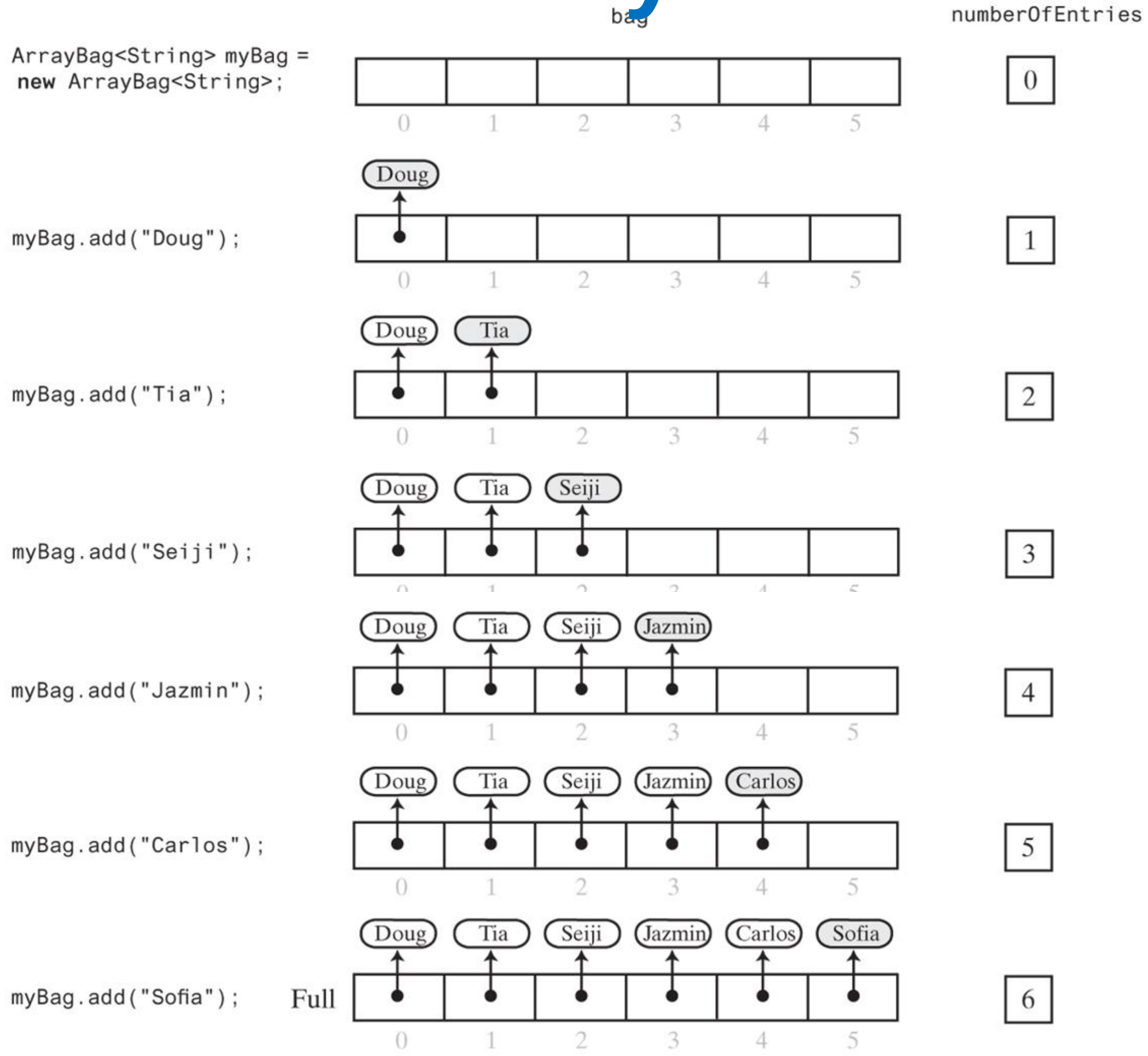


```
   public ArrayBag()
   {
      this(DEFAULT_CAPACITY);
   } // end default constructor

   /** Creates an empty bag having a given initial capacity.
       @param capacity  the integer capacity desired */
   public ArrayBag(int capacity)
   {
      numberOfEntries = 0;
      // the cast is safe because the new array contains null entries
      @SuppressWarnings("unchecked")
      T[] tempBag = (T[])new Object[capacity]; // unchecked cast
      bag = tempBag;
   } // 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.
       @return a newly allocated array of all the entries in the bag */
   public T[] toArray()
   {
      < Body to be defined >
   } // end toArray

   /** Sees whether this bag is full.
       @return true if the bag is full, or false if not */
   public boolean isFull()
   {
      < Body to be defined >
   } // end isFull

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

   . . .
} // end ArrayBag
```

Bag Implementations That Use Arrays

- The method **add**
  - If the bag is full, we cannot add anything to it. In that case, the method add should return false
  - Otherwise, we simply add newEntry immediately after the last entry in the array bag

```
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
```

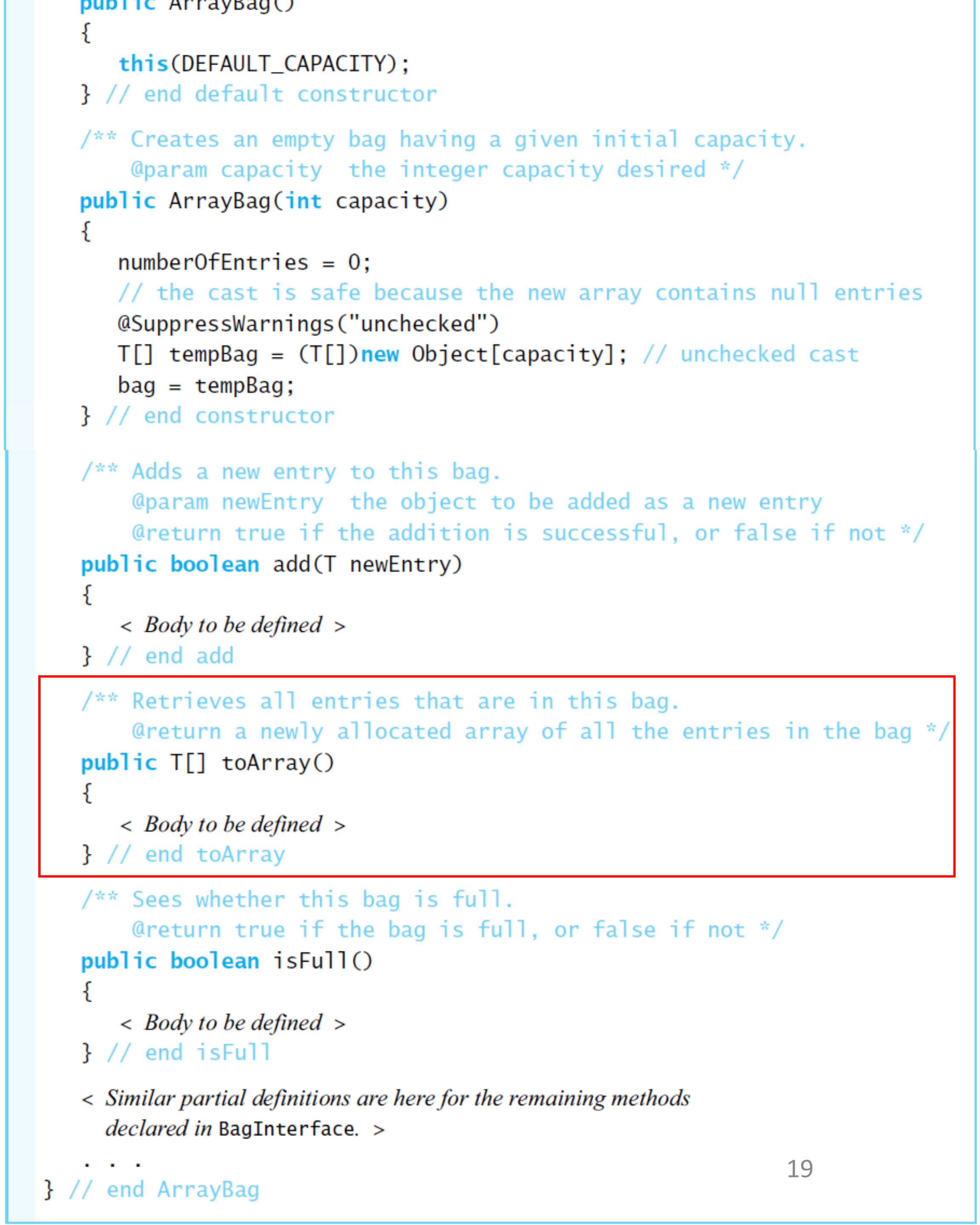


```
ArrayBag<String> myBag =
 new ArrayBag<String>;
myBag.add("Doug");
myBag.add("Tia");
myBag.add("Seiji");
myBag.add("Jazmin");
myBag.add("Carlos");
myBag.add("Sofia");
```

© 2019 Pearson Education, Inc.

Bag Implementations Th

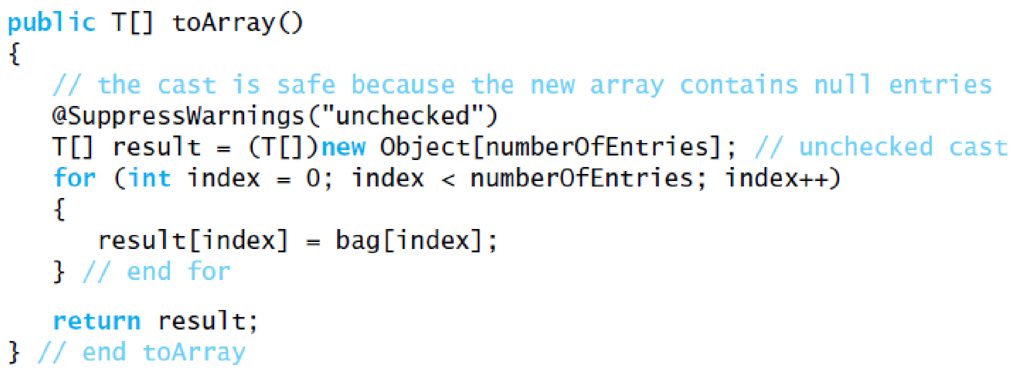
- The method **toArray**
  - <u>Retrieves</u> the entries that are in a bag
  - <u>Returns</u> them to the client within a newly allocated array.



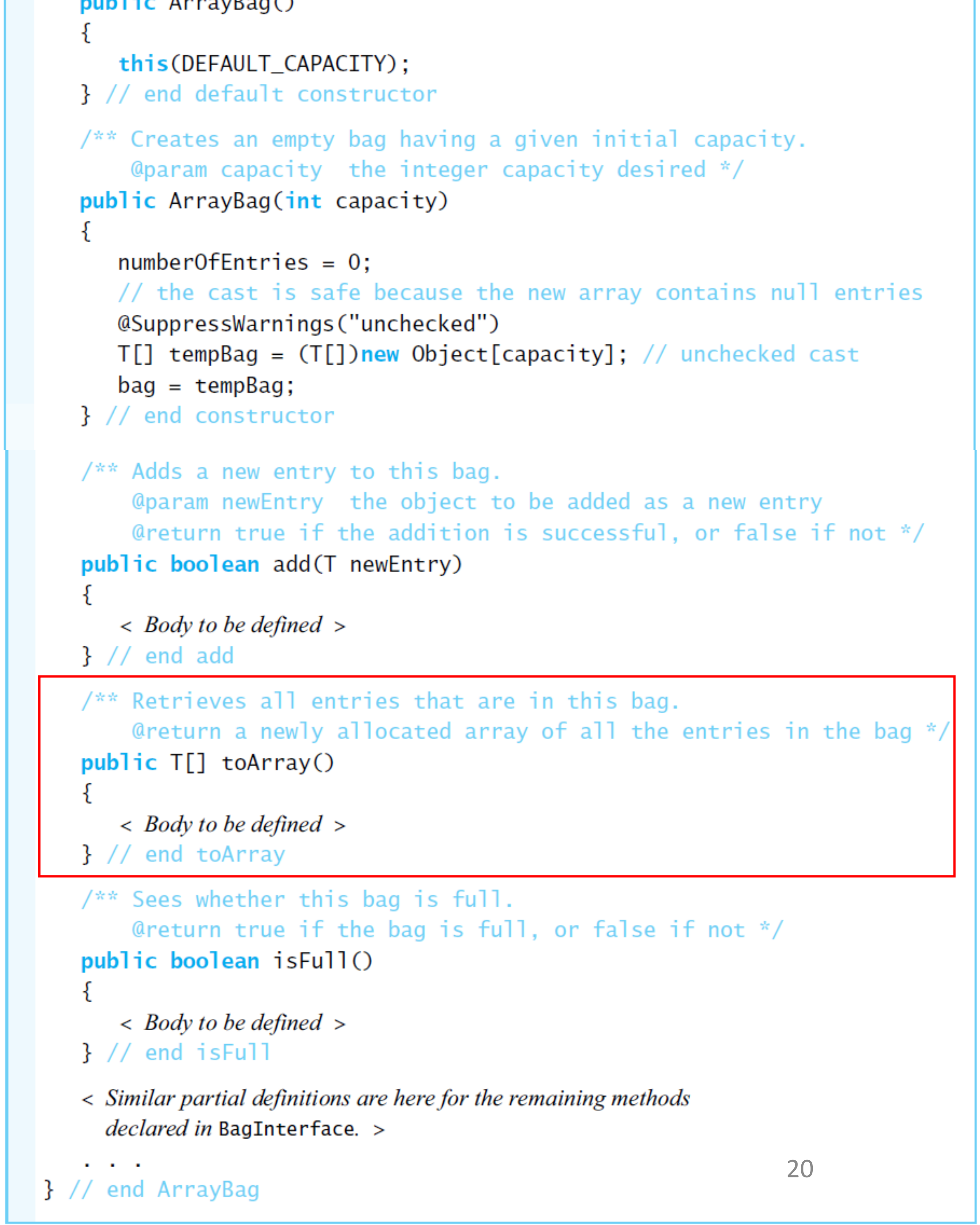
```
   public ArrayBag()
   {
      this(DEFAULT_CAPACITY);
   } // end default constructor

   /** Creates an empty bag having a given initial capacity.
       @param capacity  the integer capacity desired */
   public ArrayBag(int capacity)
   {
      numberOfEntries = 0;
      // the cast is safe because the new array contains null entries
      @SuppressWarnings("unchecked")
      T[] tempBag = (T[])new Object[capacity]; // unchecked cast
      bag = tempBag;
   } // 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.
       @return a newly allocated array of all the entries in the bag */
   public T[] toArray()
   {
      < Body to be defined >
   } // end toArray

   /** Sees whether this bag is full.
       @return true if the bag is full, or false if not */
   public boolean isFull()
   {
      < Body to be defined >
   } // end isFull

   < Similar partial definitions are here for the remaining methods
     declared in BagInterface. >
   . . .
} // end ArrayBag
```

Bag Implementations Th

- The method **toArray**
  - <u>Retrieves</u> the entries that are in a bag
  - <u>Returns</u> them to the client within a newly allocated array.



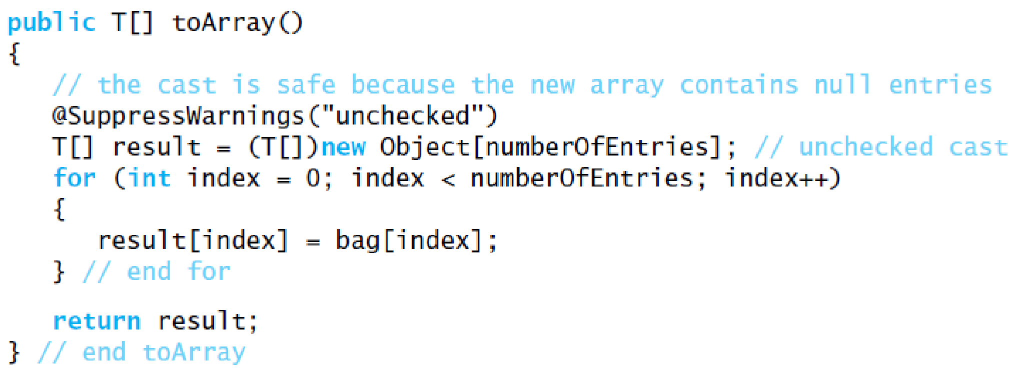
```
public T[] toArray()
{
   // the cast is safe because the new array contains null entries
   @SuppressWarnings("unchecked")
   T[] result = (T[])new Object[numberOfEntries]; // unchecked cast
   for (int index = 0; index < numberOfEntries; index++)
   {
      result[index] = bag[index];
   } // end for

   return result;
} // end toArray
```



```
   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)
   {
      numberOfEntries = 0;
      // the cast is safe because the new array contains null entries
      @SuppressWarnings("unchecked")
      T[] tempBag = (T[])new Object[capacity]; // unchecked cast
      bag = tempBag;
   } // 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.
       @return a newly allocated array of all the entries in the bag */
   public T[] toArray()
   {
      < Body to be defined >
   } // end toArray

   /** Sees whether this bag is full.
       @return true if the bag is full, or false if not */
   public boolean isFull()
   {
      < Body to be defined >
   } // end isFull

   < Similar partial definitions are here for the remaining methods
     declared in BagInterface. >
   . . .
} // end ArrayBag
```

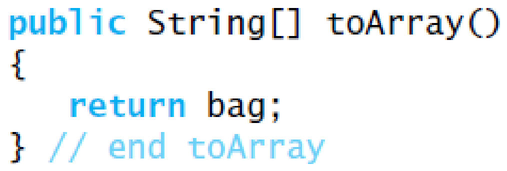
Bag Implementations Th

- The method **toArray**
  - Retrieves the entries that are in a bag
  - Returns them to the client within a newly allocated array.

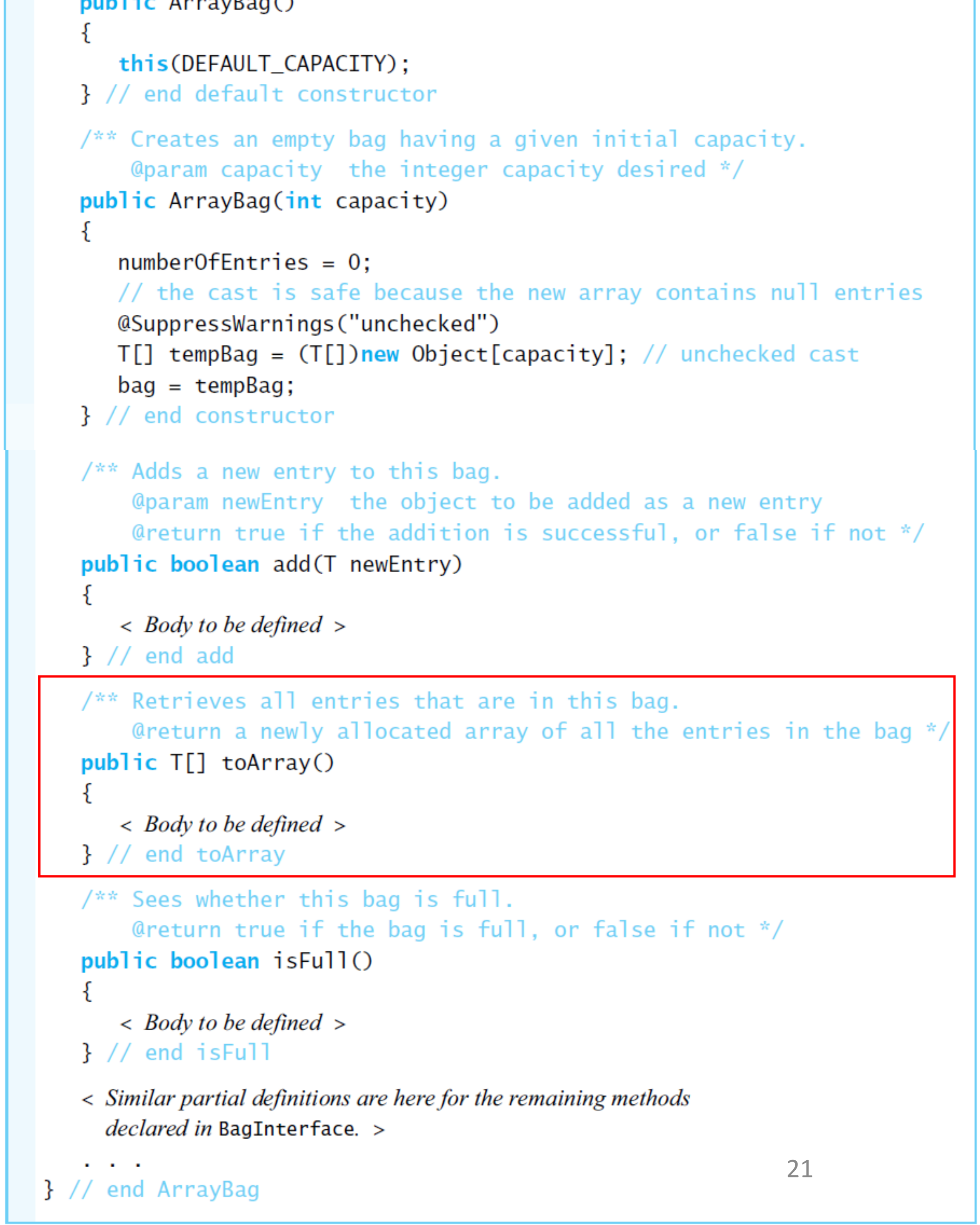


```
public T[] toArray()
{
   // the cast is safe because the new array contains null entries
   @SuppressWarnings("unchecked")
   T[] result = (T[])new Object[numberOfEntries]; // unchecked cast
   for (int index = 0; index < numberOfEntries; index++)
   {
      result[index] = bag[index];
   } // end for

   return result;
} // end toArray
```

- **Question: Can we use the following?**



```
public String[] toArray()
{
    return bag;
} // end toArray
```



```
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)
{
   numberOfEntries = 0;
   // the cast is safe because the new array contains null entries
   @SuppressWarnings("unchecked")
   T[] tempBag = (T[])new Object[capacity]; // unchecked cast
   bag = tempBag;
} // 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.
    @return a newly allocated array of all the entries in the bag */
public T[] toArray()
{
   < Body to be defined >
} // end toArray

/** Sees whether this bag is full.
    @return true if the bag is full, or false if not */
public boolean isFull()
{
   < Body to be defined >
} // end isFull

< Similar partial definitions are here for the remaining methods
  declared in BagInterface. >
   . . .
} // end ArrayBag
```

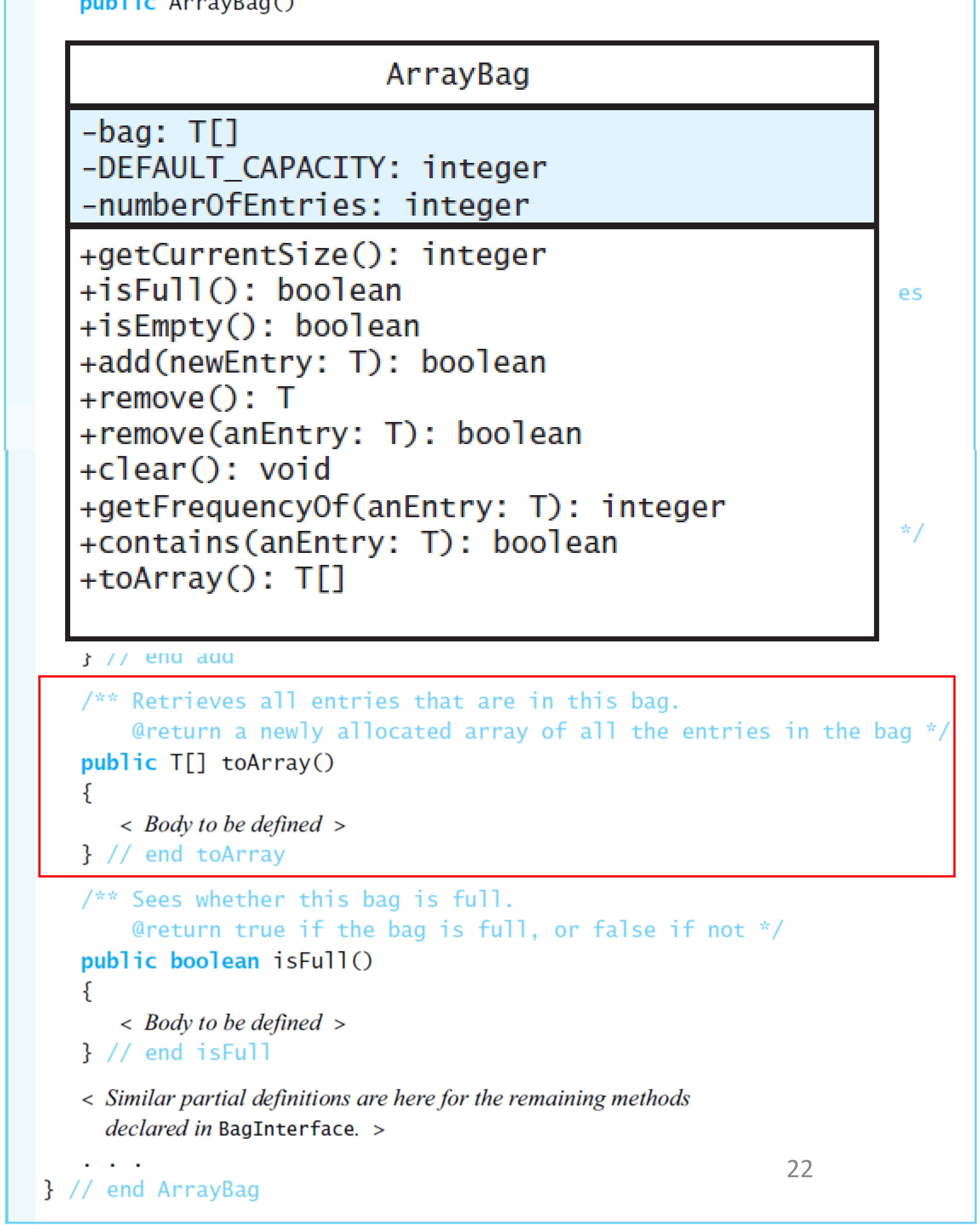
Bag Implementations Th

- The method **toArray**
  - Retrieves the entries that are in a bag
  - Returns them to the client within a newly allocated array.

```
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
```

- **Question: Can we use the following?**

```
public String[] toArray()
{
    return bag;
} // end toArray
```



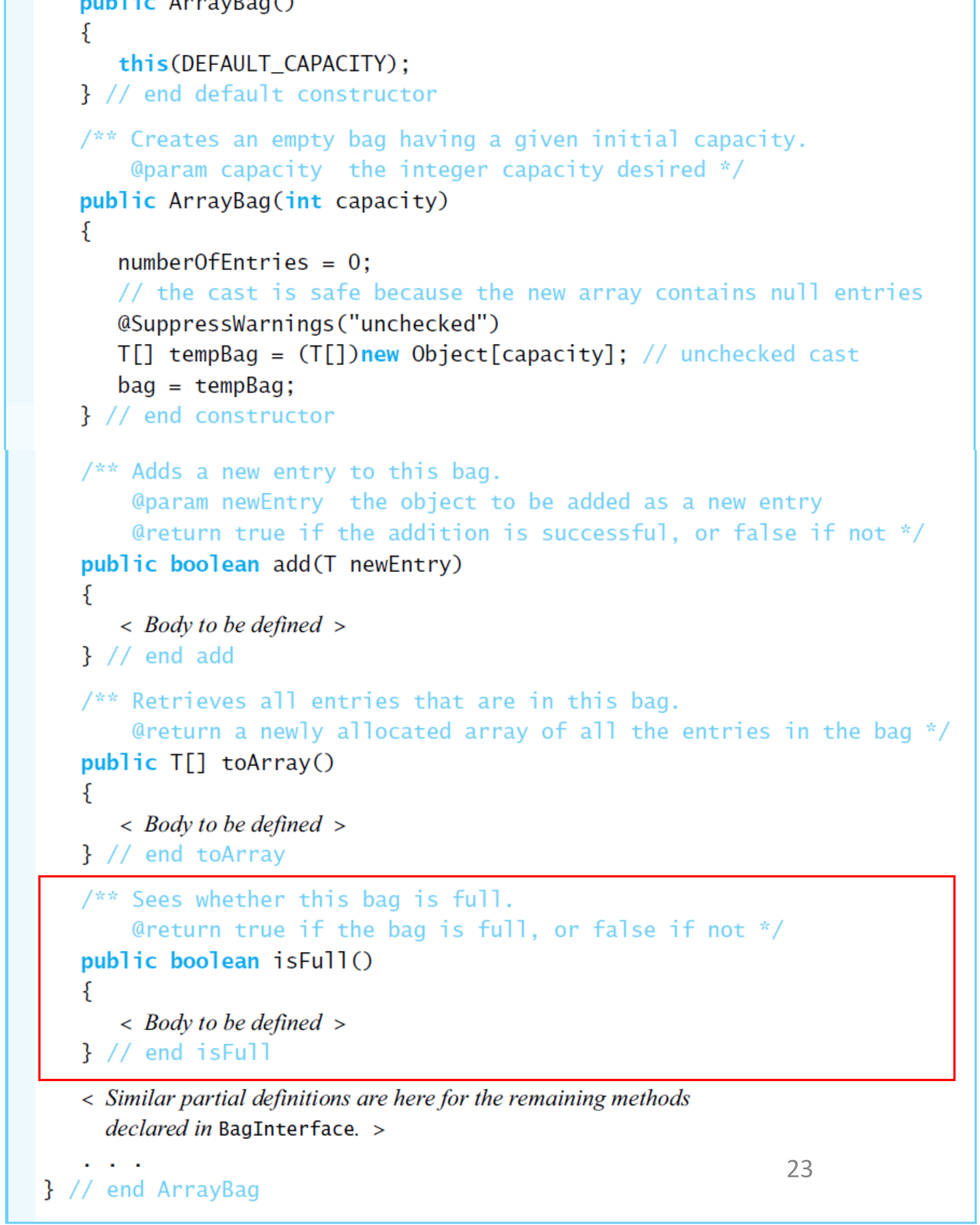
```
public ArrayBag()
```

ArrayBag
-bag: T[] -DEFAULT_CAPACITY: integer -numberOfEntries: integer
+getCurrentSize(): integer +isFull(): boolean +isEmpty(): boolean +add(newEntry: T): boolean +remove(): T +remove(anEntry: T): boolean +clear(): void +getFrequencyOf(anEntry: T): integer +contains(anEntry: T): boolean +toArray(): T[]

```
                                                                es

                                                                */
} // end add

/** 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

/** Sees whether this bag is full.
    @return true if the bag is full, or false if not */
public boolean isFull()
{
   < Body to be defined >
} // end isFull

< Similar partial definitions are here for the remaining methods
  declared in BagInterface. >
. . .
} // end ArrayBag
```

Bag Implementations Th

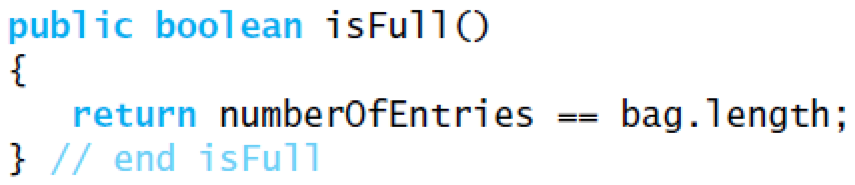
- The method **isFull()**
  - A bag is full when it contains as many objects as the array bag can accommodate.



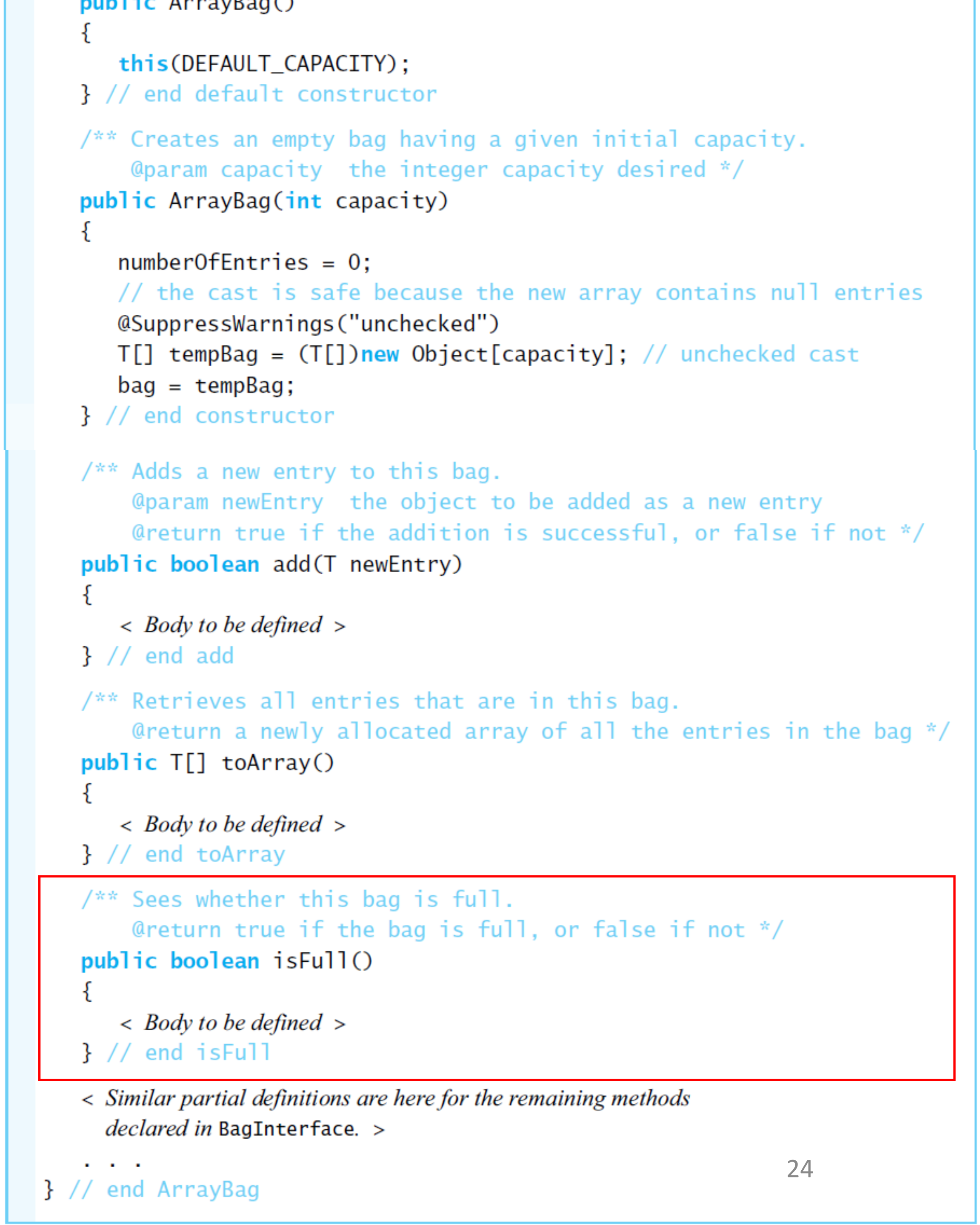
```
public ArrayBag()
{
   this(DEFAULT_CAPACITY);
} // end default constructor

/** Creates an empty bag having a given initial capacity.
    @param capacity  the integer capacity desired */
public ArrayBag(int capacity)
{
   numberOfEntries = 0;
   // the cast is safe because the new array contains null entries
   @SuppressWarnings("unchecked")
   T[] tempBag = (T[])new Object[capacity]; // unchecked cast
   bag = tempBag;
} // 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.
    @return a newly allocated array of all the entries in the bag */
public T[] toArray()
{
   < Body to be defined >
} // end toArray

/** Sees whether this bag is full.
    @return true if the bag is full, or false if not */
public boolean isFull()
{
   < Body to be defined >
} // end isFull

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

. . .
} // end ArrayBag
```

Bag Implementations Th

- The method **isFull()**
  - A bag is full when it contains as many objects as the array bag can accommodate.

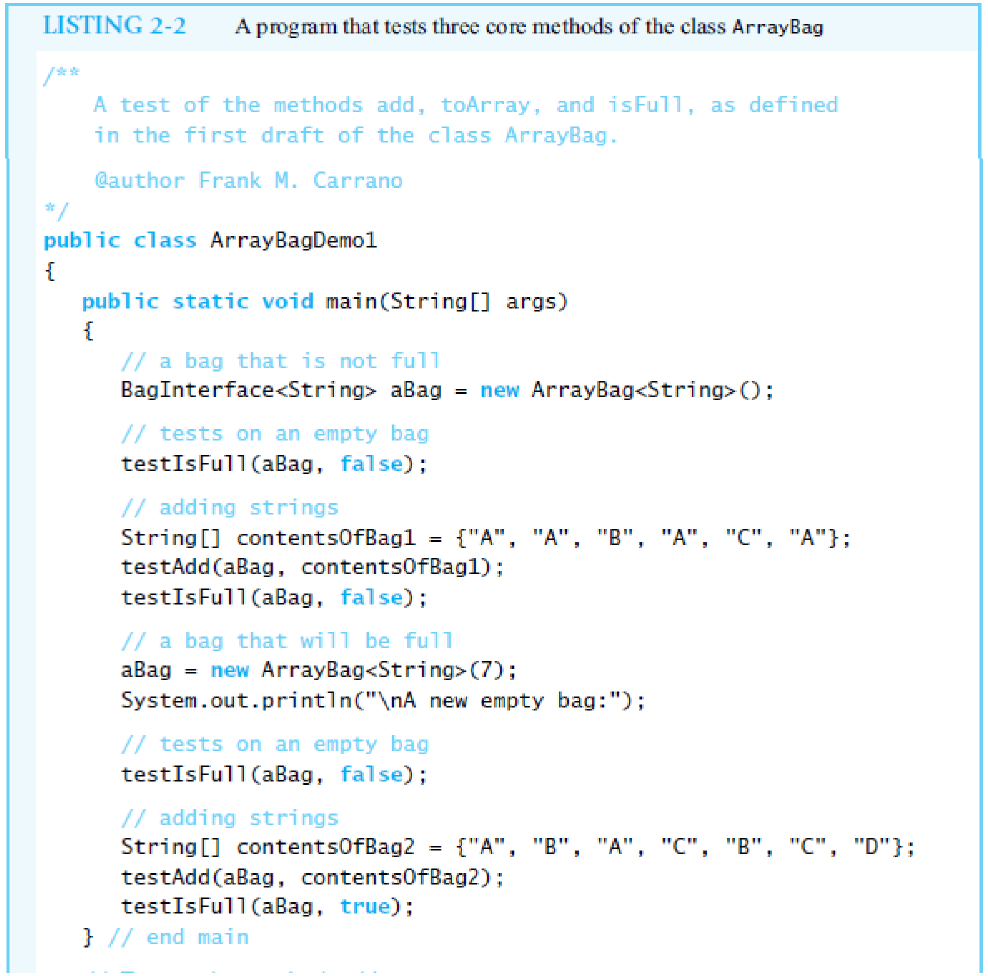


```
public boolean isFull()
{
   return numberOfEntries == bag.length;
} // end isFull
```



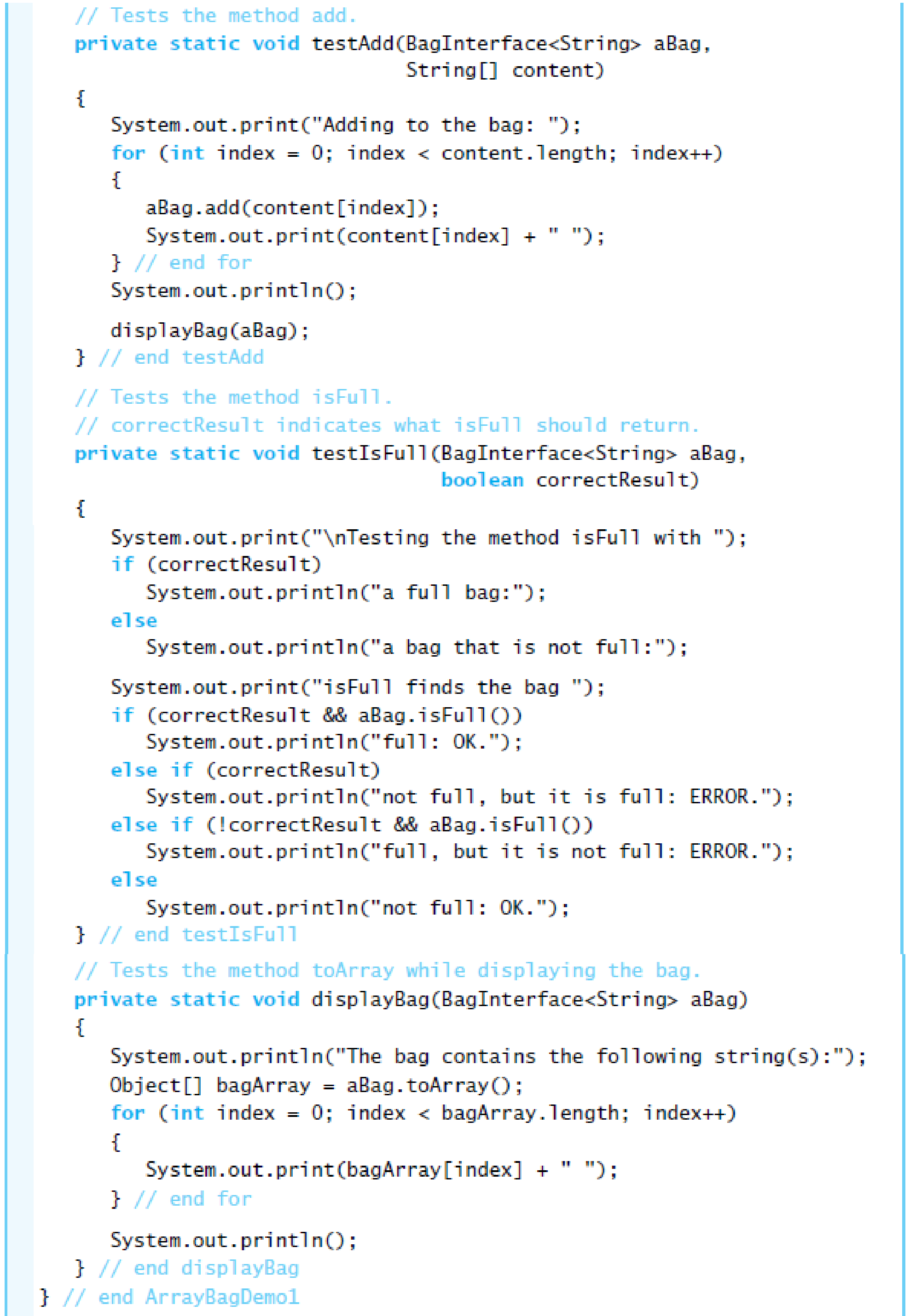
```
   public ArrayBag()
   {
      this(DEFAULT_CAPACITY);
   } // end default constructor

   /** Creates an empty bag having a given initial capacity.
       @param capacity  the integer capacity desired */
   public ArrayBag(int capacity)
   {
      numberOfEntries = 0;
      // the cast is safe because the new array contains null entries
      @SuppressWarnings("unchecked")
      T[] tempBag = (T[])new Object[capacity]; // unchecked cast
      bag = tempBag;
   } // 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.
       @return a newly allocated array of all the entries in the bag */
   public T[] toArray()
   {
      < Body to be defined >
   } // end toArray

   /** Sees whether this bag is full.
       @return true if the bag is full, or false if not */
   public boolean isFull()
   {
      < Body to be defined >
   } // end isFull

   < Similar partial definitions are here for the remaining methods
     declared in BagInterface. >
   . . .
} // end ArrayBag
```

A Test Program



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

```
/**
    A test of the methods add, toArray, and isFull, as defined
    in the first draft of the class ArrayBag.

    @author Frank M. Carrano
*/
public class ArrayBagDemo1
{
   public static void main(String[] args)
   {
      // a bag that is not full
      BagInterface<String> aBag = new ArrayBag<String>();

      // tests on an empty bag
      testIsFull(aBag, false);

      // adding strings
      String[] contentsOfBag1 = {"A", "A", "B", "A", "C", "A"};
      testAdd(aBag, contentsOfBag1);
      testIsFull(aBag, false);

      // a bag that will be full
      aBag = new ArrayBag<String>(7);
      System.out.println("\nA new empty bag:");

      // tests on an empty bag
      testIsFull(aBag, false);

      // adding strings
      String[] contentsOfBag2 = {"A", "B", "A", "C", "B", "C", "D"};
      testAdd(aBag, contentsOfBag2);
      testIsFull(aBag, true);
   } // end main
```



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

      displayBag(aBag);
   } // end testAdd

   // Tests the method isFull.
   // correctResult indicates what isFull should return.
   private static void testIsFull(BagInterface<String> aBag,
                                  boolean correctResult)
   {
      System.out.print("\nTesting the method isFull with ");
      if (correctResult)
         System.out.println("a full bag:");
      else
         System.out.println("a bag that is not full:");

      System.out.print("isFull finds the bag ");
      if (correctResult && aBag.isFull())
         System.out.println("full: OK.");
      else if (correctResult)
         System.out.println("not full, but it is full: ERROR.");
      else if (!correctResult && aBag.isFull())
         System.out.println("full, but it is not full: ERROR.");
      else
         System.out.println("not full: OK.");
   } // end testIsFull

   // 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
```

Making the Implementation Secure

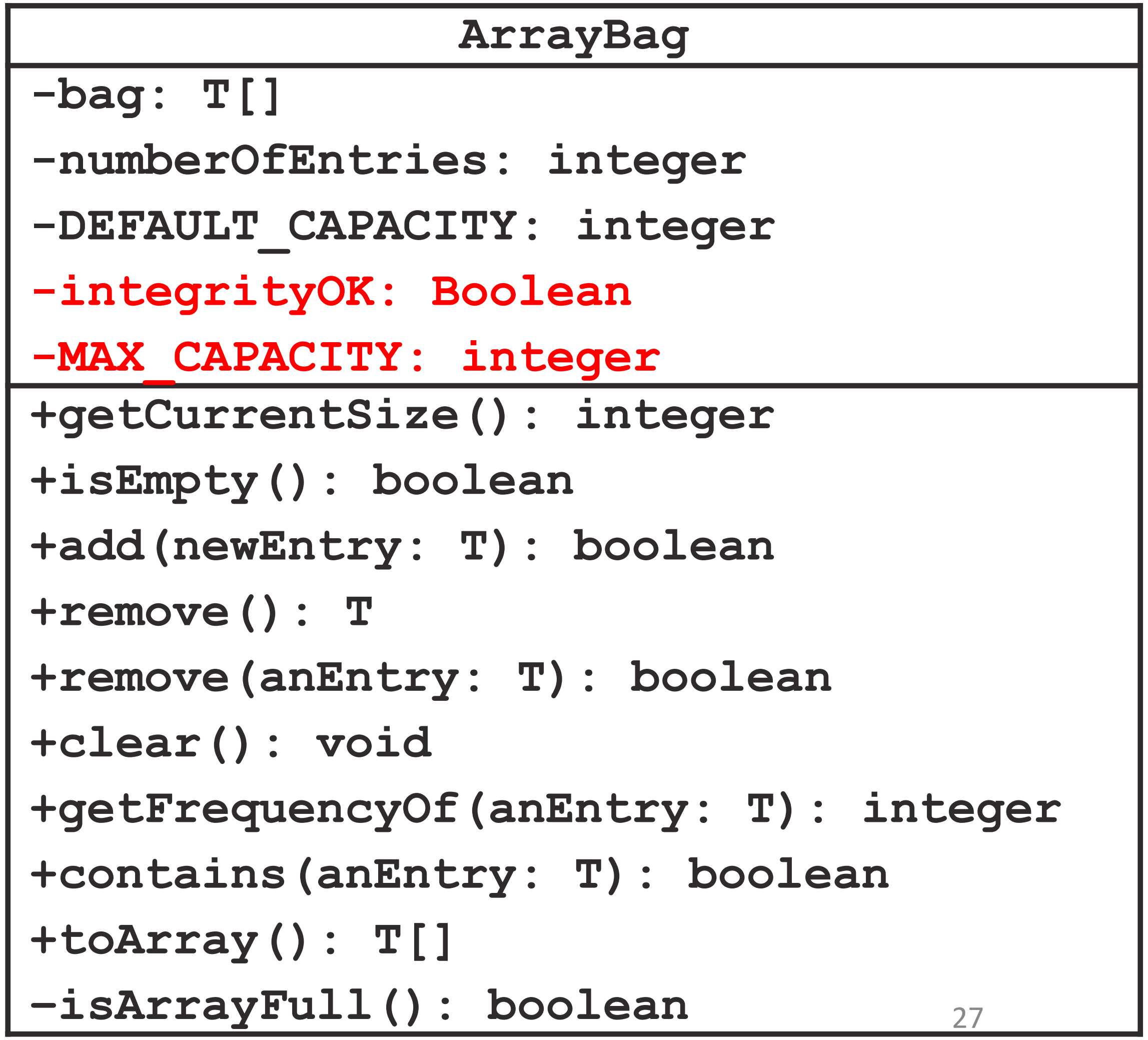
- Practice fail-safe programming by including checks for anticipated errors
- Validate input data and arguments to a method
- Refine incomplete implementation of ArrayBag to make code more secure by adding the following two data fields:

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

Making the Implementation Secure

- **Revised Constructor**

```
 /** Creates an empty bag having a given capacity.
@param desiredCapacity  The integer capacity desired. */
 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;
              integrityOK = true;
       }
       else
           throw new IllegalStateException("Attempt to create a bag whose"
       +  "capacity exceeds allowed maximum.");
 } // end constructor
```

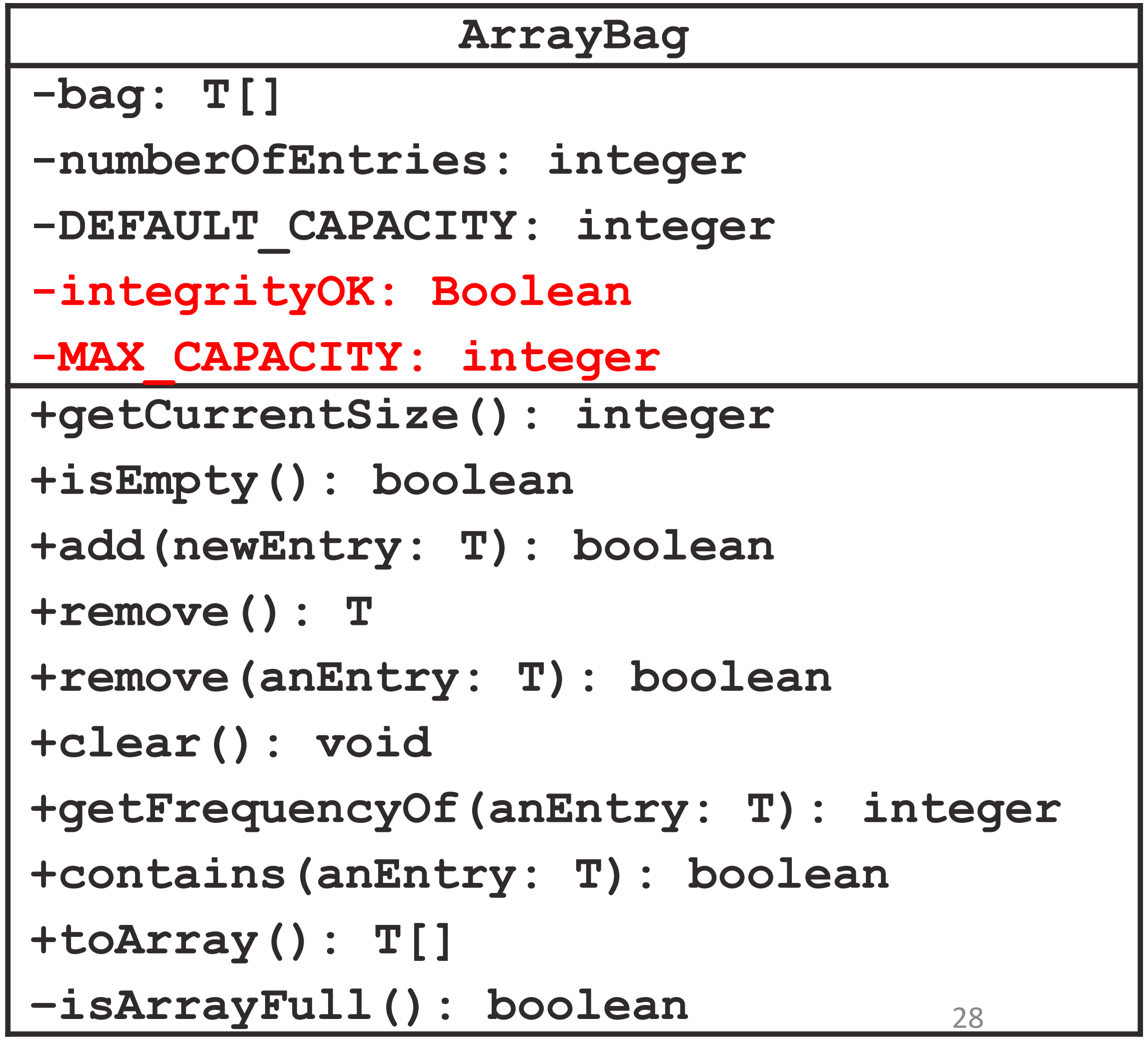


```
ArrayBag
-bag: T[]
-numberOfEntries: integer
-DEFAULT_CAPACITY: integer
-integrityOK: Boolean
-MAX_CAPACITY: integer
+getCurrentSize(): integer
+isEmpty(): boolean
+add(newEntry: T): boolean
+remove(): T
+remove(anEntry: T): boolean
+clear(): void
+getFrequencyOf(anEntry: T): integer
+contains(anEntry: T): boolean
+toArray(): T[]
-isArrayFull(): boolean
```

Making the Implementation Secure

- Method to check initialization

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



ArrayBag

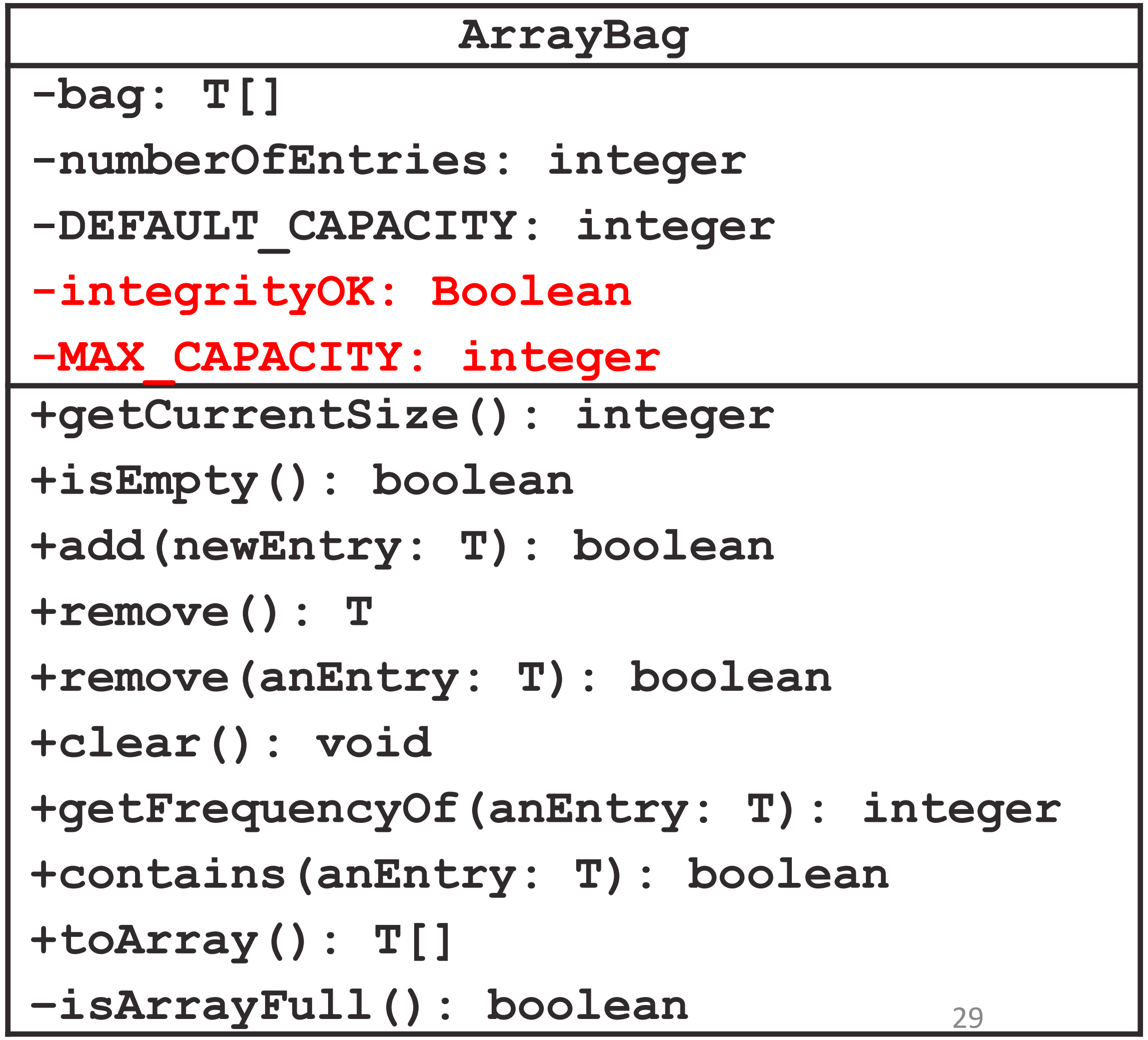
-bag: T[]
-numberOfEntries: integer
-DEFAULT_CAPACITY: integer
-integrityOK: Boolean
-MAX_CAPACITY: integer

+getCurrentSize(): integer
+isEmpty(): boolean
+add(newEntry: T): boolean
+remove(): T
+remove(anEntry: T): boolean
+clear(): void
+getFrequencyOf(anEntry: T): integer
+contains(anEntry: T): boolean
+toArray(): T[]
-isArrayFull(): boolean

Making the Implementation Secure

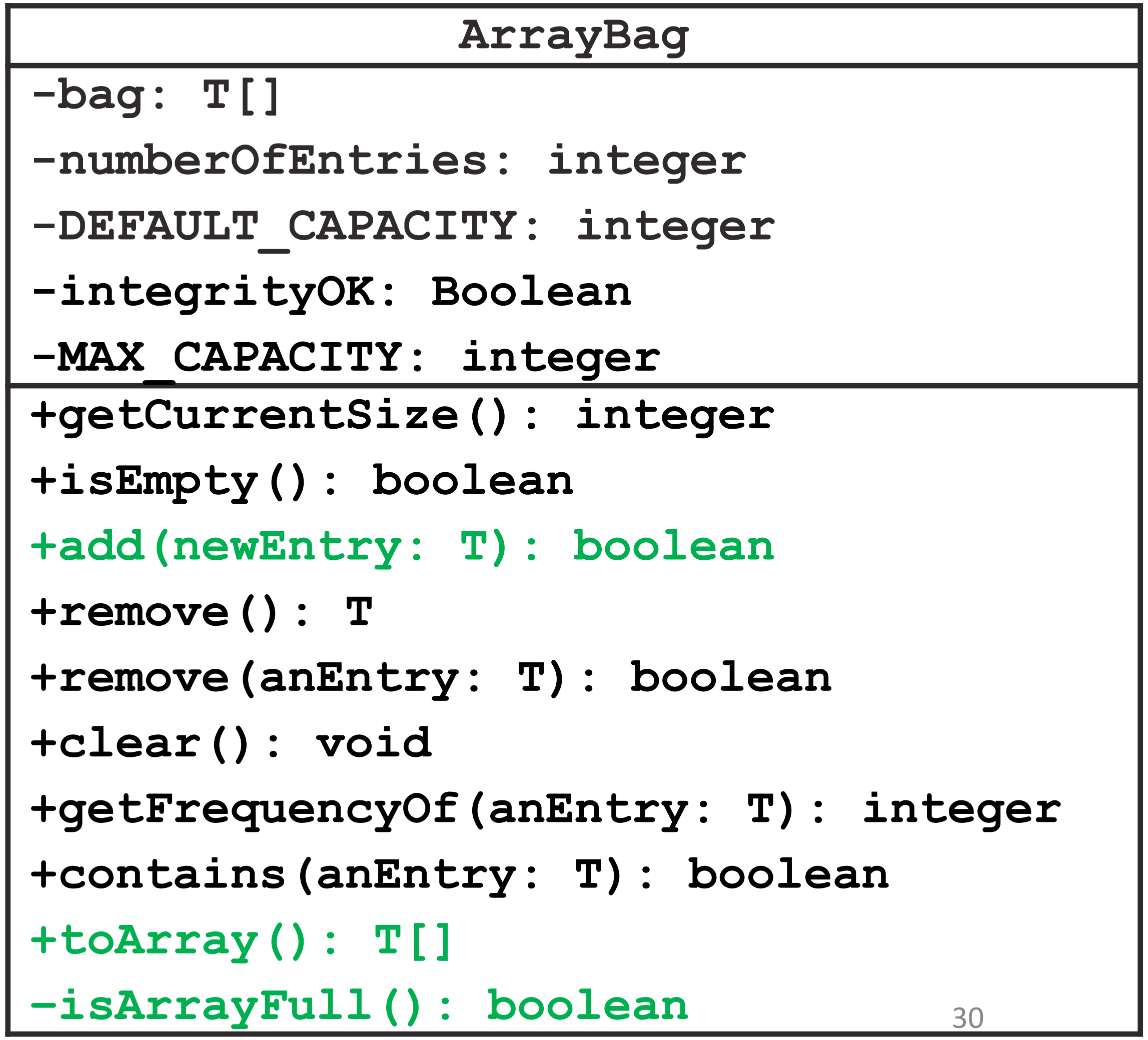
- **Revised method add**

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

       return result;
 } // end add
```



```
ArrayBag
-bag: T[]
-numberOfEntries: integer
-DEFAULT_CAPACITY: integer
-integrityOK: Boolean
-MAX_CAPACITY: integer
+getCurrentSize(): integer
+isEmpty(): boolean
+add(newEntry: T): boolean
+remove(): T
+remove(anEntry: T): boolean
+clear(): void
+getFrequencyOf(anEntry: T): integer
+contains(anEntry: T): boolean
+toArray(): T[]
-isArrayFull(): boolean
```

Implementing More Methods



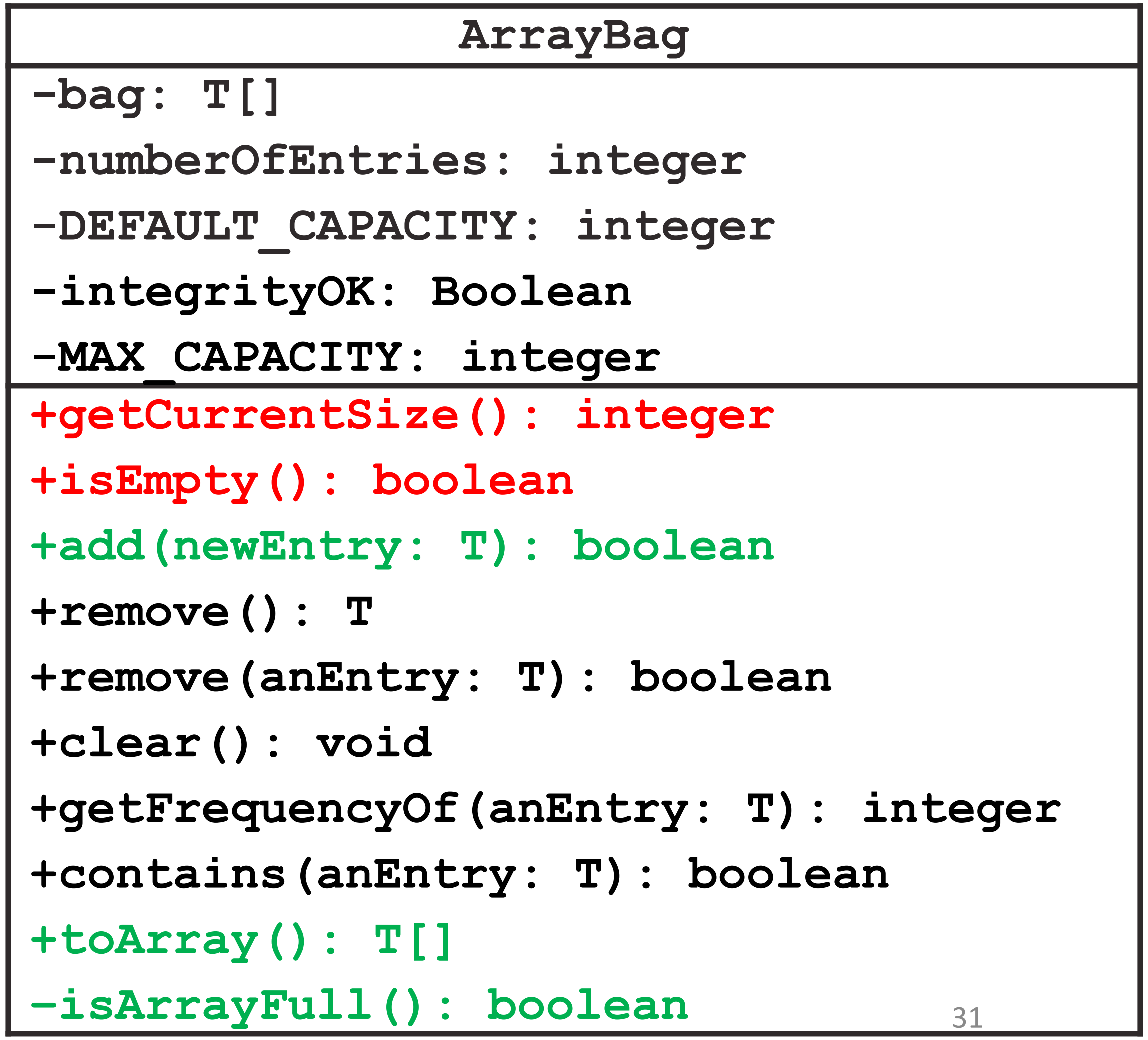
ArrayBag

-bag: T[]
-numberOfEntries: integer
-DEFAULT_CAPACITY: integer
-integrityOK: Boolean
-MAX_CAPACITY: integer

+getCurrentSize(): integer
+isEmpty(): boolean
+add(newEntry: T): boolean
+remove(): T
+remove(anEntry: T): boolean
+clear(): void
+getFrequencyOf(anEntry: T): integer
+contains(anEntry: T): boolean
+toArray(): T[]
-isArrayFull(): boolean

Implementing More Methods

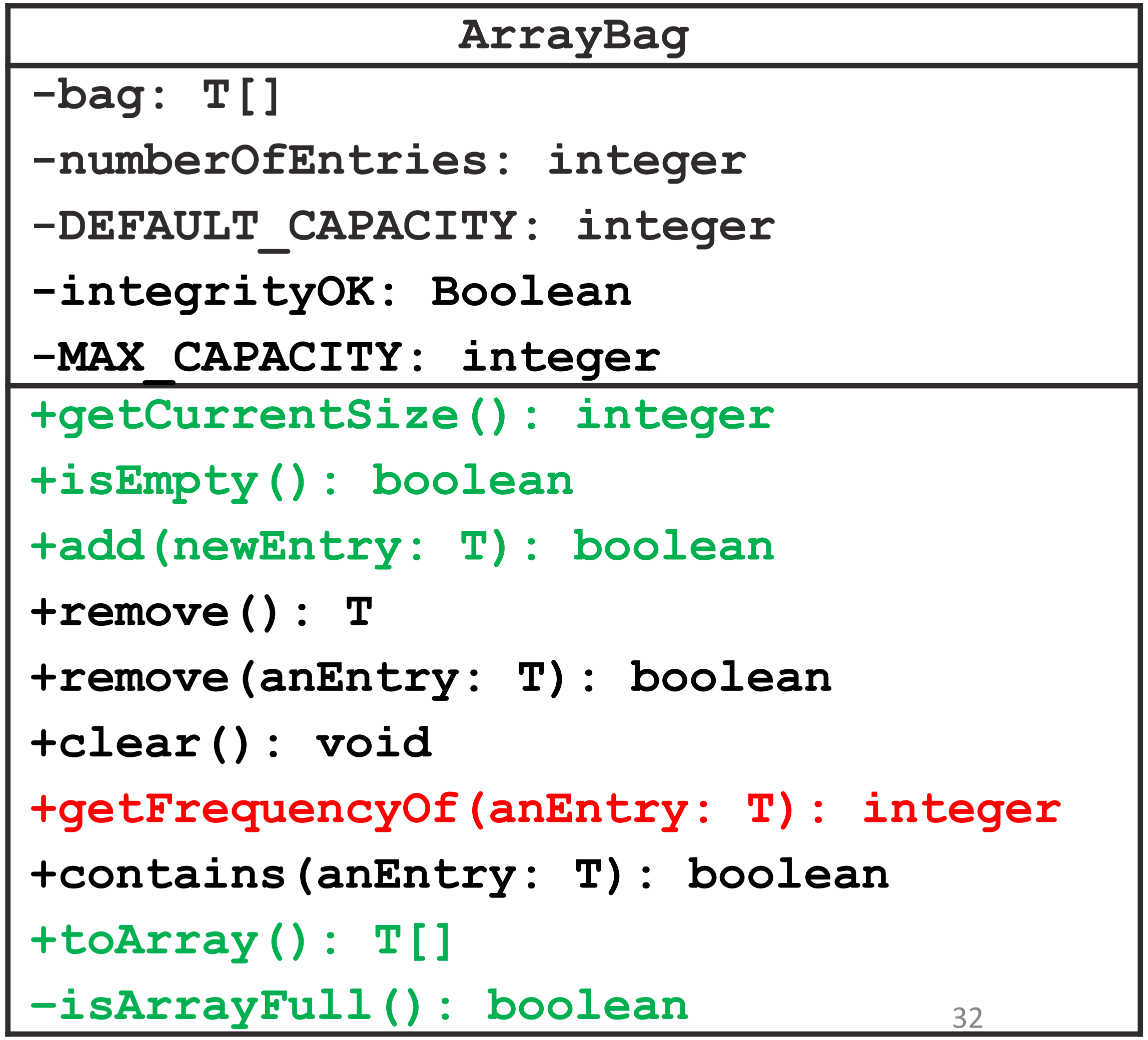
```
 /** Sees whether this bag is empty.
@return  True if this bag is empty, or false if not. */
 public boolean isEmpty()
 {
       return numberOfEntries == 0;
 } // end isEmpty


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



```
ArrayBag
-bag: T[]
-numberOfEntries: integer
-DEFAULT_CAPACITY: integer
-integrityOK: Boolean
-MAX_CAPACITY: integer
+getCurrentSize(): integer
+isEmpty(): boolean
+add(newEntry: T): boolean
+remove(): T
+remove(anEntry: T): boolean
+clear(): void
+getFrequencyOf(anEntry: T): integer
+contains(anEntry: T): boolean
+toArray(): T[]
-isArrayFull(): boolean
```

Implementing More Methods

```
 /** 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 this bag. */
 public int getFrequencyOf(T anEntry)
 {
        checkIntegrity();
        int counter = 0;

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



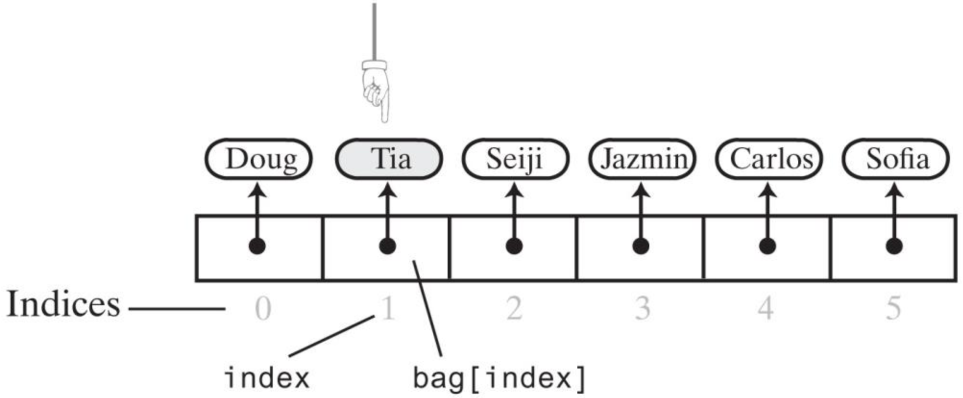
ArrayBag

-bag: T[]
-numberOfEntries: integer
-DEFAULT_CAPACITY: integer
-integrityOK: Boolean
-MAX_CAPACITY: integer

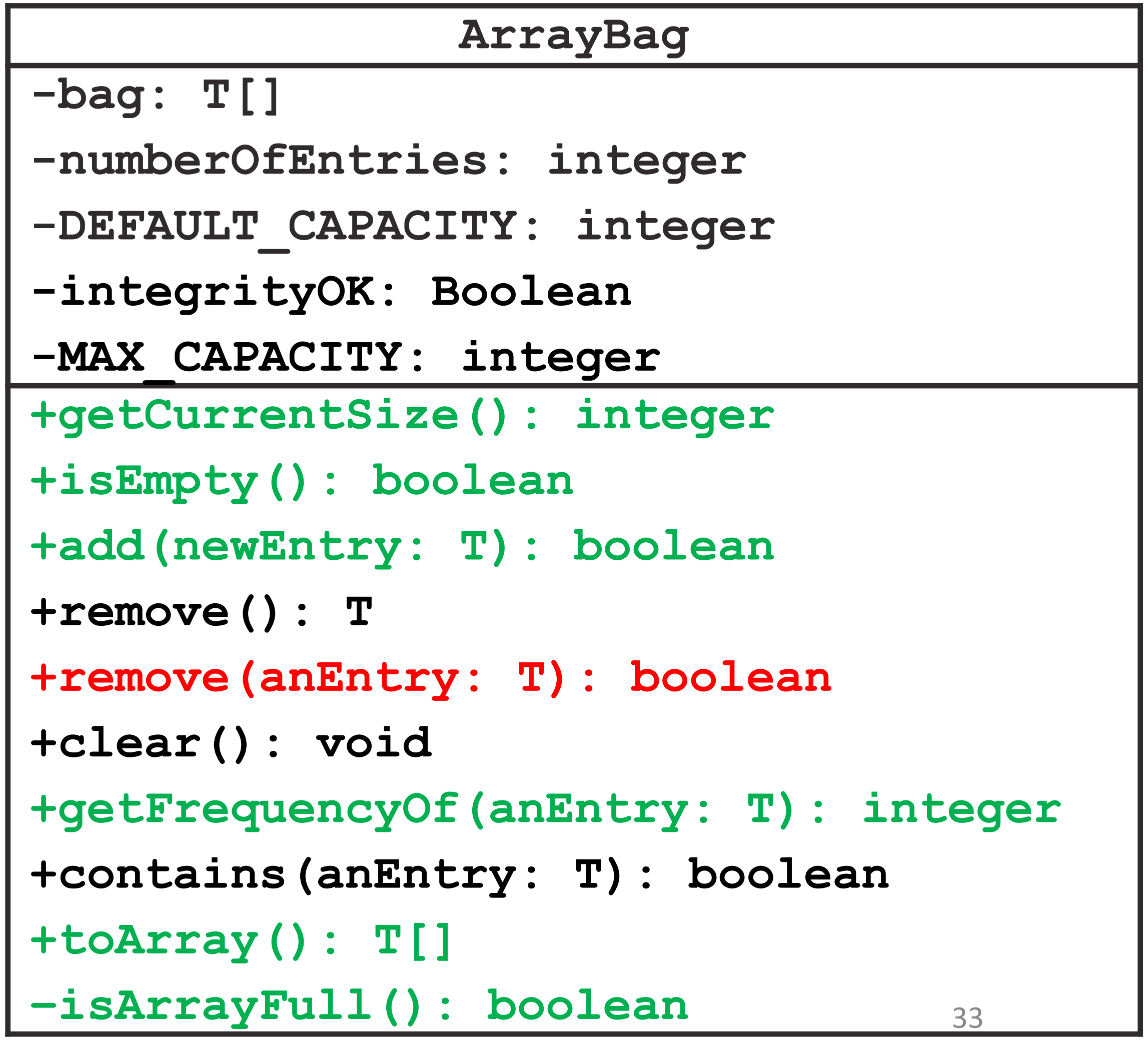
+getCurrentSize(): integer
+isEmpty(): boolean
+add(newEntry: T): boolean
+remove(): T
+remove(anEntry: T): boolean
+clear(): void
+getFrequencyOf(anEntry: T): integer
+contains(anEntry: T): boolean
+toArray(): T[]
-isArrayFull(): boolean

Implementing More Methods

- **Removing a given entry**
  - Search for the entry
  - Remove the entry from the bag



©2019 Pearson Education, Inc.

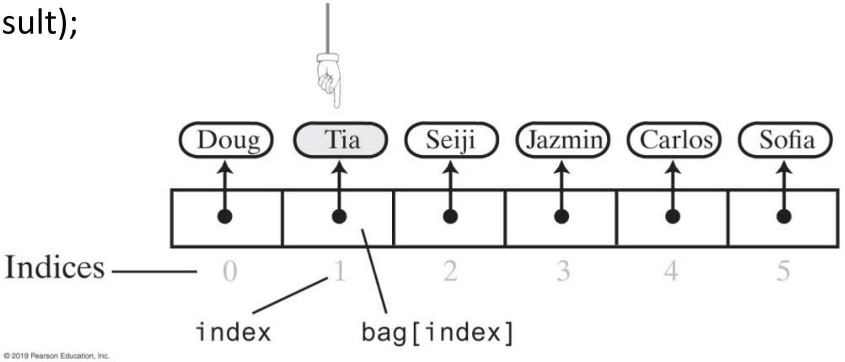


```
ArrayBag
-bag: T[]
-numberOfEntries: integer
-DEFAULT_CAPACITY: integer
-integrityOK: Boolean
-MAX_CAPACITY: integer
+getCurrentSize(): integer
+isEmpty(): boolean
+add(newEntry: T): boolean
+remove(): T
+remove(anEntry: T): boolean
+clear(): void
+getFrequencyOf(anEntry: T): integer
+contains(anEntry: T): boolean
+toArray(): T[]
-isArrayFull(): boolean
```

Implementing More Methods

- **Removing a given entry**
  - Search for the entry
  - Remove the entry from the bag

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



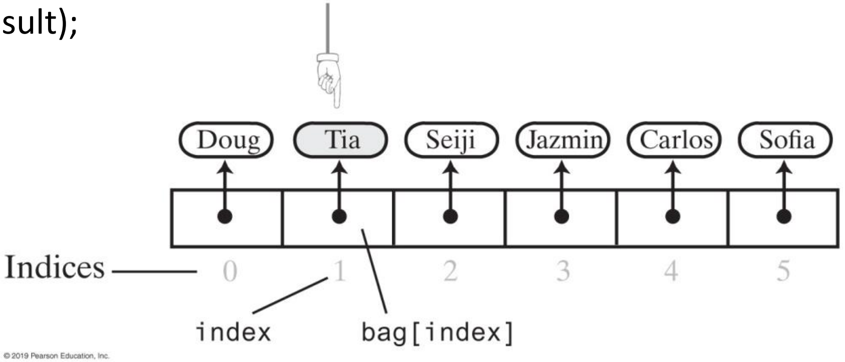
© 2019 Pearson Education, Inc.

ArrayBag
-bag: T[]
-numberOfEntries: integer
-DEFAULT_CAPACITY: integer
-integrityOK: Boolean
-MAX_CAPACITY: integer
+getCurrentSize(): integer
+isEmpty(): boolean
+add(newEntry: T): boolean
+remove(): T
+remove(anEntry: T): boolean
+clear(): void
+getFrequencyOf(anEntry: T): integer
+contains(anEntry: T): boolean
+toArray(): T[]
-isArrayFull(): boolean

Implementing More Methods

- **Removing a given entry**
  - **Search for the entry**
  - Remove the entry from the bag

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

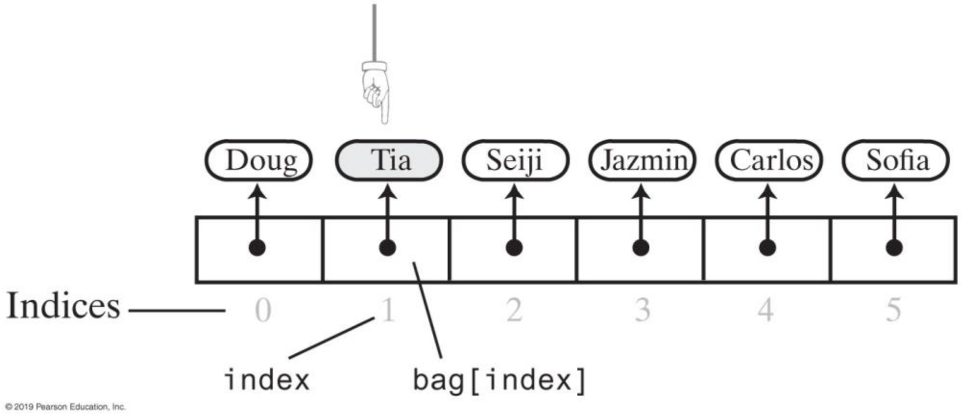


```
// Locates a given entry within the array bag.
// Returns the index of the entry, if located, or -1 otherwise.
// Precondition: checkIntegrity has been called.
private int getIndexOf(T anEntry)
{
        int where = -1;
        boolean found = false;
        int index = 0;

        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
```

Implementing More Methods

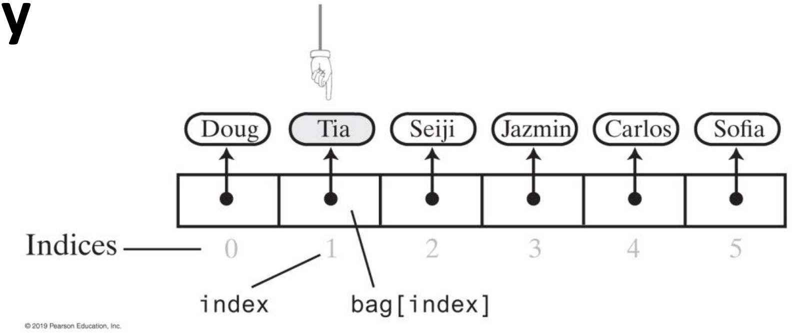
- **Removing a given entry**
  - Search for the entry
  - **Remove the entry from the bag**

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



Implementing More Methods

- **Removing a given entry**

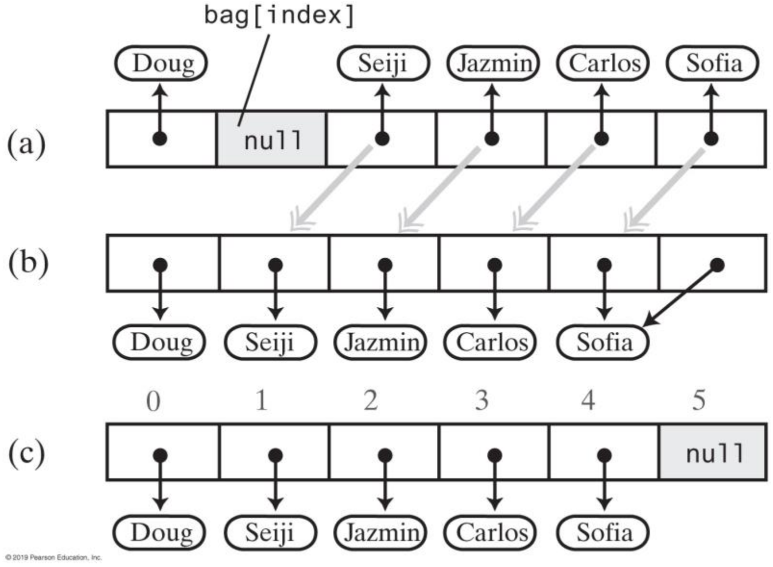


© 2019 Pearson Education, Inc.

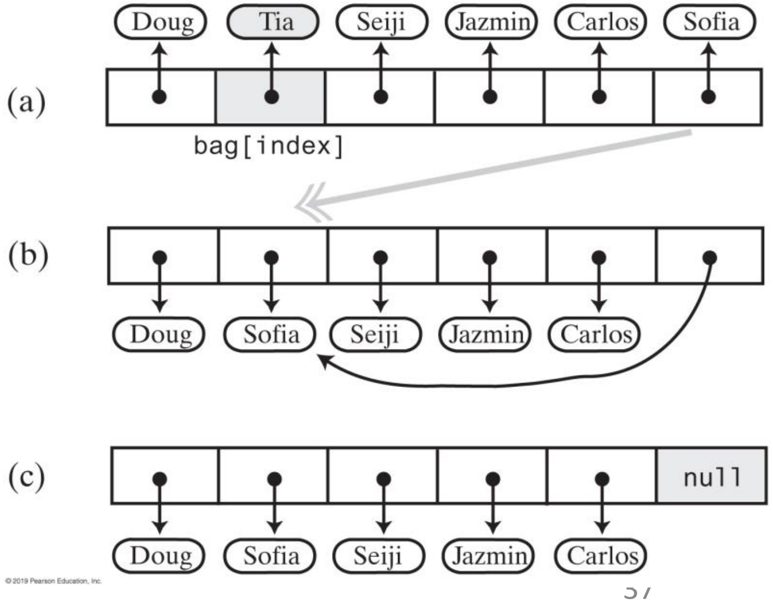
Option 1

?

Option 2

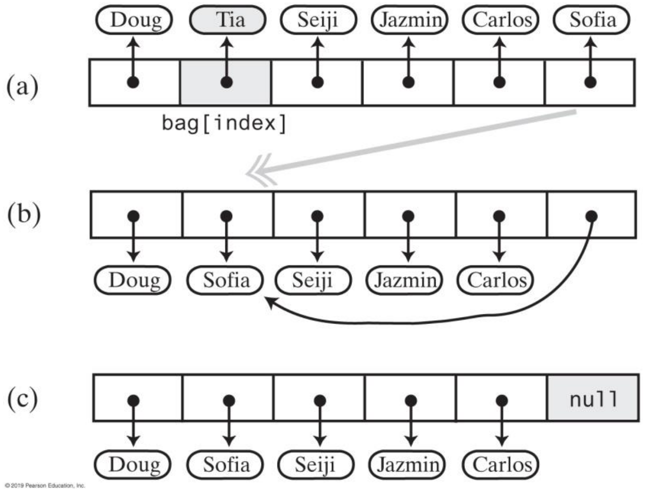


© 2019 Pearson Education, Inc.



© 2019 Pearson Education, Inc.

Implementing More Methods



© 2019 Pearson Education, Inc.

- **Removing a given entry**
  - Search for the entry
  - **Remove the entry from the bag**

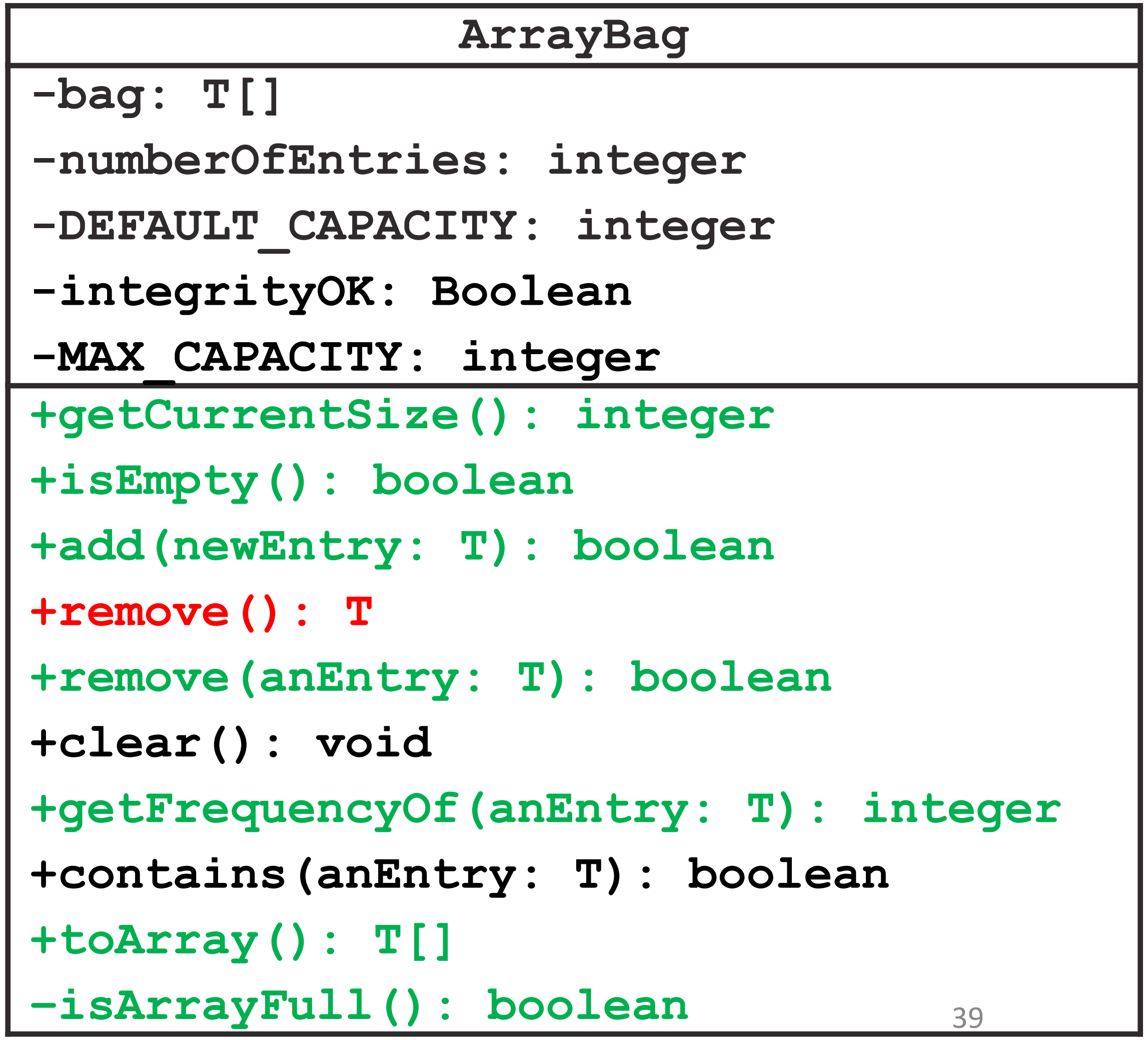
```
  /** Removes one occurrence of a given entry from this bag.
@param anEntry  The entry to be removed.
@return  True if the removal was successful, or false if not. */
  public boolean remove(T anEntry)
  {
        checkIntegrity();
        int index = getIndexOf(anEntry);
        T result = removeEntry(index);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

```
// Removes and returns the entry at a given index within the array bag.
// If no such entry exists, returns null.
// Preconditions: 0 <= givenIndex < numberOfEntries;
//               checkIntegrity 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
```

Implementing More Methods

- **Removing an unspecified entry**
  - Simply remove the last entry

```
/** Removes one unspecified entry from this bag, if possible.
@return  Either the removed entry, if the removal was successful,
      or null otherwise. */
public T remove()
{
      checkIntegrity();
     T result = removeEntry(numberOfEntries - 1);
      return result;
} // end remove
```

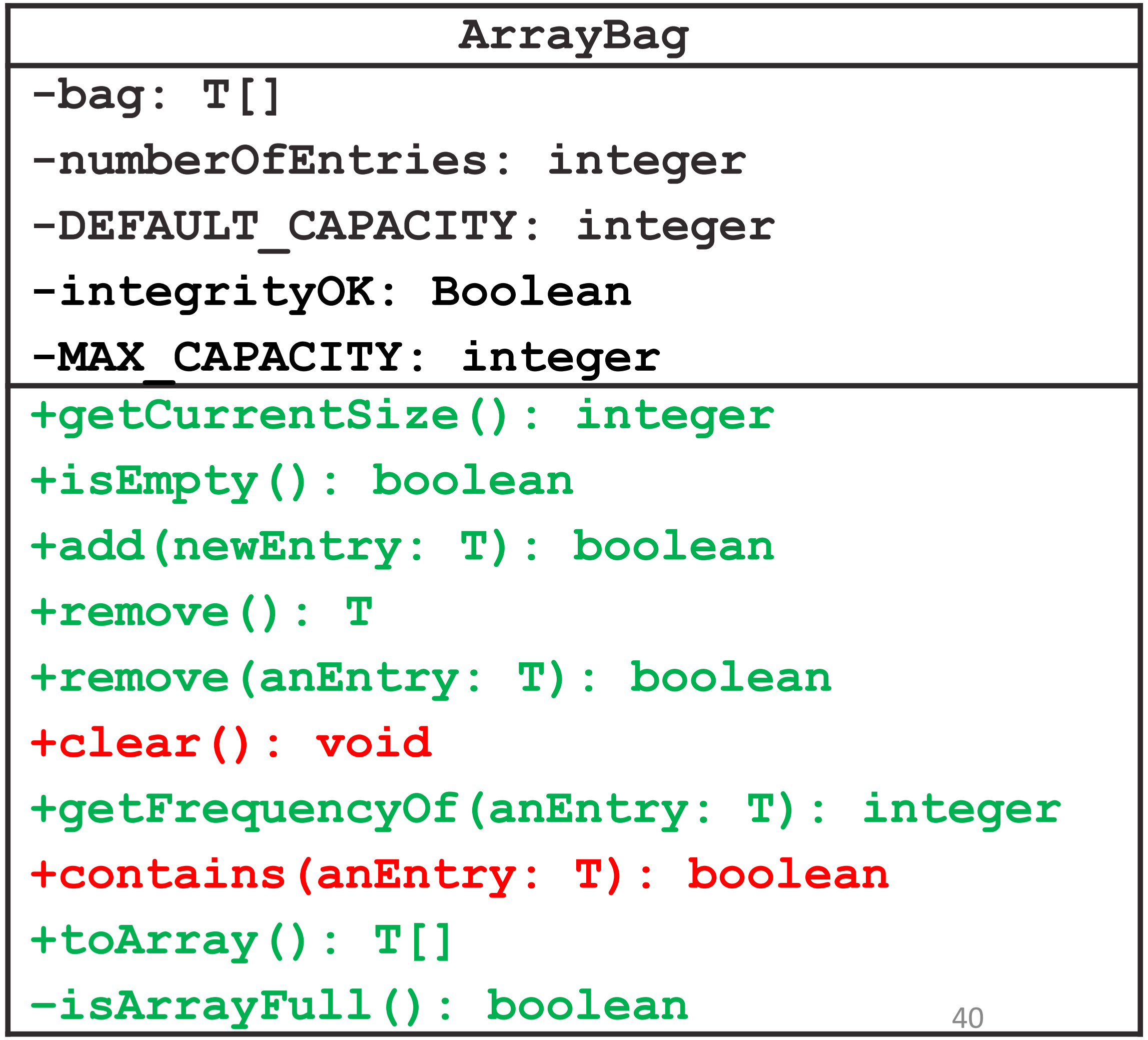


```
ArrayBag
-bag: T[]
-numberOfEntries: integer
-DEFAULT_CAPACITY: integer
-integrityOK: Boolean
-MAX_CAPACITY: integer
+getCurrentSize(): integer
+isEmpty(): boolean
+add(newEntry: T): boolean
+remove(): T
+remove(anEntry: T): boolean
+clear(): void
+getFrequencyOf(anEntry: T): integer
+contains(anEntry: T): boolean
+toArray(): T[]
-isArrayFull(): boolean
```

Implementing More Methods

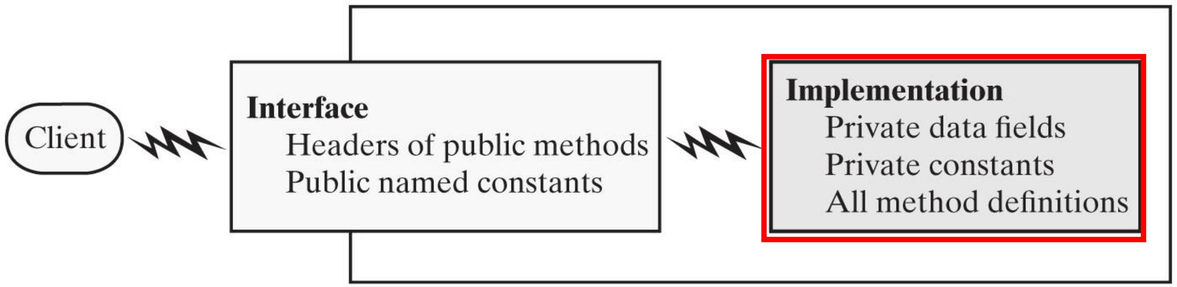
```
/** Removes all entries from this bag. */
public void clear()
{
      while (!isEmpty())
            remove();
} // end clear
```

```
/** Tests whether this bag contains a given entry.
     @param anEntry  The entry to locate.
     @return  True if this bag contains anEntry, or false otherwise. */
public boolean contains(T anEntry)
{
        checkIntegrity();
        return getIndexOf(anEntry) > -1; // or >= 0
} // end contains
```



```
ArrayBag
-bag: T[]
-numberOfEntries: integer
-DEFAULT_CAPACITY: integer
-integrityOK: Boolean
-MAX_CAPACITY: integer
+getCurrentSize(): integer
+isEmpty(): boolean
+add(newEntry: T): boolean
+remove(): T
+remove(anEntry: T): boolean
+clear(): void
+getFrequencyOf(anEntry: T): integer
+contains(anEntry: T): boolean
+toArray(): T[]
-isArrayFull(): boolean
```

Putting all pieces together to form **ArrayBag.java**



Client

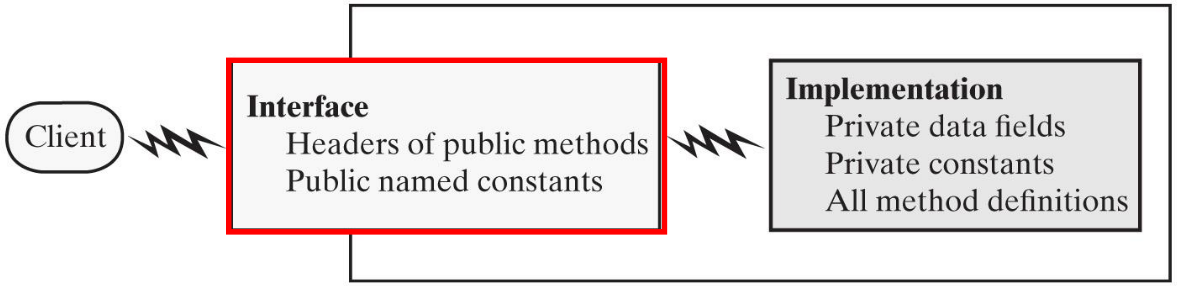
Interface
Headers of public methods
Public named constants

Implementation
Private data fields
Private constants
All method definitions

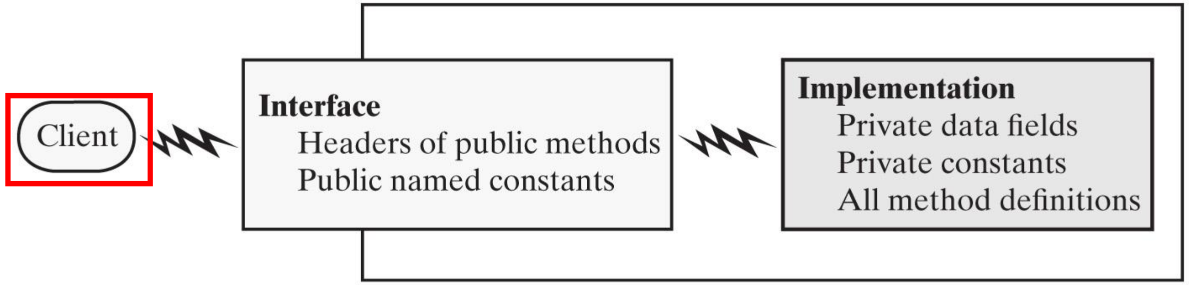
© 2019 Pearson Education, Inc.

ArrayBag
-bag: T[] -numberOfEntries: integer -DEFAULT_CAPACITY: integer -integrityOK: Boolean -MAX_CAPACITY: integer
+getCurrentSize(): integer +isEmpty(): boolean +add(newEntry: T): boolean +remove(): T +remove(anEntry: T): boolean +clear(): void +getFrequencyOf(anEntry: T): integer +contains(anEntry: T): boolean +toArray(): T[] -isArrayFull(): boolean

```
/** An interface that describes the operations of a bag of objects. */
public interface BagInterface<T>
{
        /** Gets the current number of entries in this bag.
            @return  The integer number of entries currently in the bag. */
        public int getCurrentSize();

        /** Sees whether this bag is empty.
           @return  True if the bag is empty, or false if not. */
        public boolean isEmpty();

        /** 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);

        /** Removes one unspecified entry from this bag, if possible.
           @return  Either the removed entry, if the removal was successful, or null. */
        public T remove();

        /** Removes one occurrence of a given entry from this bag, if possible.
          @param anEntry  The entry to be removed.
          @return  True if the removal was successful, or false if not. */
      public boolean remove(T anEntry);

        /** Removes all entries from this bag. */
        public void clear();

        /** 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);

        /** Tests whether this bag contains a given entry.
            @param anEntry  The entry to find.
            @return  True if the bag contains anEntry, or false if not. */
        public boolean contains(T anEntry);

        /** Retrieves all entries that are in this bag.
            @return  A newly allocated array of all the entries in the bag. Note: If the bag is empty, the
        returned array is empty. */
        public T[] toArray();

} // end BagInterface
```



BagInterface.java



© 2019 Pearson Education, Inc.

OnlineShopper.java

```java
/** A class that maintains a shopping cart for an online store. */
public class OnlineShopper
{
        public static void main(String[] args)
        {
                Item[] items = {new Item("Bird feeder", 2050),
                              new Item("Squirrel guard", 1547),
                              new Item("Bird bath", 4499),
                              new Item("Sunflower seeds", 1295)};

                BagInterface<Item> shoppingCart = new ArrayBag<>();
                int totalCost = 0;

                 // Statements that add selected items to the shopping cart:
                for (int index = 0; index < items.length; index++)
                {
                              Item nextItem = items[index]; // Simulate getting item from shopper
                              shoppingCart.add(nextItem);
                              totalCost = totalCost + nextItem.getPrice();
                } // end for

                // Simulate checkout
                while (!shoppingCart.isEmpty())
                        System.out.println(shoppingCart.remove());

                 System.out.println("Total cost: " + "\t$" + totalCost / 100 + "." + totalCost % 100);
        } // end main
} // end OnlineShopper
```

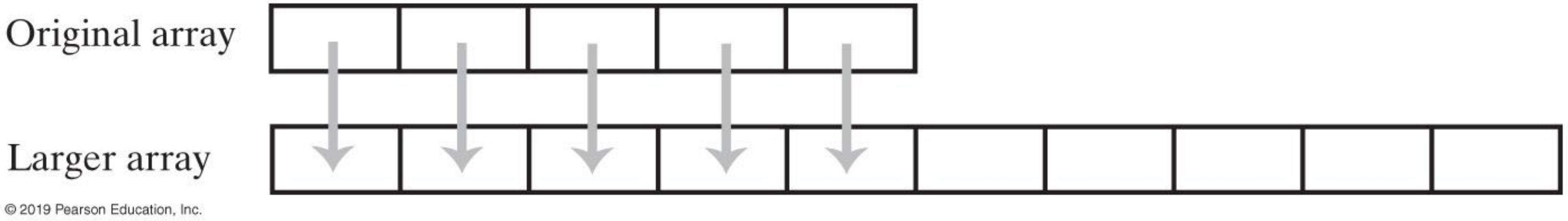
Item.java

```java
public class Item
{
  private String description;
  private int    price;

public Item(String productDescription, int productPrice)
{
     description = productDescription;
     price = productPrice;
} // end constructor

public String getDescription()
{
     return description;
} // end getDescription

public int getPrice()
{
     return price;
} // end getPrice

public String toString()
{
     return description + "\t$" + price / 100 + "." + price % 100;
} // end toString
} // end Item
```

Implementations of a Bag

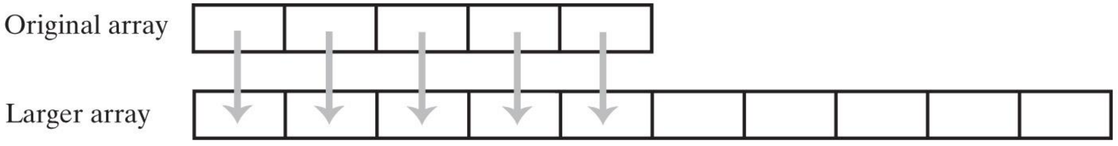
- Using Fixed-Size Arrays
- **Using Array Resizing**
- Using Linked Data

Using a fixed-size array to implement the ADT bag, therefore, limits the size of the bag.



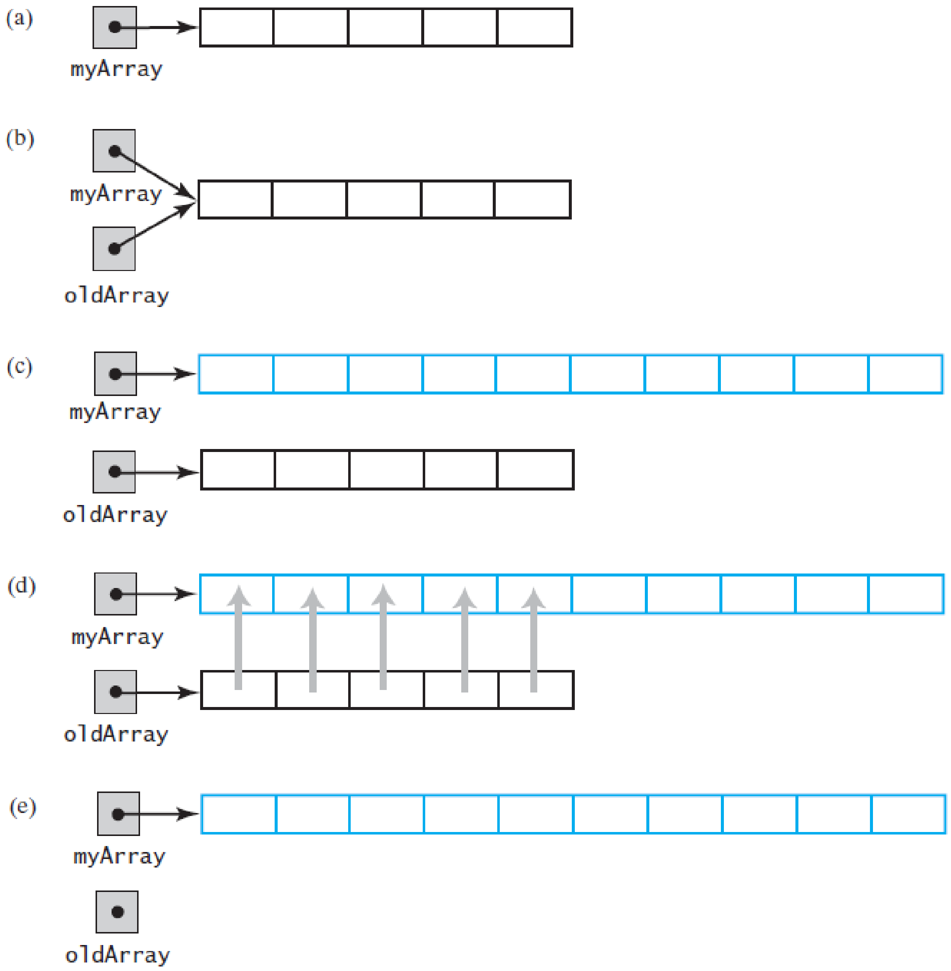
Bag Implementations That Use Array Resizing

- The process of array resizing



© 2019 Pearson Education, Inc.

FIGURE 2-9 (a) An array; (b) two references to the same array; (c) the original array variable now references a new, larger array; (d) the entries in the original array are copied to the new array; (e) the original array is discarded

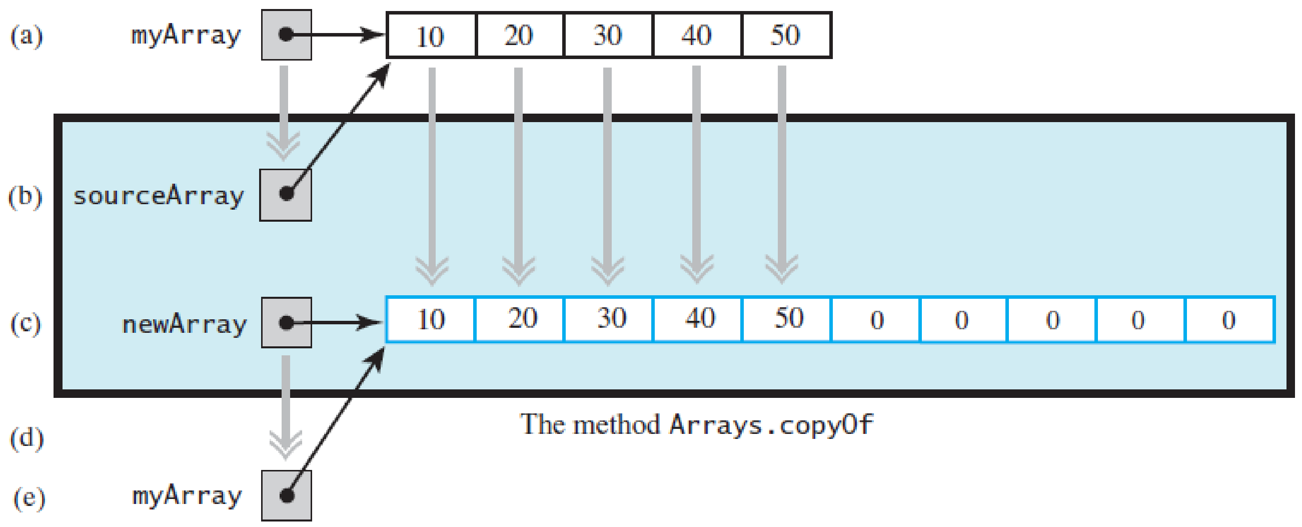


Bag Implementations That Use Array Resizing

- Doubling the size of an array each time it becomes full is a typical approach

```
import java.util.Arrays;

int[] myArray = {10, 20, 30, 40, 50};

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

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



Bag Implementations That Use Array Resizing

- Revised method add using array resizing

```
/** Adds a new entry to this bag.
   @param newEntry  The object to be added as a new entry.
   @return  True.  */
public boolean add(T newEntry)
{
  checkIntegrity();
  boolean result = true;
  if (isArrayFull())
  {
    doubleCapacity();
  } // end if

  bag[numberOfEntries] = newEntry;
  numberOfEntries++;

  return true;
} // end add
```

```
// Throws an exception if the client requests a capacity that is too large.
private void checkCapacity(int capacity)
{
  if (capacity > MAX_CAPACITY)
    throw new IllegalStateException("Attempt to create a bag whose " +
                                     "capacity exeeds allowed " +
                                     "maximum of " + MAX_CAPACITY);
} // end checkCapacity
```

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

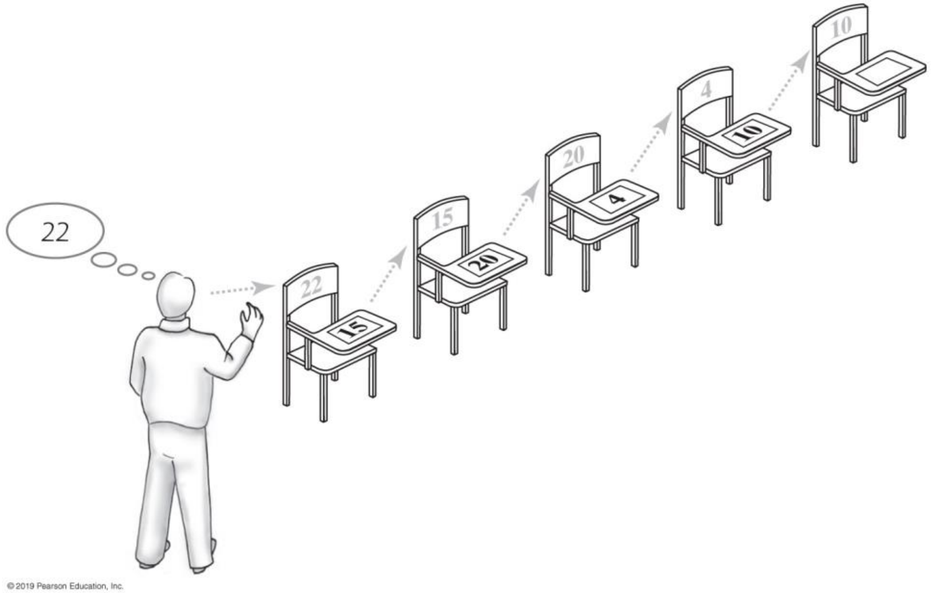
Pros and Cons of Using an Array

- +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

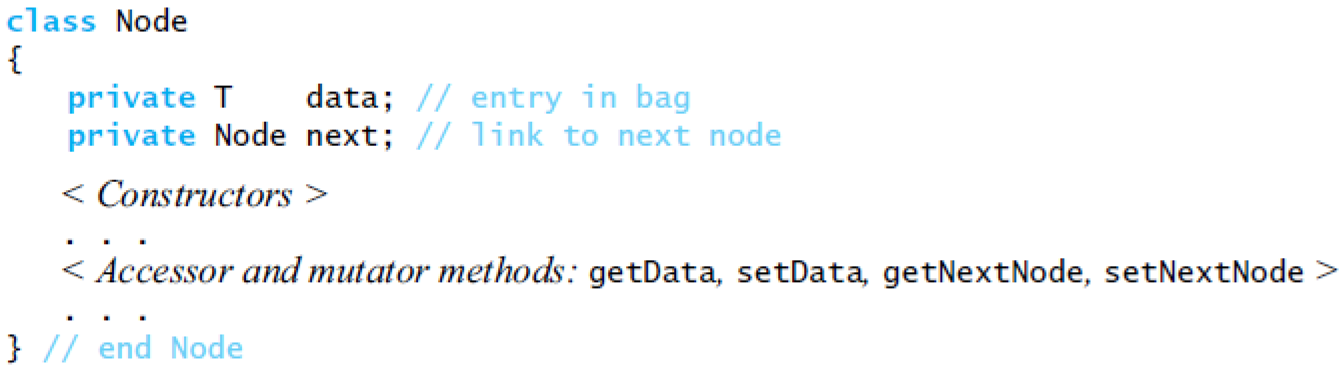
Implementations of a Bag

- Using Fixed-Size Arrays
  - Array has fixed size and may become full.
  - Alternatively may have wasted space.
- Using Array Resizing
  - Resizing is possible but requires overhead of time
- **Using Linked Data**

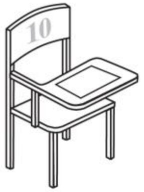


A Linked Implementation of a Bag

- Node with two data fields

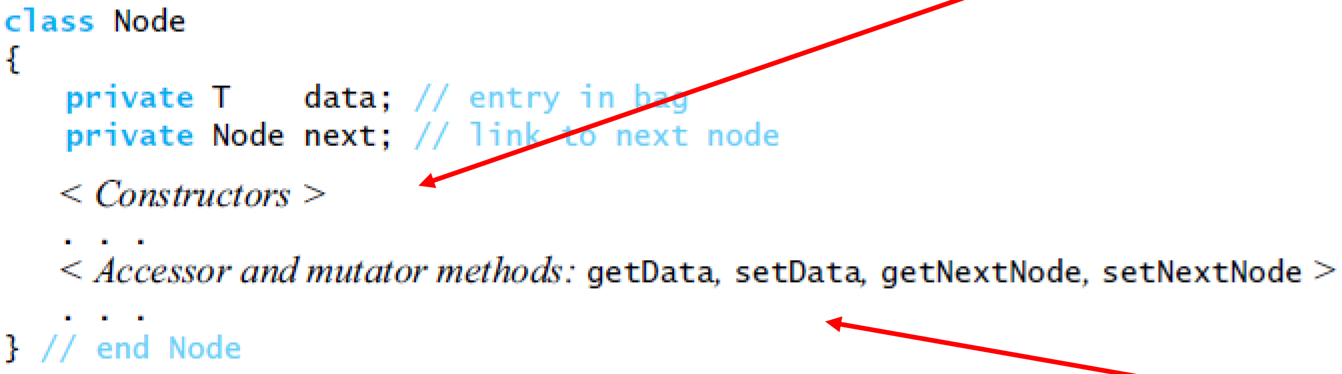


```
class Node
{
   private T    data; // entry in bag
   private Node next; // link to next node

   < Constructors >
   . . .
   < Accessor and mutator methods: getData, setData, getNextNode, setNextNode >
   . . .
} // end Node
```

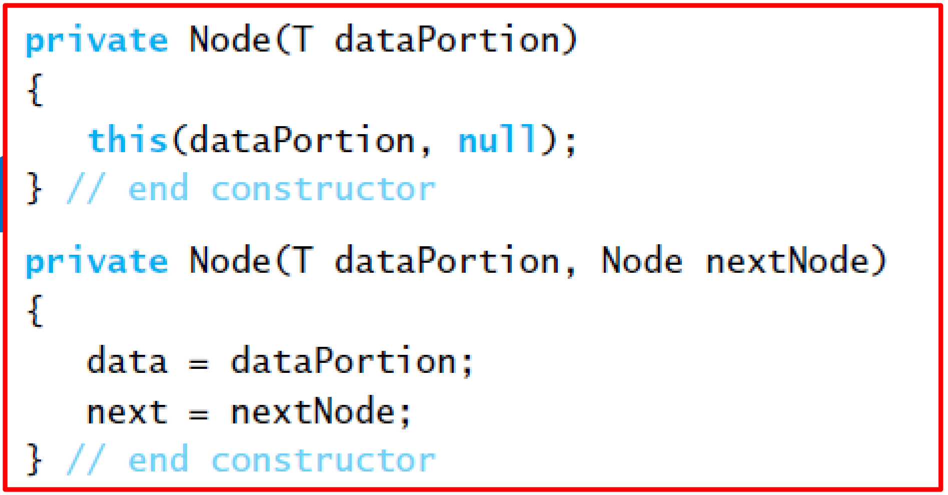


A Linked Implementation of

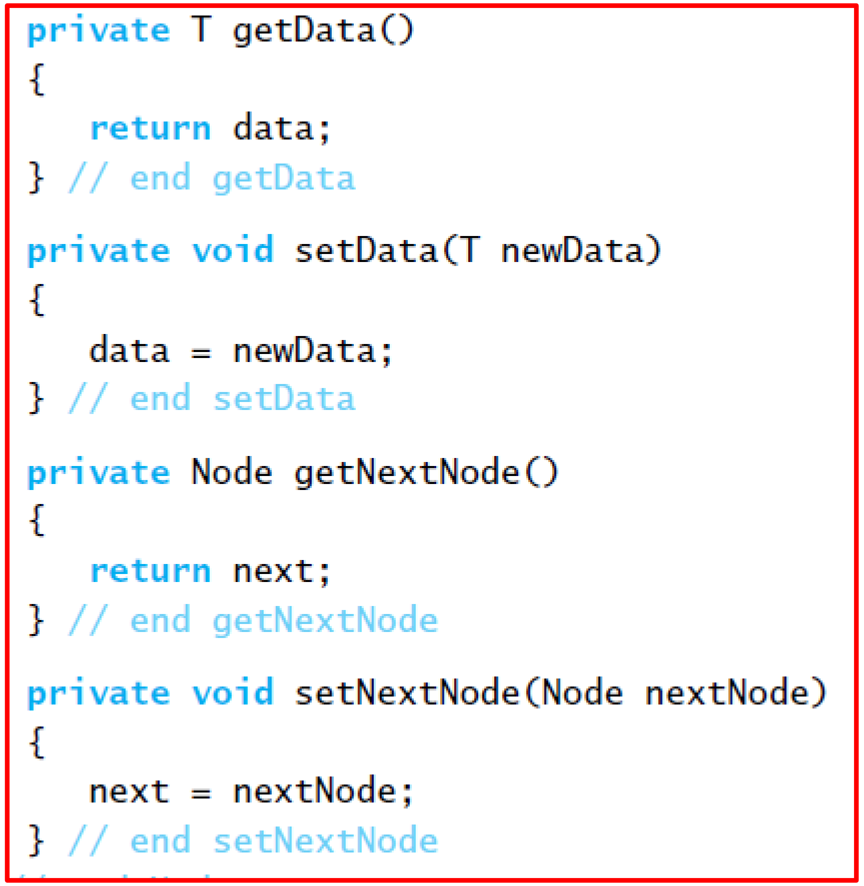
- Node with two data fields



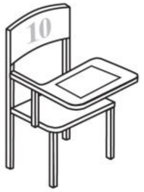
```
class Node
{
   private T    data; // entry in bag
   private Node next; // link to next node

   < Constructors >
   . . .
   < Accessor and mutator methods: getData, setData, getNextNode, setNextNode >
   . . .
} // end Node
```



```
private Node(T dataPortion)
{
   this(dataPortion, null);
} // end constructor

private Node(T dataPortion, Node nextNode)
{
   data = dataPortion;
   next = nextNode;
} // end constructor
```



```
private T getData()
{
   return data;
} // end getData

private void setData(T newData)
{
   data = newData;
} // end setData

private Node getNextNode()
{
   return next;
} // end getNextNode

private void setNextNode(Node nextNode)
{
   next = nextNode;
} // end setNextNode
```



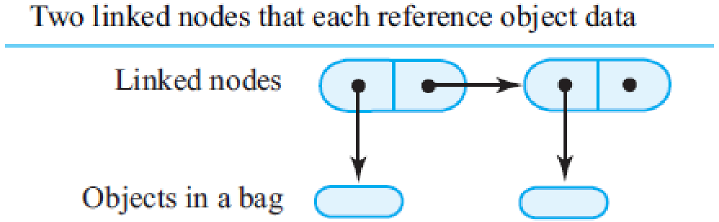
A Linked Implementation of

- Node with two data fields

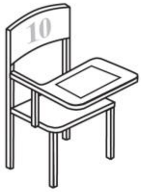
```
class Node
{
   private T    data; // entry in bag
   private Node next; // link to next node

   < Constructors >
   . . .
   < Accessor and mutator methods: getData, setData, getNextNode, setNextNode >
   . . .
} // end Node
```

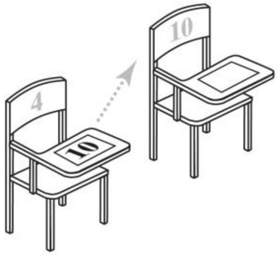
```
private Node(T dataPortion)
{
   this(dataPortion, null);
} // end constructor

private Node(T dataPortion, Node nextNode)
{
   data = dataPortion;
   next = nextNode;
} // end constructor
```

```
private T getData()
{
   return data;
} // end getData

private void setData(T newData)
{
   data = newData;
} // end setData

private Node getNextNode()
{
   return next;
} // end getNextNode

private void setNextNode(Node nextNode)
{
   next = nextNode;
} // end setNextNode
```



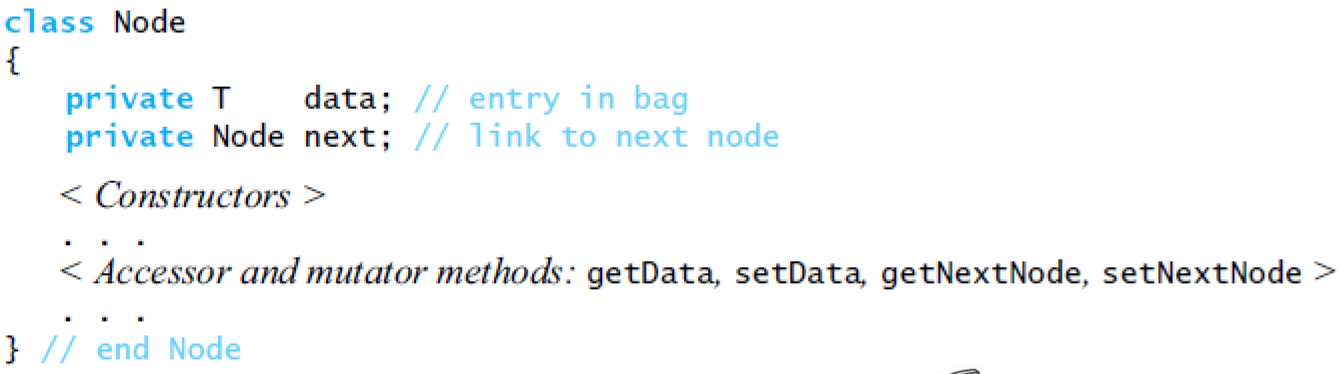
Two linked nodes that each reference object data



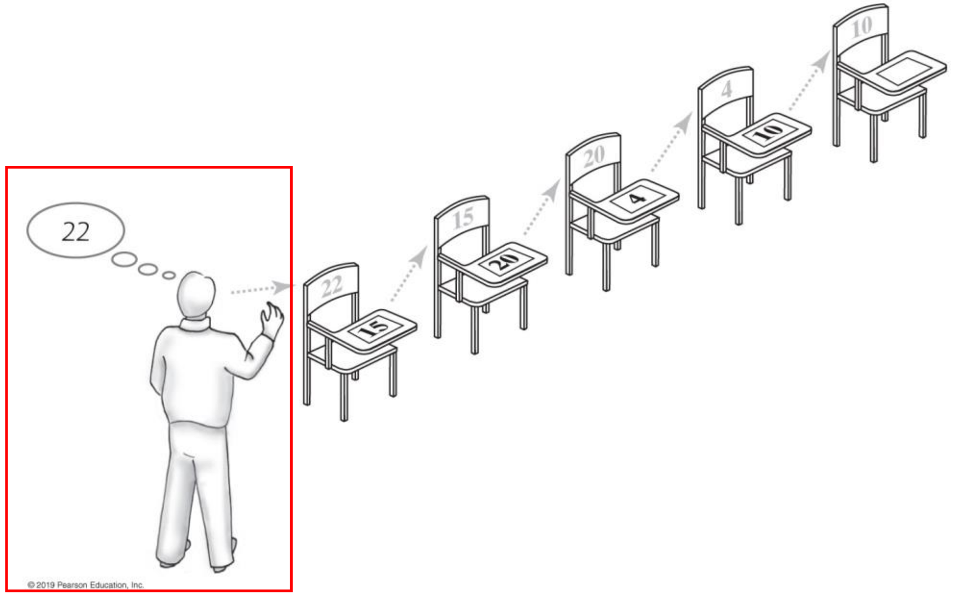


A Linked Implementation of a Bag

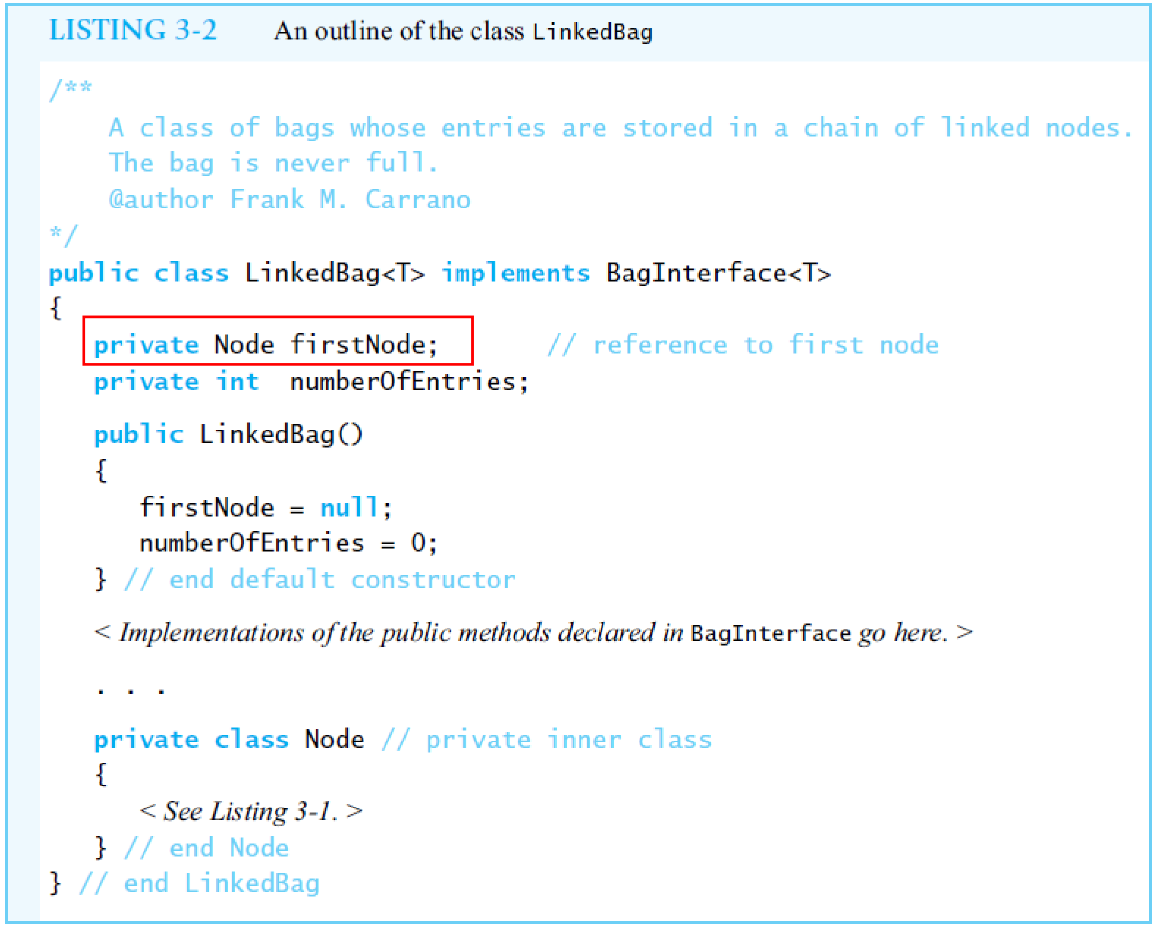
- Organize data by linking it together



```
class Node
{
   private T    data; // entry in bag
   private Node next; // link to next node
   < Constructors >
   . . .
   < Accessor and mutator methods: getData, setData, getNextNode, setNextNode >
   . . .
} // end Node
```



© 2019 Pearson Education, Inc.

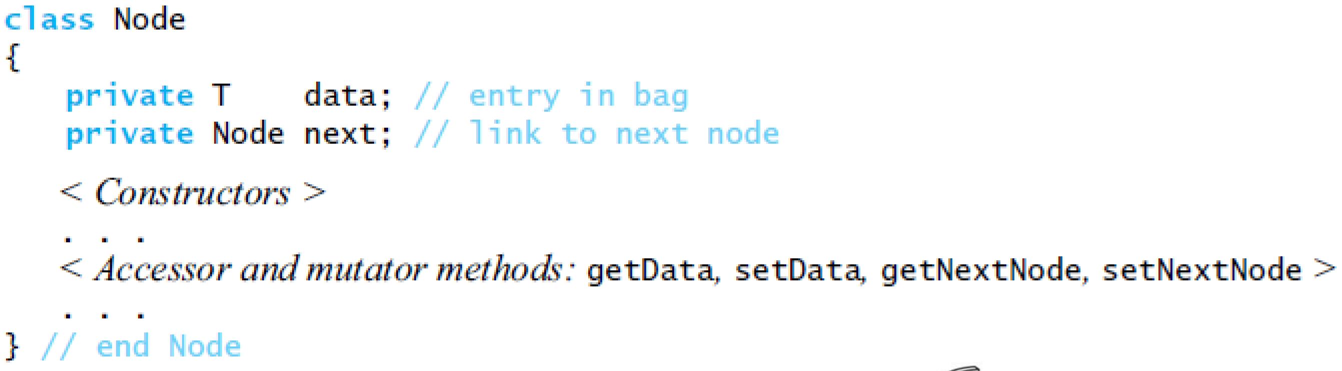


LISTING 3-2 An outline of the class LinkedBag

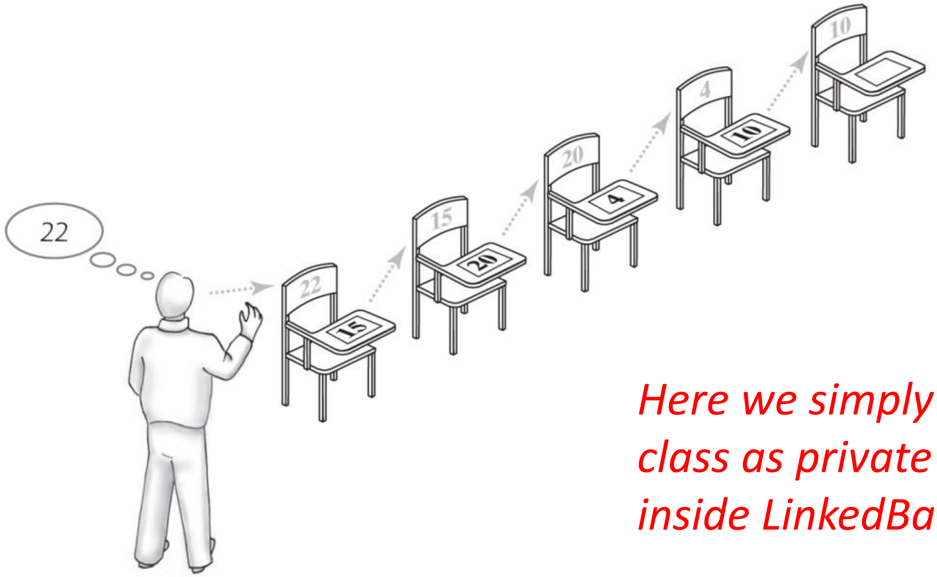
```
/**
    A class of bags whose entries are stored in a chain of linked nodes.
    The bag is never full.
    @author Frank M. Carrano
*/
public class LinkedBag<T> implements BagInterface<T>
{
   private Node firstNode;        // reference to first node
   private int  numberOfEntries;

   public LinkedBag()
   {
      firstNode = null;
      numberOfEntries = 0;
   } // end default constructor

   < Implementations of the public methods declared in BagInterface go here. >

   . . .

   private class Node // private inner class
   {
      < See Listing 3-1. >
   } // end Node
} // end LinkedBag
```

A Linked Implementation of a Bag

- Organize data by linking it together

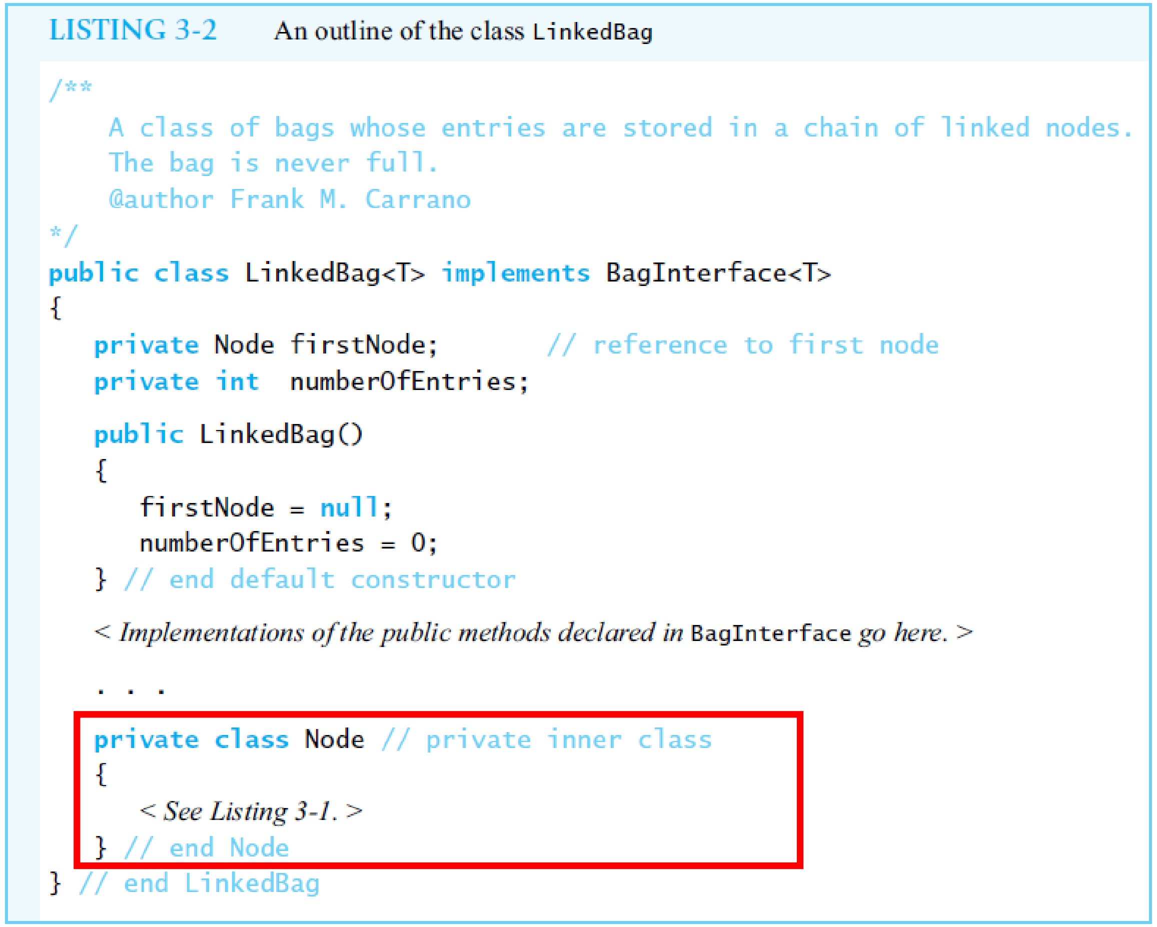


```
class Node
{
   private T    data; // entry in bag
   private Node next; // link to next node

   < Constructors >
   . . .
   < Accessor and mutator methods: getData, setData, getNextNode, setNextNode >
   . . .
} // end Node
```



© 2019 Pearson Education, Inc.

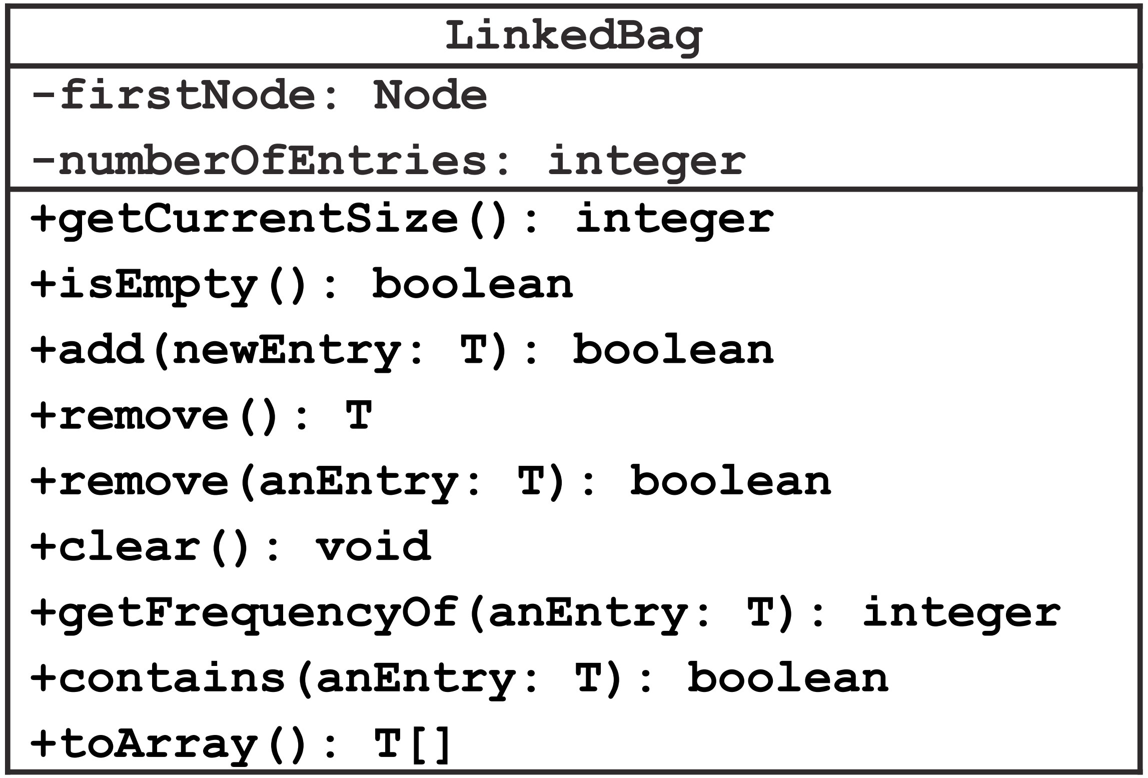
Here we simply place Node class as private class member inside LinkedBag class



LISTING 3-2 An outline of the class LinkedBag

```
/**
   A class of bags whose entries are stored in a chain of linked nodes.
   The bag is never full.
   @author Frank M. Carrano
*/
public class LinkedBag<T> implements BagInterface<T>
{
   private Node firstNode;       // reference to first node
   private int  numberOfEntries;

   public LinkedBag()
   {
      firstNode = null;
      numberOfEntries = 0;
   } // end default constructor

   < Implementations of the public methods declared in BagInterface go here. >

   . . .

   private class Node // private inner class
   {
      < See Listing 3-1. >
   } // end Node
} // end LinkedBag
```

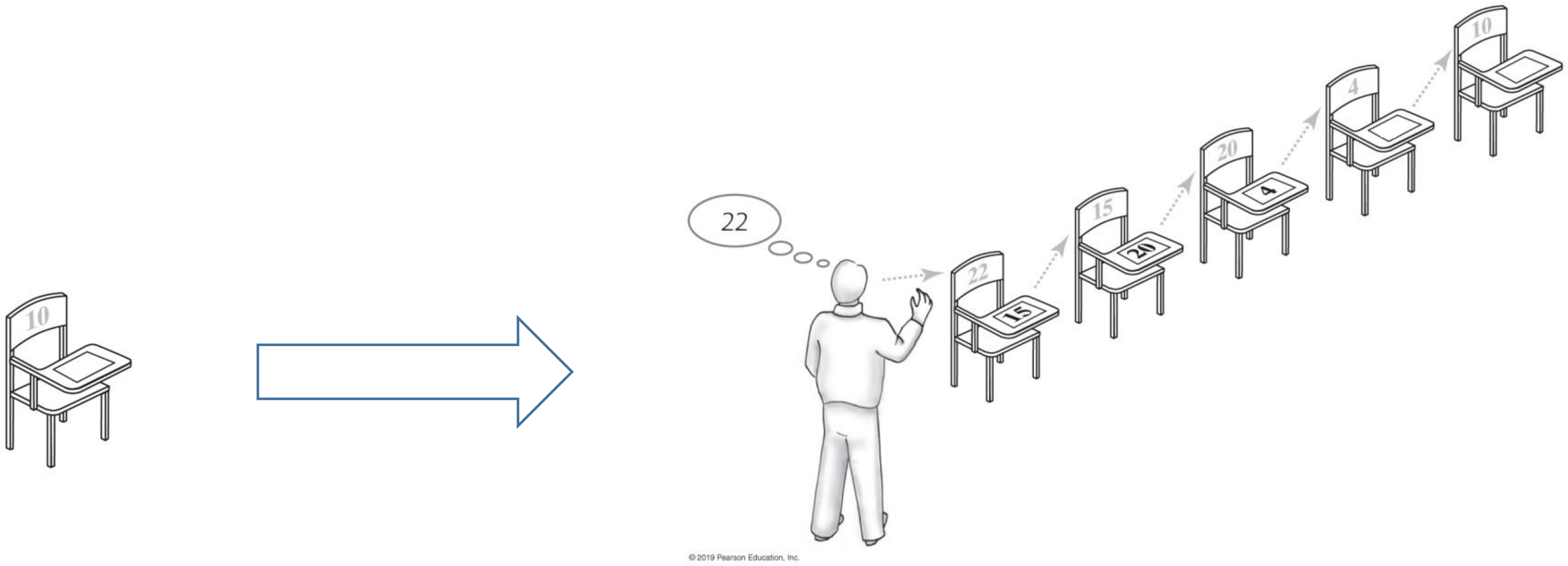
A Linked Implementation of a Bag



```
LinkedBag
-firstNode: Node
-numberOfEntries: integer
+getCurrentSize(): integer
+isEmpty(): boolean
+add(newEntry: T): boolean
+remove(): T
+remove(anEntry: T): boolean
+clear(): void
+getFrequencyOf(anEntry: T): integer
+contains(anEntry: T): boolean
+toArray(): T[]
```

A Linked Implementation of a Bag

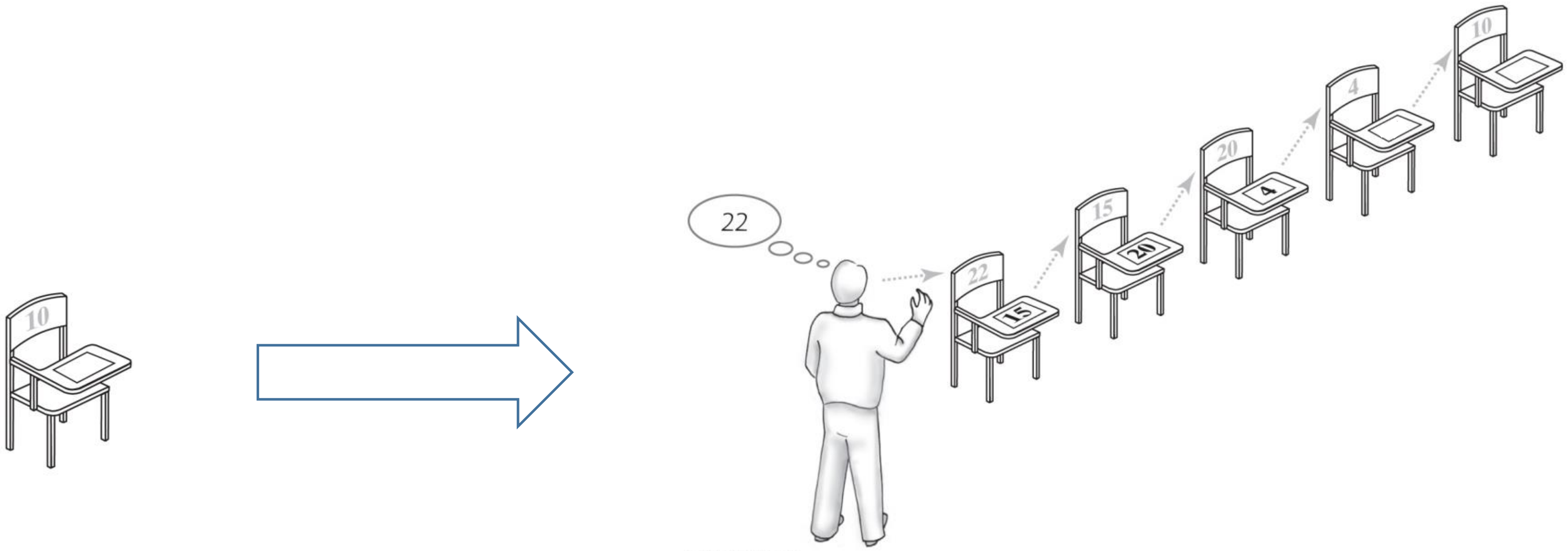
- The method **add**



© 2019 Pearson Education, Inc.

A Linked Implementation of a Bag

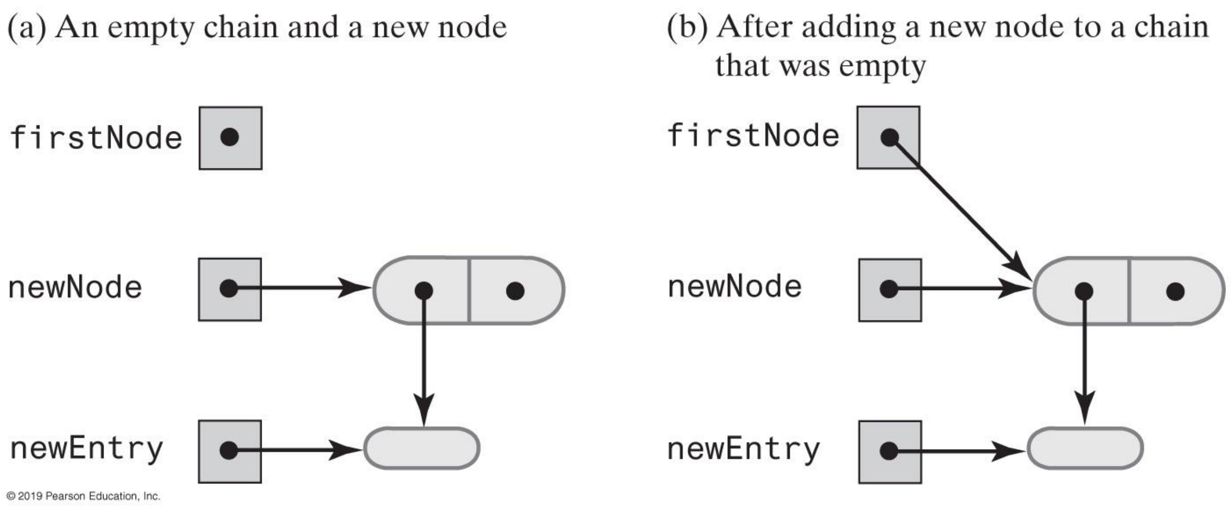
- The method **add**
  - Adding a new node to an empty chain
  - Adding a new node to a non-empty chain



© 2019 Pearson Education, Inc.

A Linked Implementation of a Bag

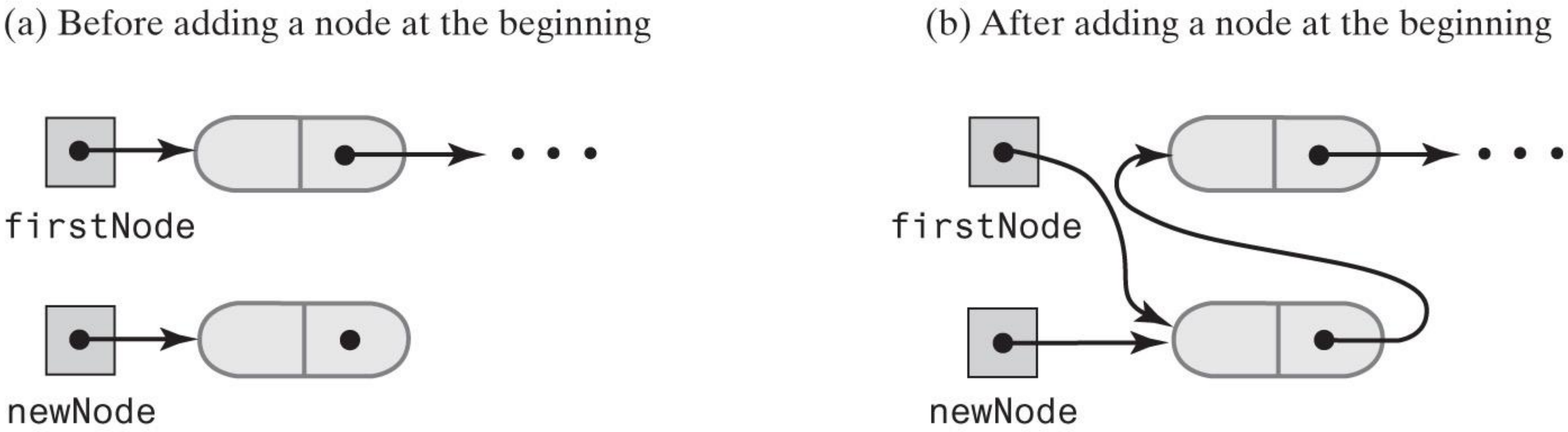
- The method **add**
  - **Adding a new node to an empty chain**
  - Adding a new node to a non-empty chain


(a) An empty chain and a new node
firstNode
newNode
newEntry
(b) After adding a new node to a chain that was empty
firstNode
newNode
newEntry

© 2019 Pearson Education, Inc.

A Linked Implementation of a Bag

- The method **add**
  - Adding a new node to an empty chain
  - **Adding a new node to a non-empty chain**



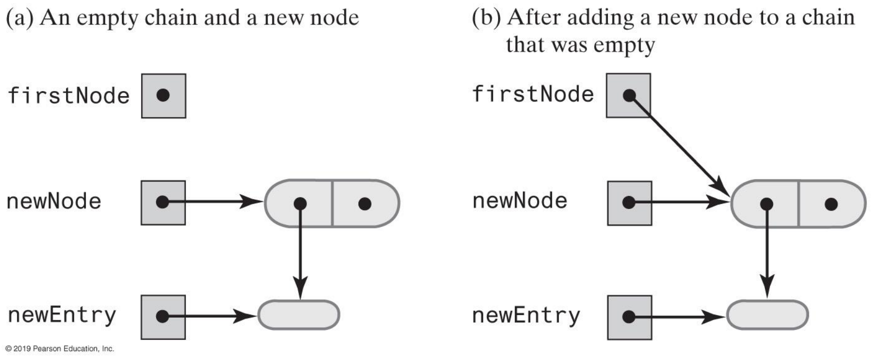
(a) Before adding a node at the beginning

(b) After adding a node at the beginning

A Linked Implementation of a Bag

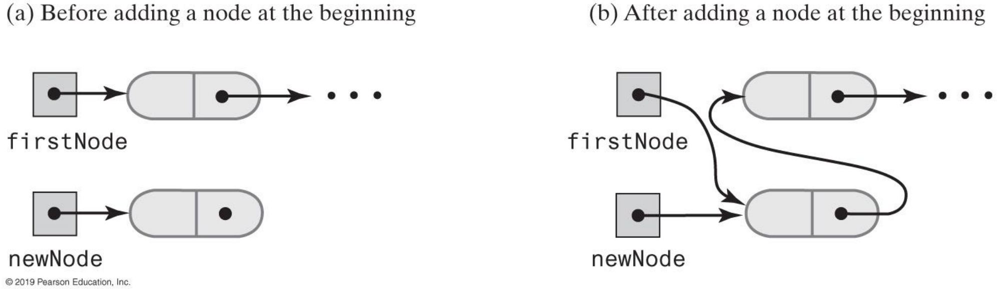
- The method **add**

```
/** 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)        // OutOfMemoryError possible
{
     // Add to beginning of chain:
     Node newNode = new Node(newEntry);
     newNode.setNextNode(firstNode); // Make new node reference rest of
     chain // (firstNode is null if chain is empty)

     firstNode = newNode;      // New node is at beginning of chain
     numberOfEntries++;

     return true;
} // end add
```


(a) An empty chain and a new node

(b) After adding a new node to a chain that was empty

© 2019 Pearson Education, Inc.

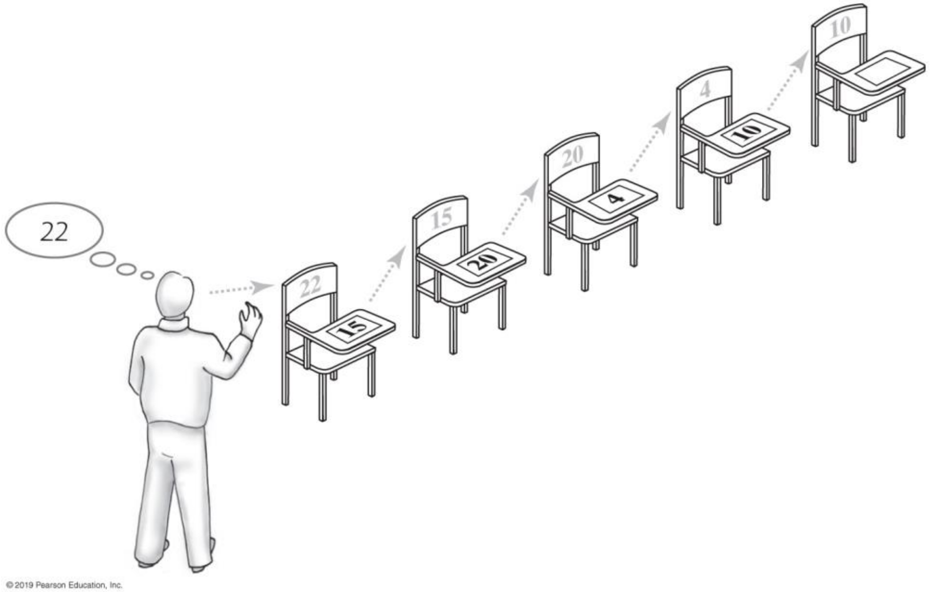

(a) Before adding a node at the beginning

(b) After adding a node at the beginning

© 2019 Pearson Education, Inc.

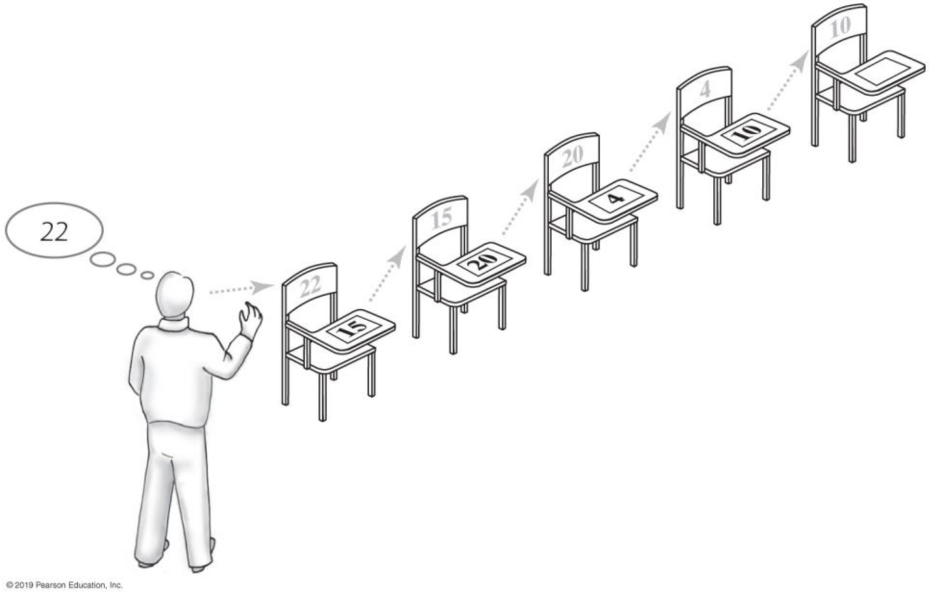
A Linked Implementation of a Bag

- The method **remove**



A Linked Implementation of a Bag

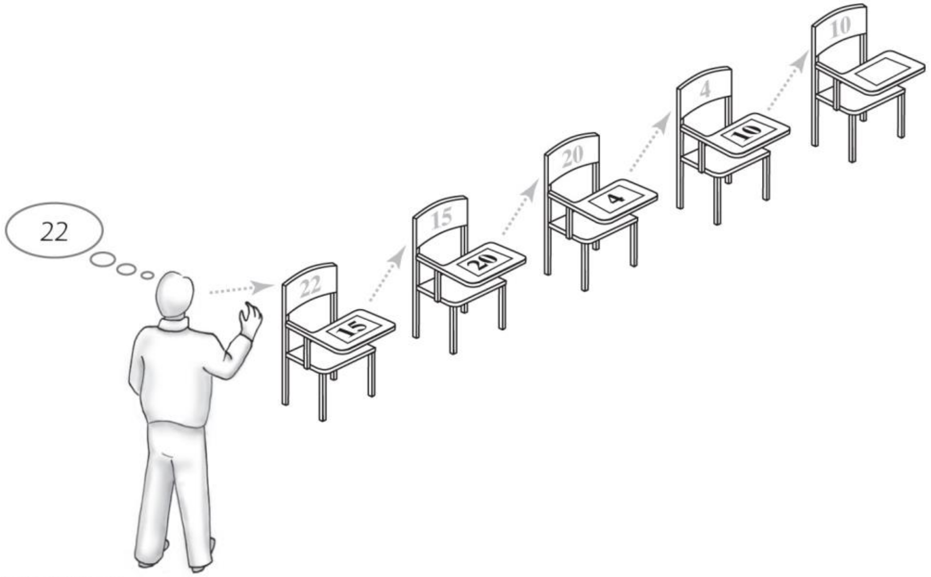
- The method **remove**
  - Removing an unspecified entry from a bag
  - Removing a given entry from a bag



© 2019 Pearson Education, Inc.

A Linked Implementation of a Bag

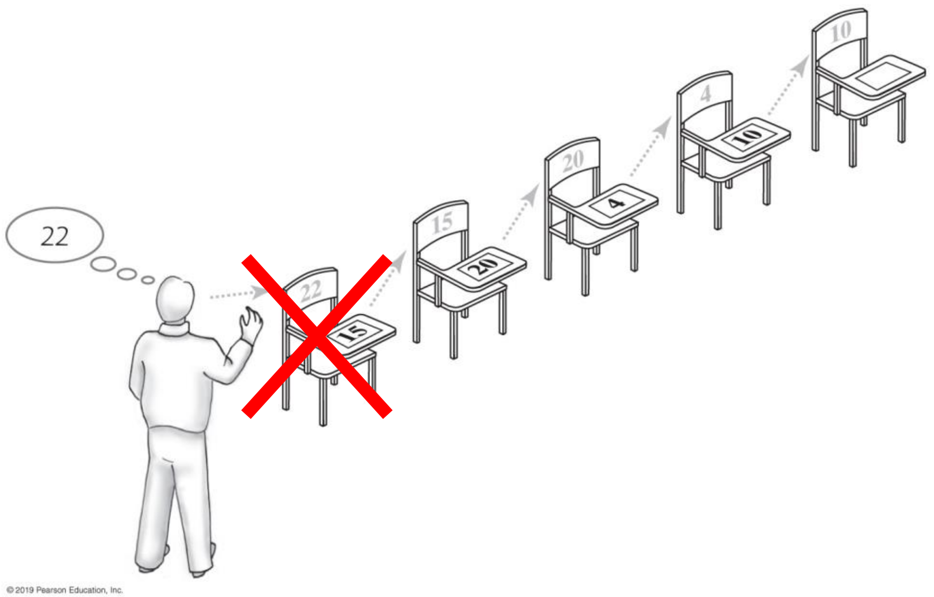
- The method **remove**
  - **Removing an unspecified entry from a bag**
  - Removing a given entry from a bag



© 2019 Pearson Education, Inc.

A Linked Implementation of a Bag

- The method **remove**
  - **Removing an unspecified entry from a bag**
  - Removing a given entry from a bag

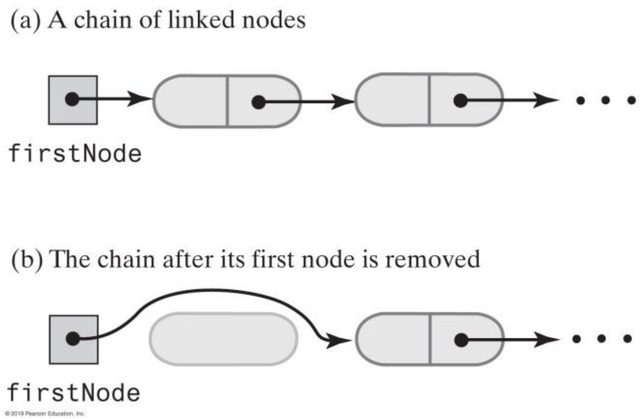


© 2019 Pearson Education, Inc.

A Linked Implementation of a Bag

- The method **remove**
  - **Removing an unspecified entry from a bag**
  - Removing a given entry from a bag

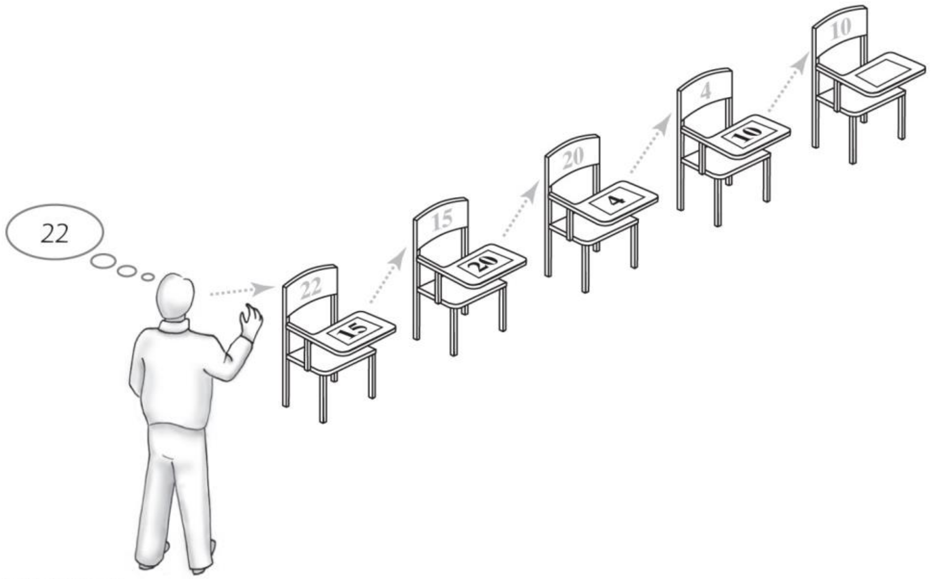
```java
/** Removes one unspecified entry from this bag, if possible.
@return  Either the removed entry, if the removal was successful, or null. */
public T remove()
{
    T result = null;
    if (firstNode != null)
    {
         result = firstNode.getData();
         firstNode = firstNode.getNextNode(); // Remove first node from chain
         numberOfEntries--;
     } // end if

     return result;
} // end remove
```


(a) A chain of linked nodes

(b) The chain after its first node is removed

A Linked Implementation of a Bag

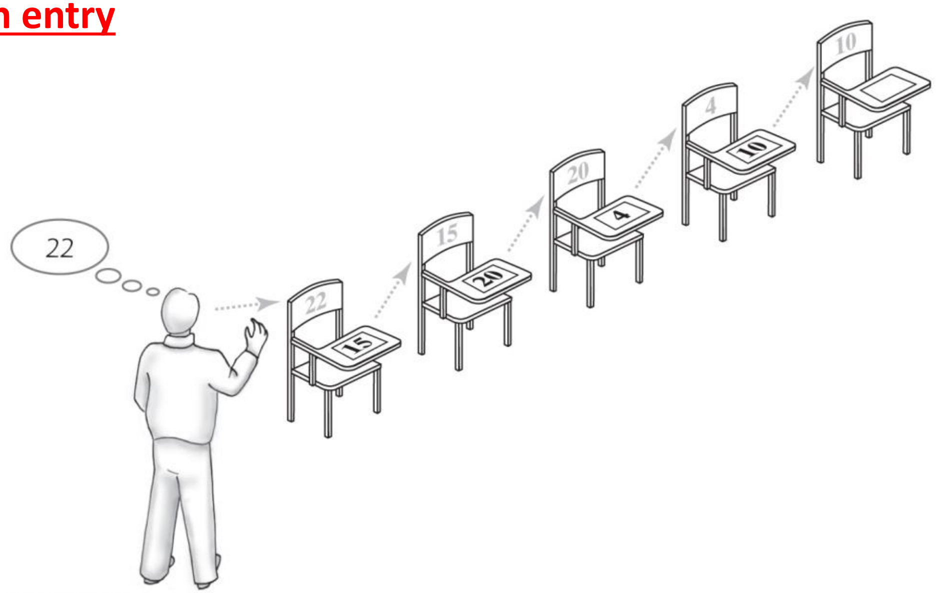
- The method **remove**
  - Removing an unspecified entry from a bag
  - **Removing a given entry from a bag**



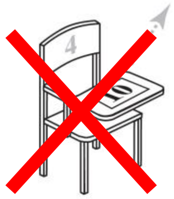
© 2019 Pearson Education, Inc.

A Linked Implementation of a Bag

- The method **remove**
  - Removing an unspecified entry from a bag
  - **Removing a given entry from a bag**
    - Search for the given entry in a bag
    - Remove the given entry



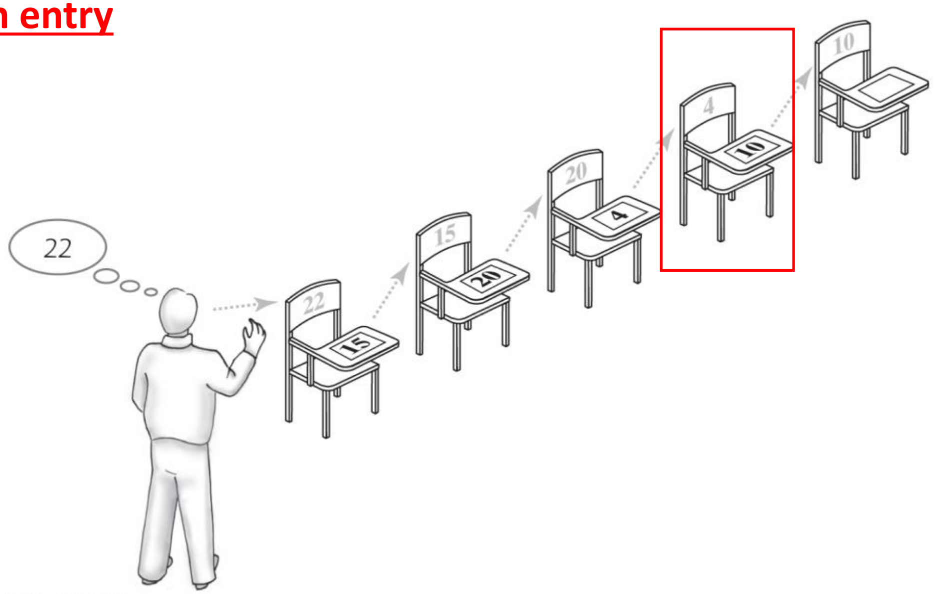
© 2019 Pearson Education, Inc.



A Linked Implementation of a Bag

- The method **remove**
  - Removing an unspecified entry from a bag
  - **Removing a given entry from a bag**
    - Search for the given entry in a bag
    - **Remove the given entry**

How can be remove the given entry easily and efficiently?

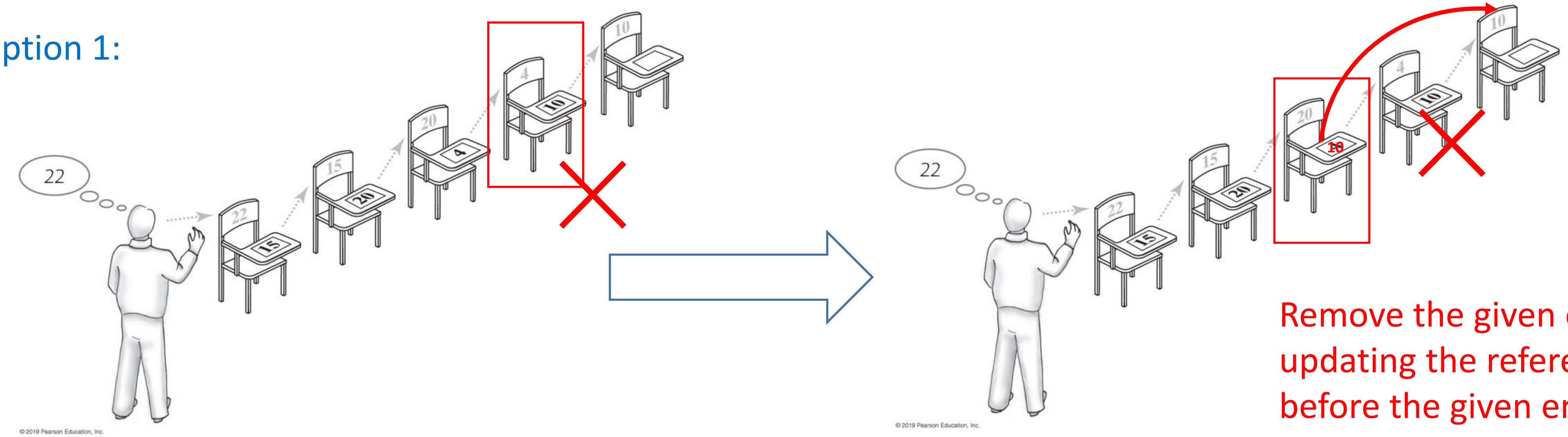


© 2019 Pearson Education, Inc.

A Linked Implementation of a Bag

- The method **remove**
  - Removing an unspecified entry from a bag
  - **Removing a given entry from a bag**
    - Search for the given entry in a bag
    - **Remove the given entry**

Option 1:

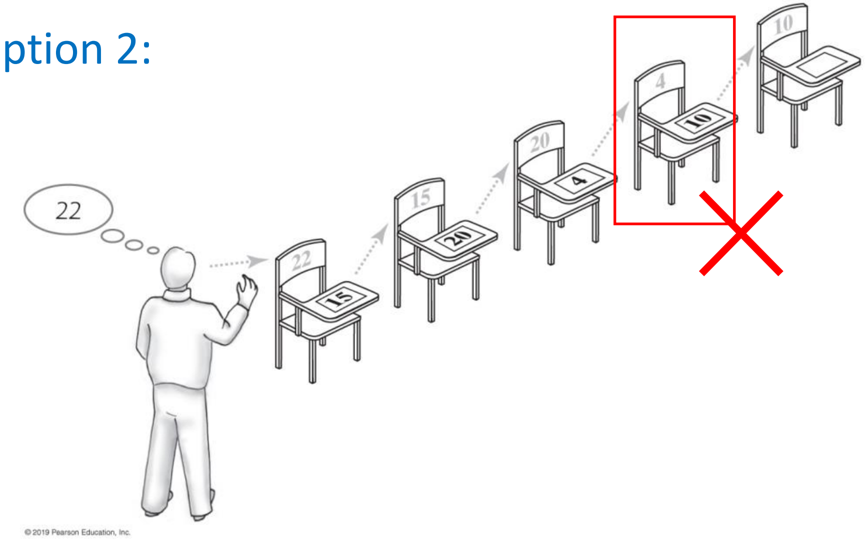

22
22
15
20
4
10
22
22
15
20
10
4
10
© 2019 Pearson Education, Inc.
© 2019 Pearson Education, Inc.

Remove the given entry directly by updating the reference in the entry before the given entry

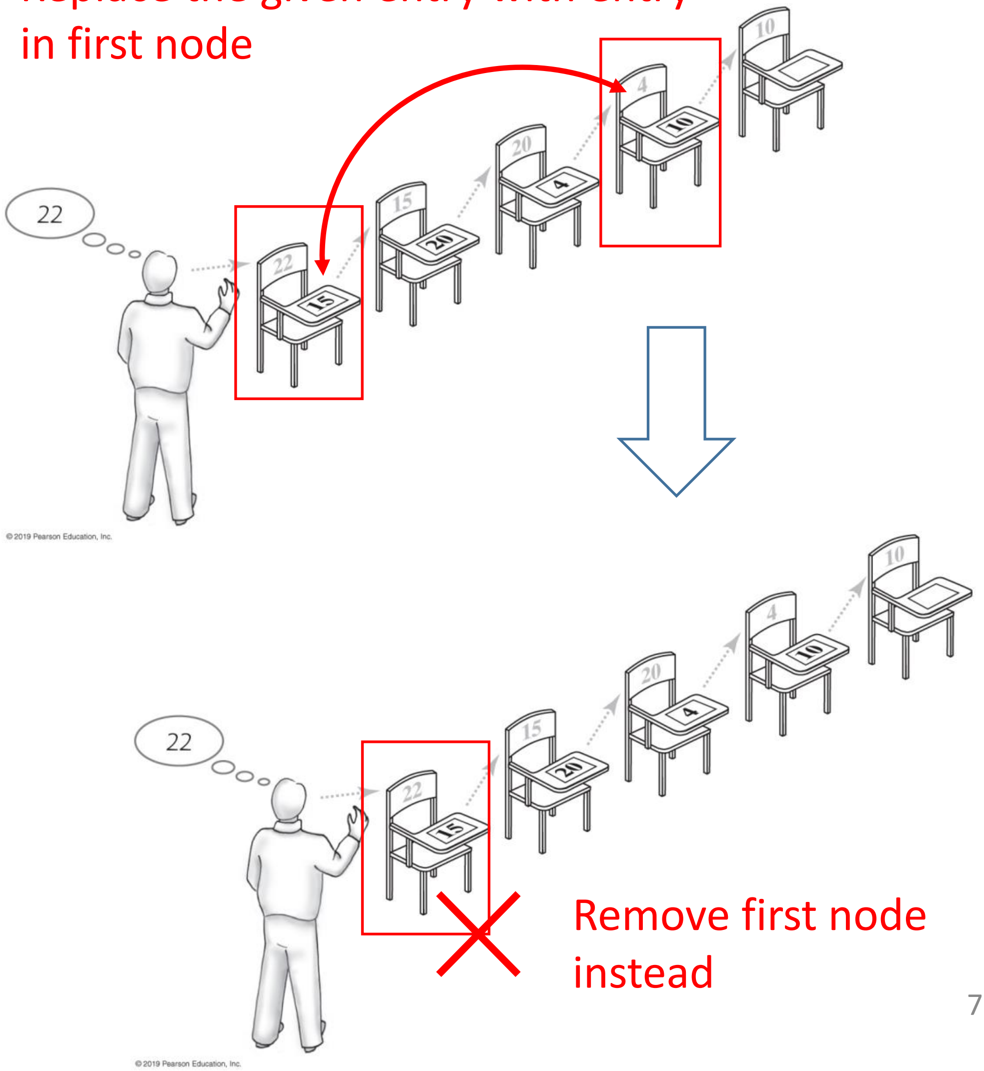
A Linked Implementation of a Bag

- The method **remove**
  - Removing an unspecified entry from a bag
  - **Removing a given entry from a bag**
    - Search for the given entry in a bag
    - Remove the given entry

Option 2:



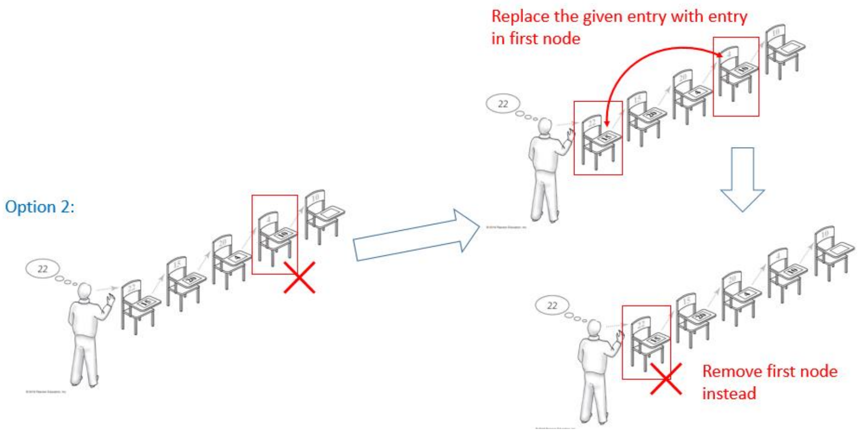
Replace the given entry with entry in first node



22
22
15
15
20
20
4
4
10
10
© 2019 Pearson Education, Inc.
22
22
15
15
20
20
4
4
10
10
© 2019 Pearson Education, Inc.

Remove first node instead

```
// Locates a given entry within this bag.
// Returns a reference to the node containing the //
entry, if located, or null otherwise.
private Node getReferenceTo(T anEntry)
{
     boolean found = false;
     Node currentNode = firstNode;

     while (!found && (currentNode != null))
     {
          if (anEntry.equals(currentNode.getData()))
                   found = true;
          else
          currentNode = currentNode.getNextNode();
     } // end while

     return currentNode;
} // end getReferenceTo
```

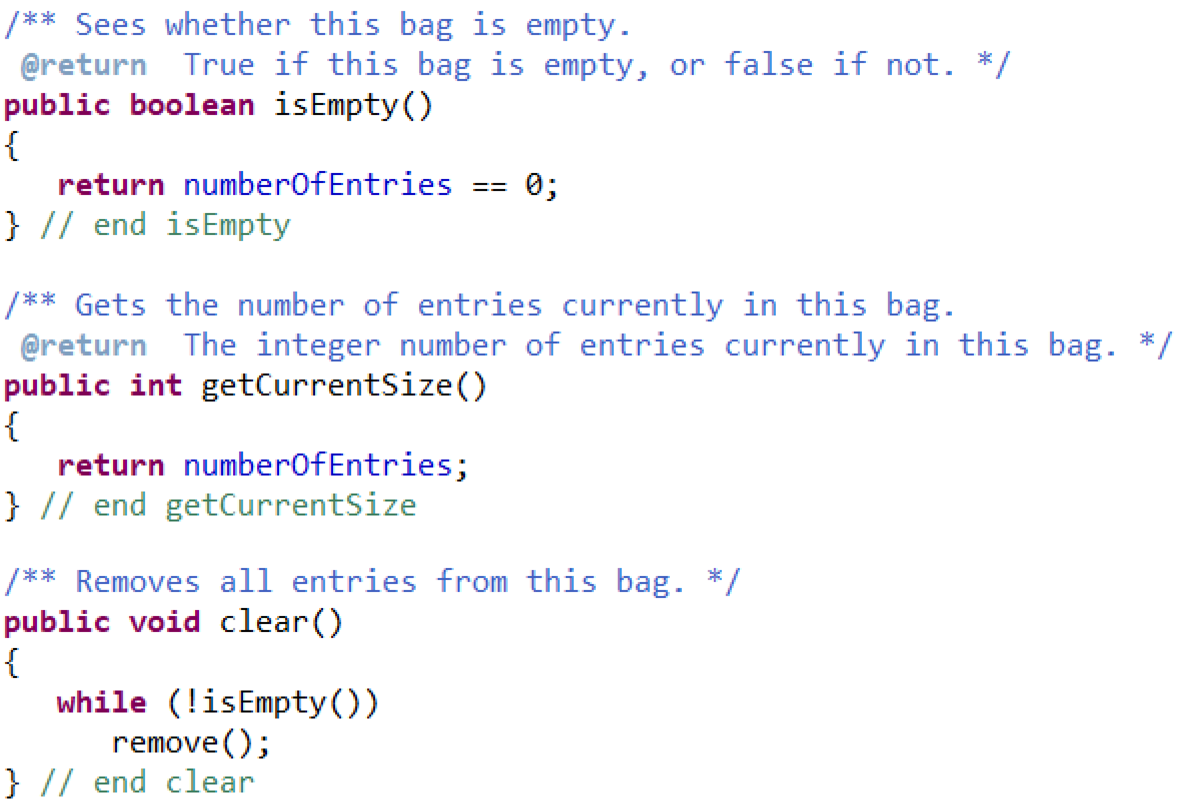

Option 2:
22
Replace the given entry with entry in first node
22
22
Remove first node instead

```
/** Removes one occurrence of a given entry from this bag,
if possible.
    @param anEntry  The entry to be removed.
    @return  True if the removal was successful, or false
otherwise. */
   public boolean remove(T anEntry)
   {
      boolean result = false;
      Node nodeN = getReferenceTo(anEntry);

      if (nodeN != null)
      {
         // Replace located entry with entry in first node
         nodeN.setData(firstNode.getData());
         // Remove first node
         firstNode = firstNode.getNextNode();

         numberOfEntries--;

         result = true;
      } // end if

      return result;
   } // end remove
```

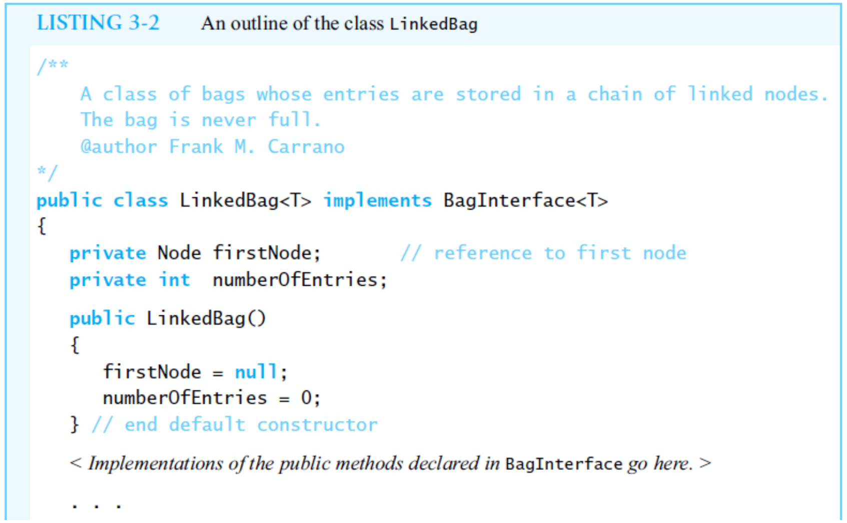
- **Removing a given entry from a bag**
    - **Search for the given entry in a bag**
    - **Remove the given entry**

A Linked Implementation of a Bag

- The methods **isEmpty**, **getCurrentSize**, and **clear**

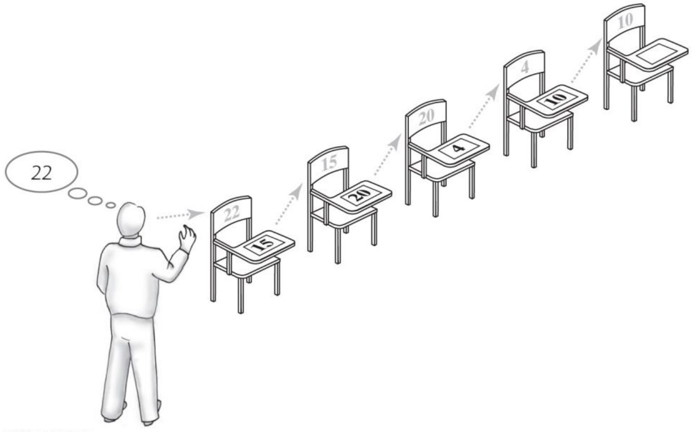


```
/** Sees whether this bag is empty.
 @return  True if this bag is empty, or false if not. */
public boolean isEmpty()
{
   return numberOfEntries == 0;
} // end isEmpty

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

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



LISTING 3-2 An outline of the class `LinkedBag`

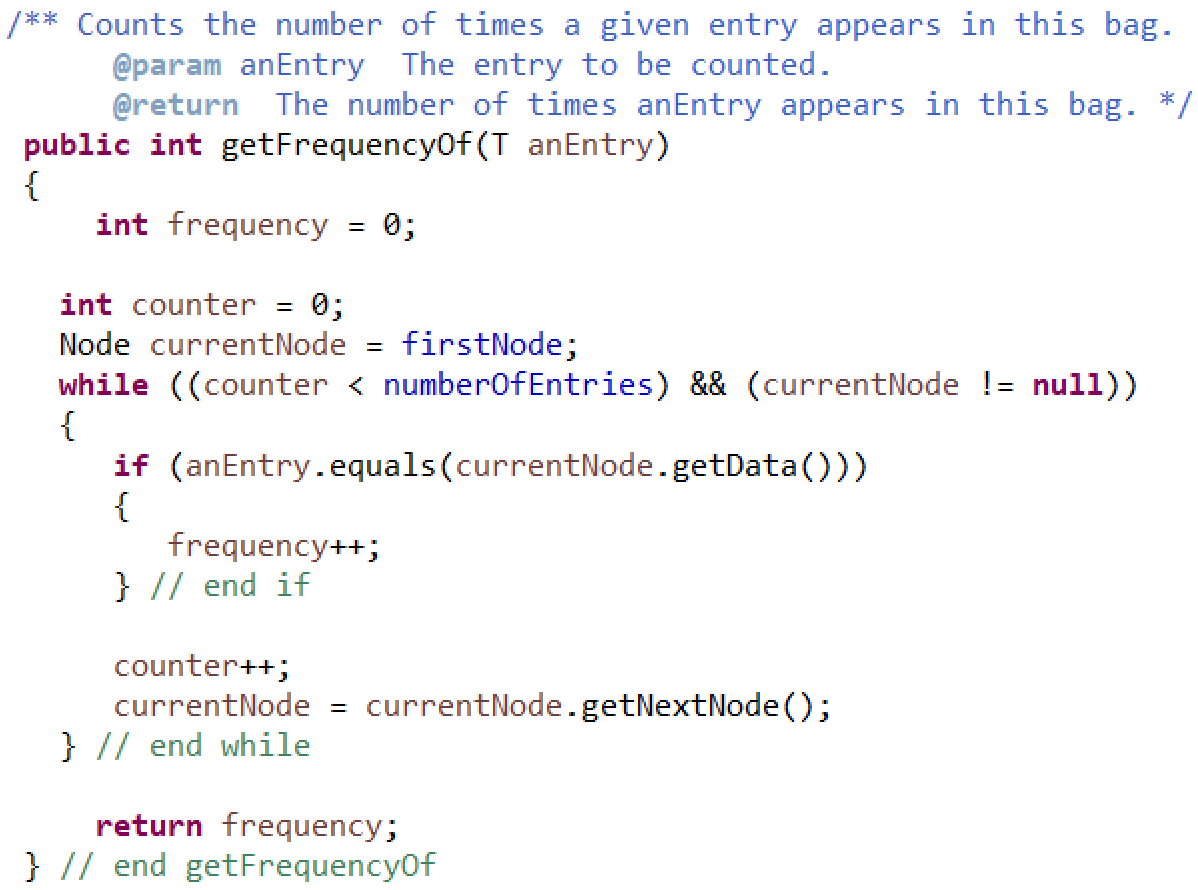
```
/**
   A class of bags whose entries are stored in a chain of linked nodes.
   The bag is never full.
   @author Frank M. Carrano
*/
public class LinkedBag<T> implements BagInterface<T>
{
   private Node firstNode;       // reference to first node
   private int  numberOfEntries;

   public LinkedBag()
   {
      firstNode = null;
      numberOfEntries = 0;
   } // end default constructor

   < Implementations of the public methods declared in BagInterface go here. >

   . . .
```



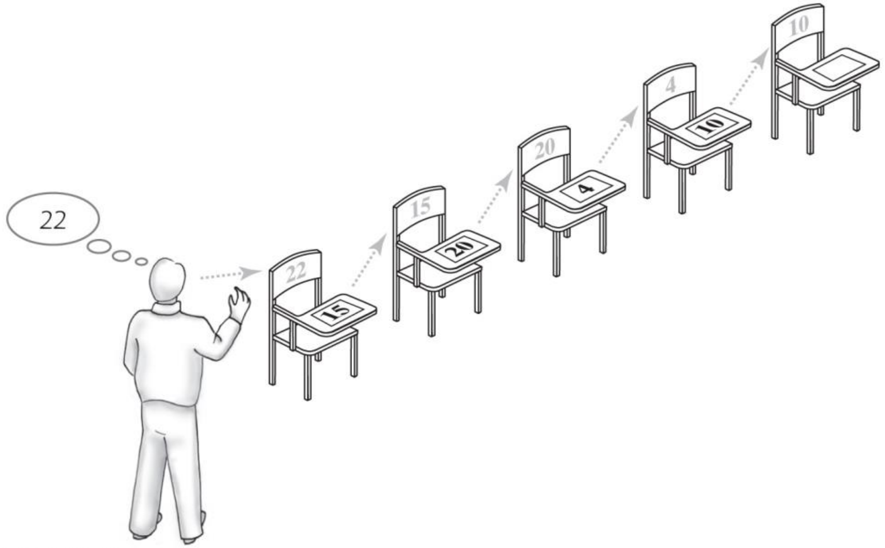
©2019 Pearson Education, Inc.

A Linked Implementation of a Bag

- The method **getFrequencyOf**



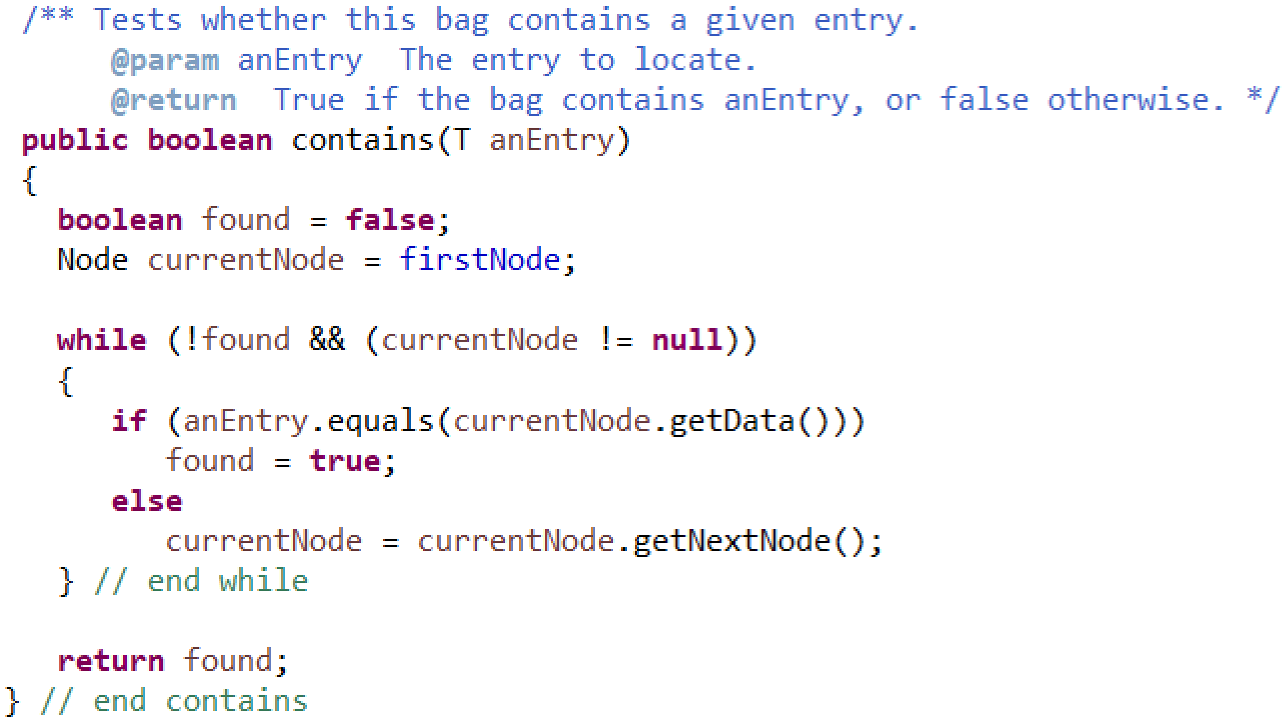
```
/** Counts the number of times a given entry appears in this bag.
      @param anEntry  The entry to be counted.
      @return  The number of times anEntry appears in this bag. */
 public int getFrequencyOf(T anEntry)
 {
     int frequency = 0;

   int counter = 0;
   Node currentNode = firstNode;
   while ((counter < numberOfEntries) && (currentNode != null))
   {
      if (anEntry.equals(currentNode.getData()))
      {
         frequency++;
      } // end if

      counter++;
      currentNode = currentNode.getNextNode();
   } // end while

     return frequency;
 } // end getFrequencyOf
```



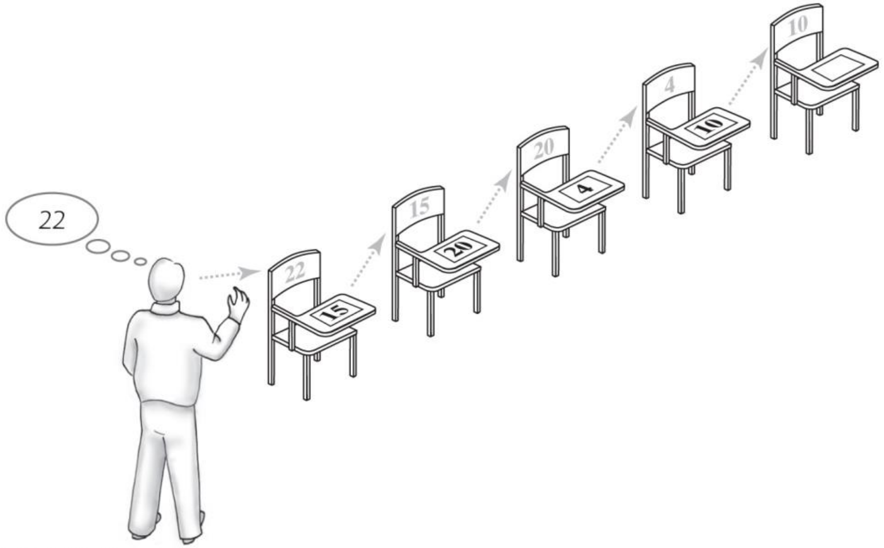
© 2019 Pearson Education, Inc.

A Linked Implementation of a Bag

- The method **contains**



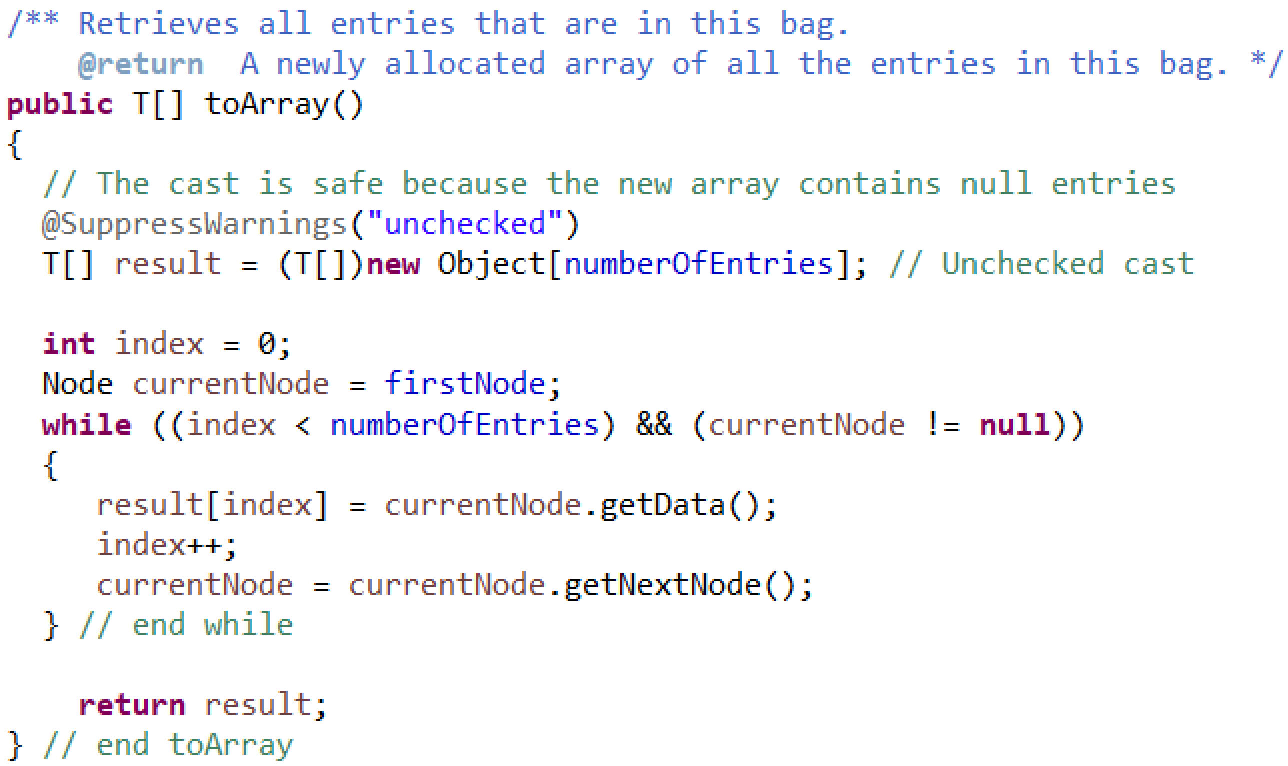
```java
/** Tests whether this bag contains a given entry.
    @param anEntry  The entry to locate.
    @return  True if the bag contains anEntry, or false otherwise. */
public boolean contains(T anEntry)
{
   boolean found = false;
   Node currentNode = firstNode;

   while (!found && (currentNode != null))
   {
      if (anEntry.equals(currentNode.getData()))
         found = true;
      else
         currentNode = currentNode.getNextNode();
   } // end while

   return found;
} // end contains
```



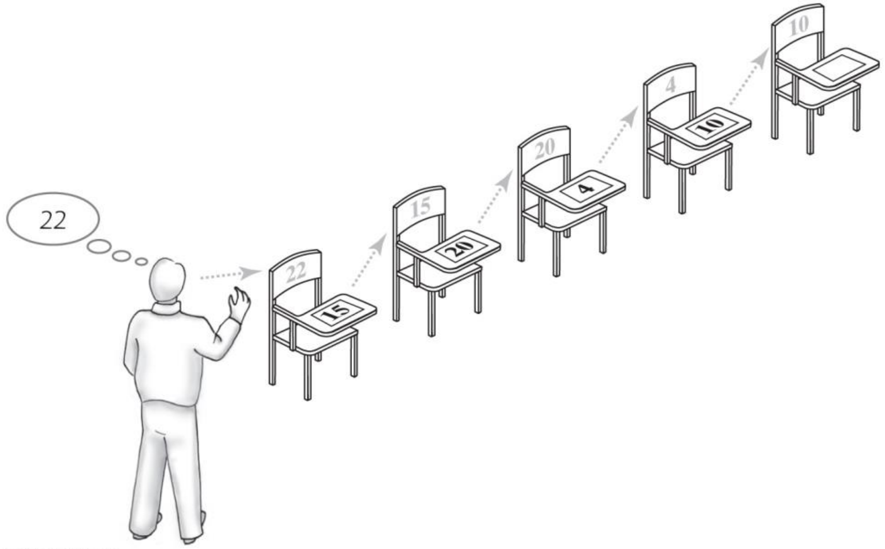
© 2019 Pearson Education, Inc.

A Linked Implementation of a Bag

- The method **toArray**

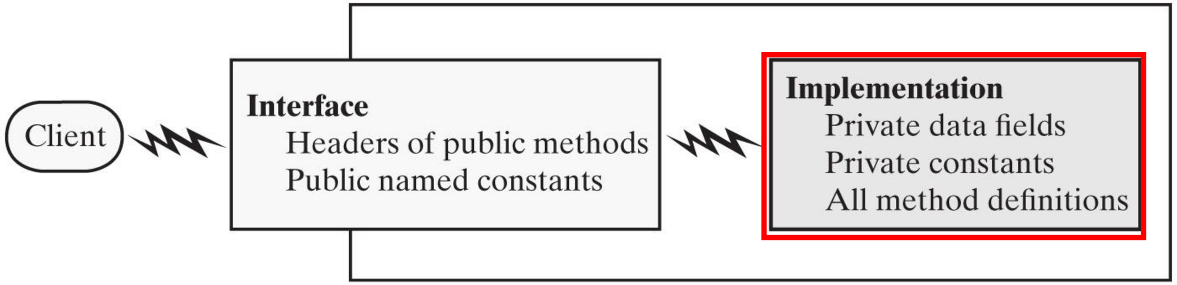


```java
/** Retrieves all entries that are in this bag.
    @return  A newly allocated array of all the entries in this bag. */
public T[] toArray()
{
  // The cast is safe because the new array contains null entries
  @SuppressWarnings("unchecked")
  T[] result = (T[])new Object[numberOfEntries]; //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

  int index = 0;
  Node currentNode = firstNode;
  while ((index < numberOfEntries) && (currentNode != null))
  {
     result[index] = currentNode.getData();
     index++;
     currentNode = currentNode.getNextNode();
  } // end while

    return result;
} // end toArray
```



© 2019 Pearson Education, Inc.

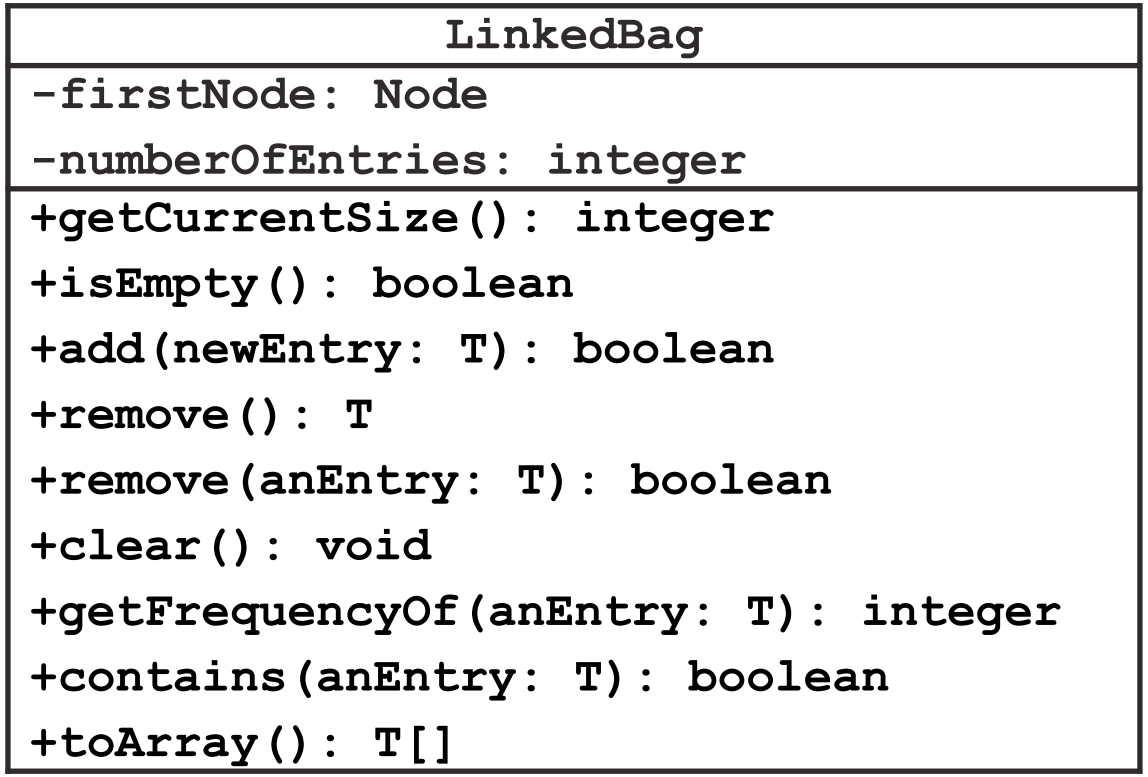
Putting all pieces together to form LinkedBag.java



Client

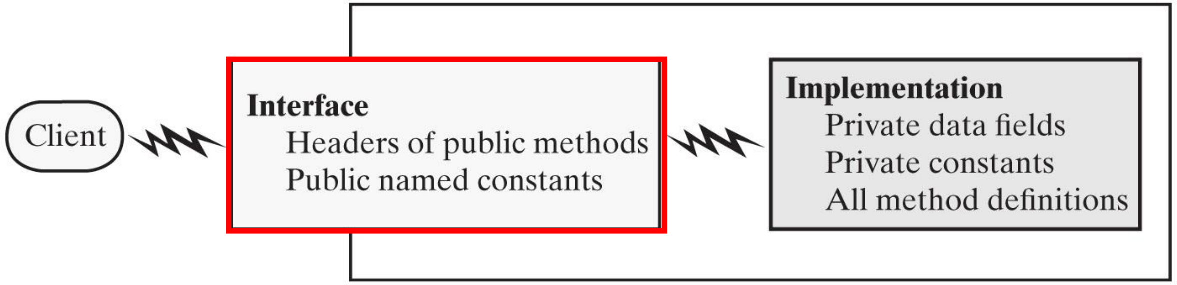
Interface
Headers of public methods
Public named constants

Implementation
Private data fields
Private constants
All method definitions



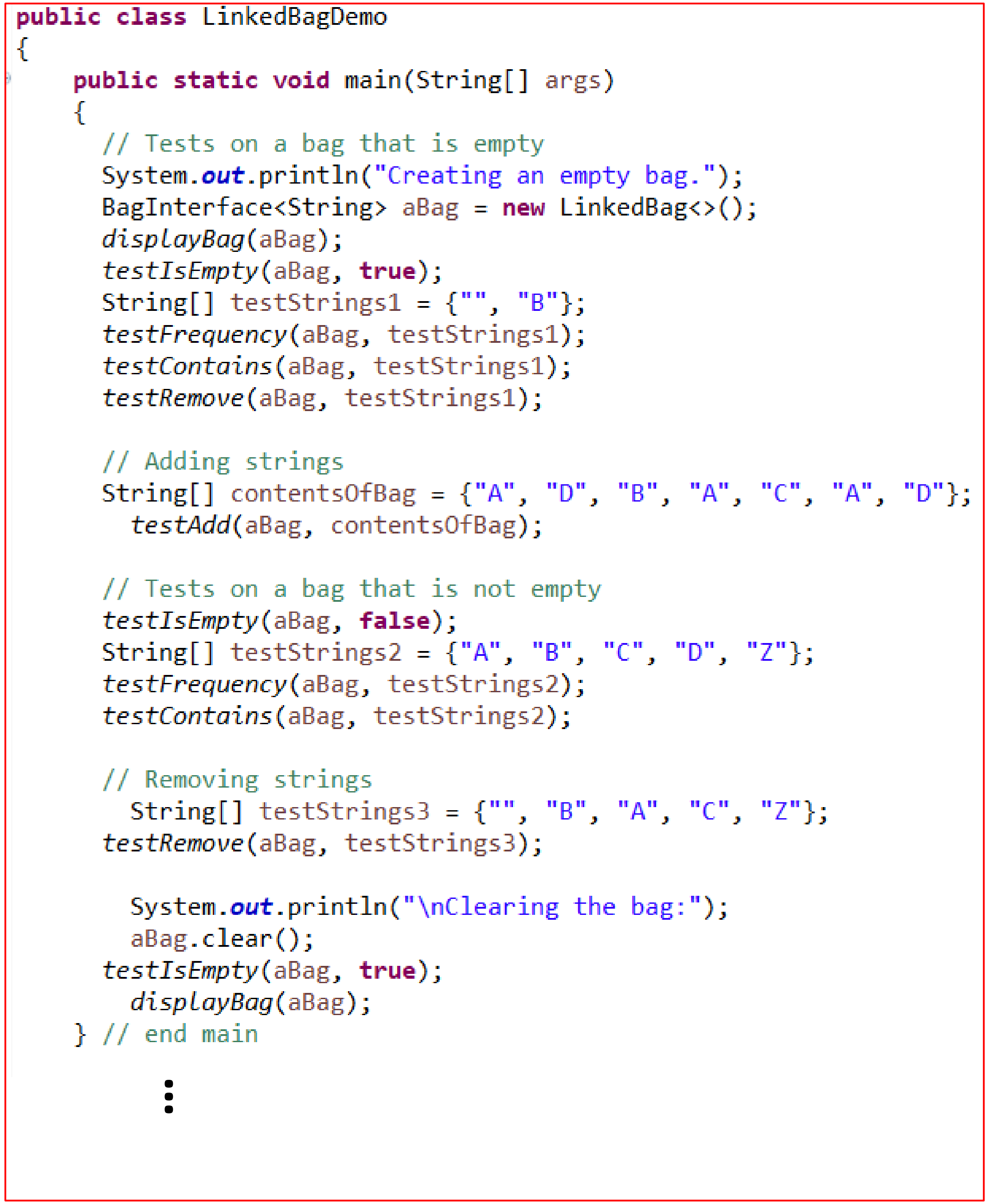
```
LinkedBag
-firstNode: Node
-numberOfEntries: integer
+getCurrentSize(): integer
+isEmpty(): boolean
+add(newEntry: T): boolean
+remove(): T
+remove(anEntry: T): boolean
+clear(): void
+getFrequencyOf(anEntry: T): integer
+contains(anEntry: T): boolean
+toArray(): T[]
```

```java
/** An interface that describes the operations of a bag of objects. */
public interface BagInterface<T>
{
    /** Gets the current number of entries in this bag.
      @return  The integer number of entries currently in the bag. */
    public int getCurrentSize();

    /** Sees whether this bag is empty.
      @return  True if the bag is empty, or false if not. */
    public boolean isEmpty();

    /** 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);

    /** Removes one unspecified entry from this bag, if possible.
      @return  Either the removed entry, if the removal was successful, or null. */
    public T remove();

    /** Removes one occurrence of a given entry from this bag, if possible.
      @param anEntry  The entry to be removed.
      @return  True if the removal was successful, or false if not. */
   public boolean remove(T anEntry);

    /** Removes all entries from this bag. */
    public void clear();

    /** 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);

    /** Tests whether this bag contains a given entry.
       @param anEntry  The entry to find.
       @return  True if the bag contains anEntry, or false if not. */
    public boolean contains(T anEntry);

    /** Retrieves all entries that are in this bag.
       @return  A newly allocated array of all the entries in the bag. Note: If the bag is empty, the
    returned array is empty. */
    public T[] toArray();

} // end BagInterface
```

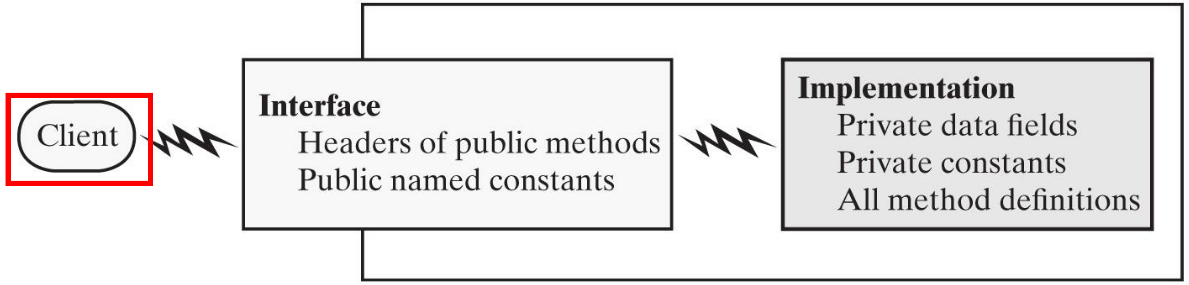


© 2019 Pearson Education, Inc.

BagInterface.java



```
public class LinkedBagDemo
{
    public static void main(String[] args)
    {
      // Tests on a bag that is empty
      System.out.println("Creating an empty bag.");
      BagInterface<String> aBag = new LinkedBag<>();
      displayBag(aBag);
      testIsEmpty(aBag, true);
      String[] testStrings1 = {"", "B"};
      testFrequency(aBag, testStrings1);
      testContains(aBag, testStrings1);
      testRemove(aBag, testStrings1);

      // Adding strings
      String[] contentsOfBag = {"A", "D", "B", "A", "C", "A", "D"};
        testAdd(aBag, contentsOfBag);

      // Tests on a bag that is not empty
      testIsEmpty(aBag, false);
      String[] testStrings2 = {"A", "B", "C", "D", "Z"};
      testFrequency(aBag, testStrings2);
      testContains(aBag, testStrings2);

      // Removing strings
        String[] testStrings3 = {"", "B", "A", "C", "Z"};
      testRemove(aBag, testStrings3);

        System.out.println("\nClearing the bag:");
        aBag.clear();
      testIsEmpty(aBag, true);
        displayBag(aBag);
    } // end main

          ⋮
```



© 2019 Pearson Education, Inc.

LinkedBagDemo.java

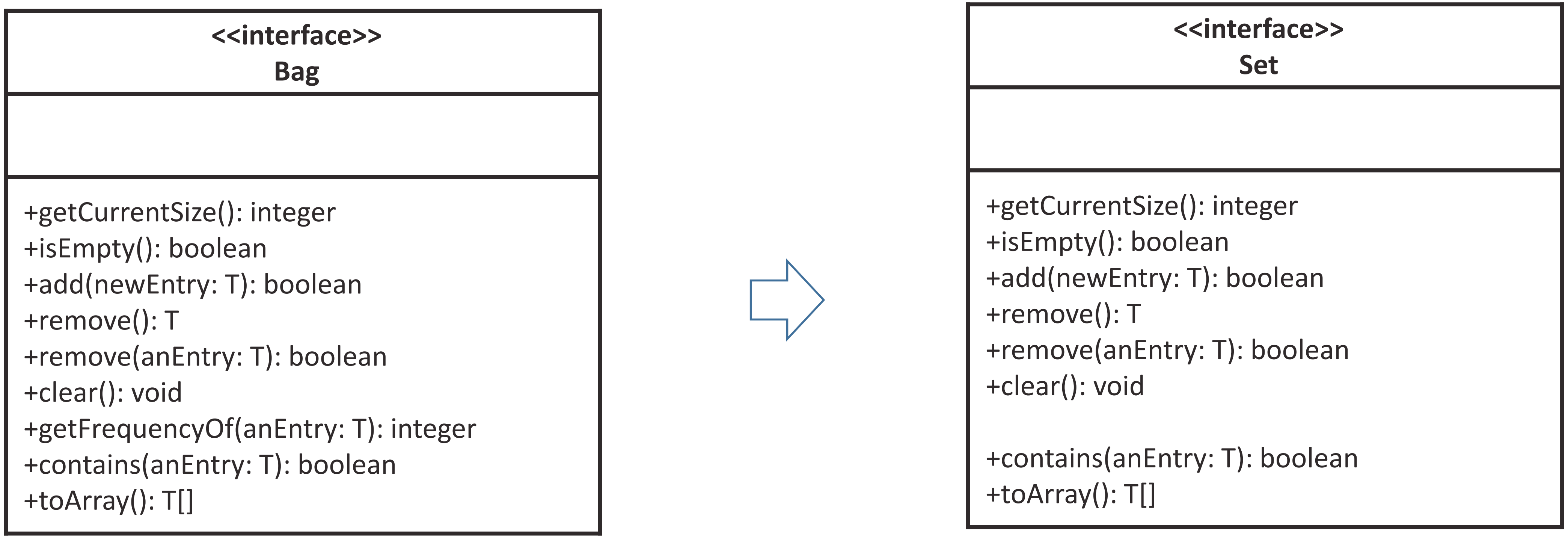
Pros and Cons of Using a Chain

- +Bag can grow and shrink in size as necessary.
- +Remove and recycle nodes that are no longer needed
- +Adding new entry to end of array or to beginning of chain both relatively simple
- +Similar for removal

- -Removing specific entry requires search of chain
- -Chain requires more memory than array of same length

The ADT Set

- A set is a special kind of bag, one that does not allow duplicate entries



<<interface>>
Bag

+getCurrentSize(): integer
+isEmpty(): boolean
+add(newEntry: T): boolean
+remove(): T
+remove(anEntry: T): boolean
+clear(): void
+getFrequencyOf(anEntry: T): integer
+contains(anEntry: T): boolean
+toArray(): T[]

<<interface>>
Set

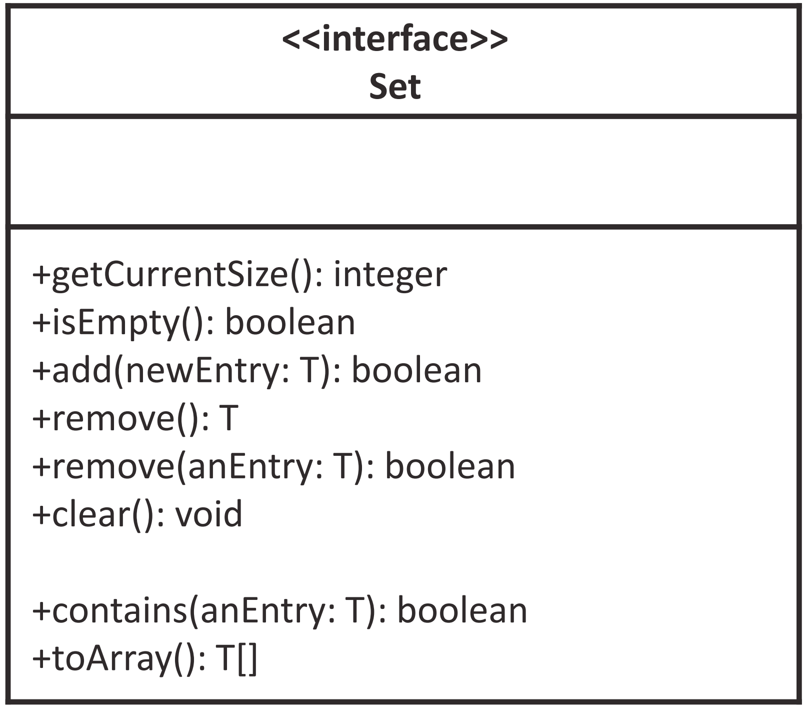
+getCurrentSize(): integer
+isEmpty(): boolean
+add(newEntry: T): boolean
+remove(): T
+remove(anEntry: T): boolean
+clear(): void

+contains(anEntry: T): boolean
+toArray(): T[]

The ADT Set

- A set is a special kind of bag, one that does not allow duplicate entries

```
/** An interface that describes the operations of a set of objects. */
public interface SetInterface<T>
{
        public int getCurrentSize();
        public boolean isEmpty();

        /** Adds a new entry to this set, avoiding duplicates.
            @param newEntry  The object to be added as a new entry.
            @return  True if the addition is successful, or
              false if the item already is in the set. */
        public boolean add(T newEntry);

        /** Removes a specific entry from this set, if possible.
      @param anEntry  The entry to be removed.
      @return  True if the removal was successful, or false if not. */
        public boolean remove(T anEntry);

        public T remove();
        public void clear();
        public boolean contains(T anEntry);
        public T[] toArray();
} // end SetInterface
```

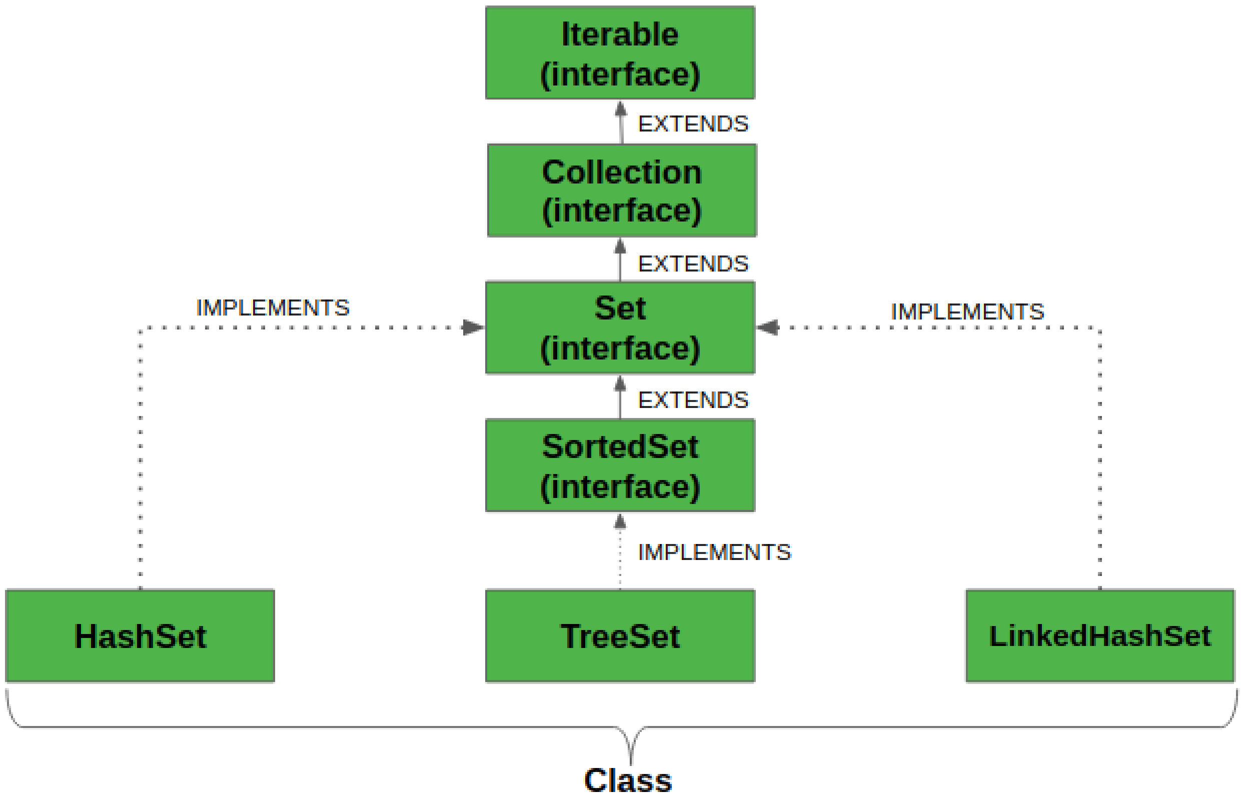


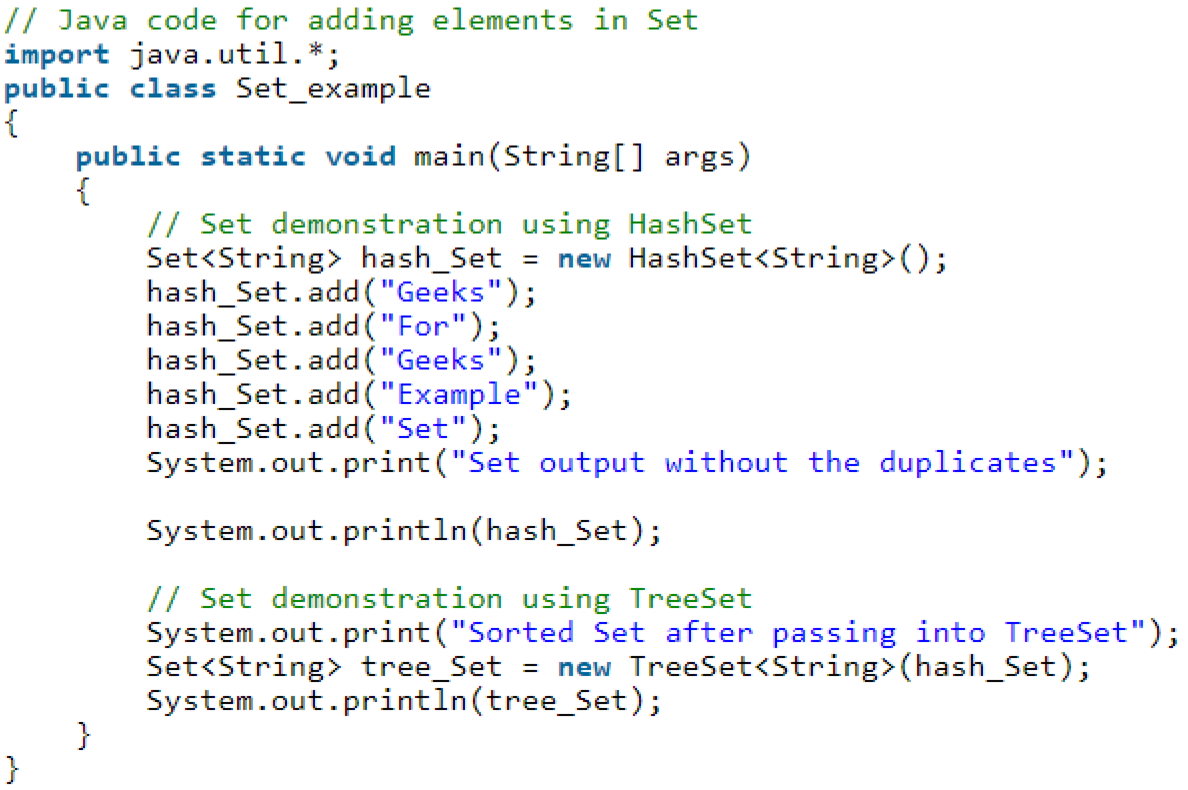
<<interface>>
Set

+getCurrentSize(): integer
+isEmpty(): boolean
+add(newEntry: T): boolean
+remove(): T
+remove(anEntry: T): boolean
+clear(): void

+contains(anEntry: T): boolean
+toArray(): T[]

Java Class Library: The Interface Set

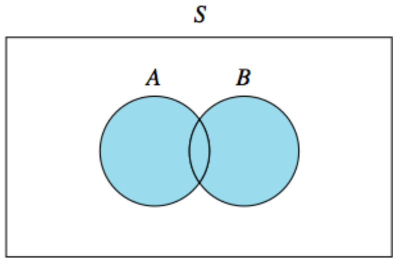




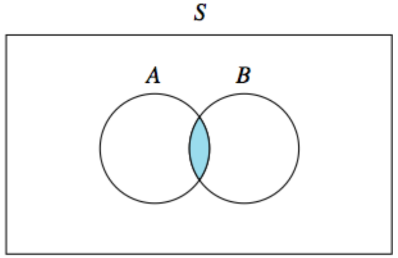
```
// Java code for adding elements in Set
import java.util.*;
public class Set_example
{
    public static void main(String[] args)
    {
        // Set demonstration using HashSet
        Set<String> hash_Set = new HashSet<String>();
        hash_Set.add("Geeks");
        hash_Set.add("For");
        hash_Set.add("Geeks");
        hash_Set.add("Example");
        hash_Set.add("Set");
        System.out.print("Set output without the duplicates");

        System.out.println(hash_Set);

        // Set demonstration using TreeSet
        System.out.print("Sorted Set after passing into TreeSet");
        Set<String> tree_Set = new TreeSet<String>(hash_Set);
        System.out.println(tree_Set);
    }
}
```

https://www.geeksforgeeks.org/set-in-java/
https://www.programcreek.com/2013/03/hashset-vs-treeset-vs-linkedhashset/

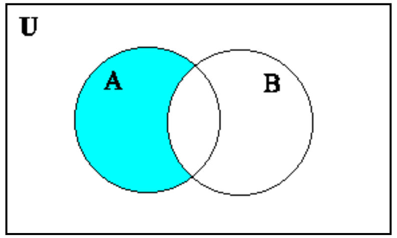
Java Class Library: The Interface Set

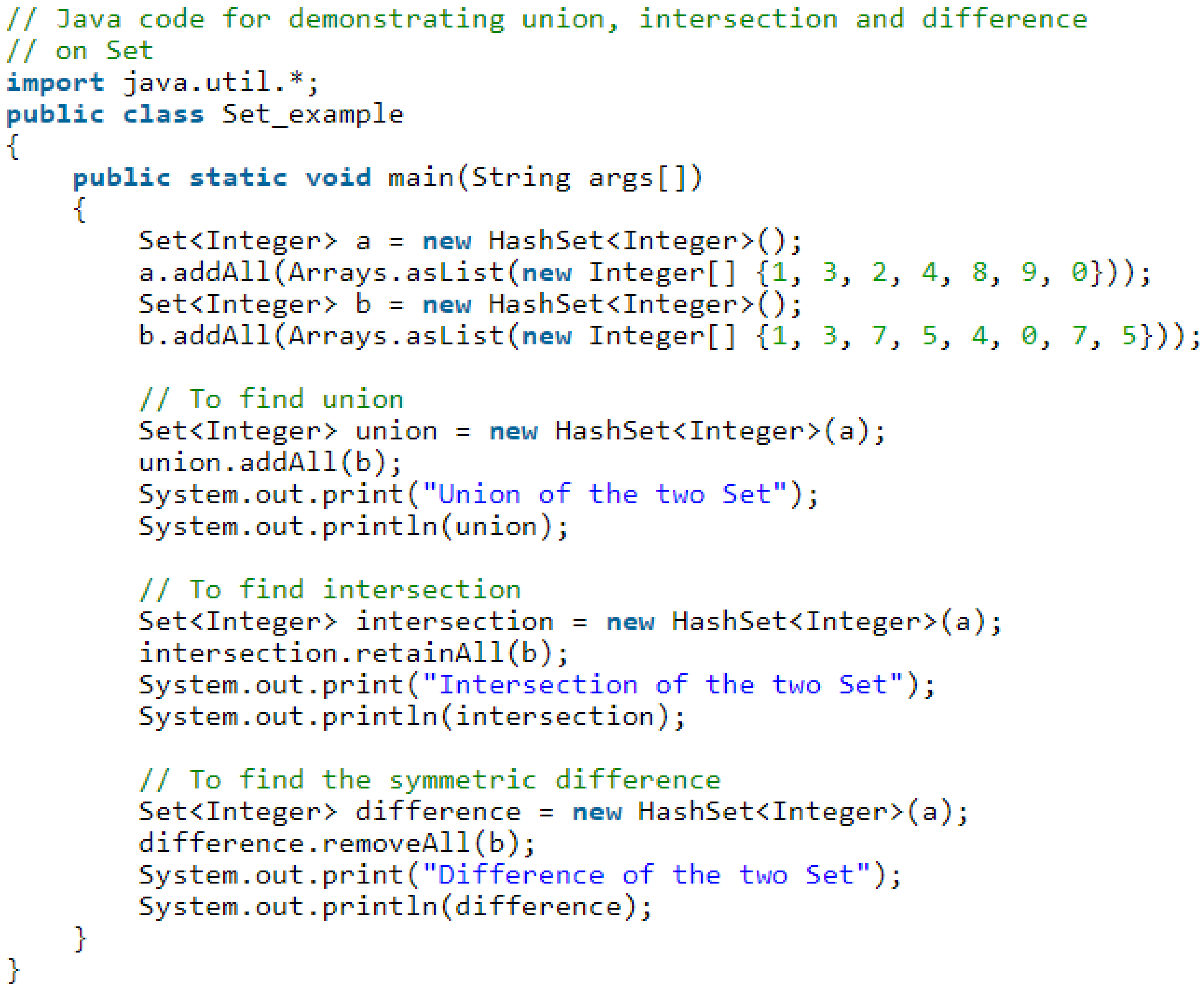


A ∪ B



A ∩ B





```
// Java code for demonstrating union, intersection and difference
// on Set
import java.util.*;
public class Set_example
{
    public static void main(String args[])
    {
        Set<Integer> a = new HashSet<Integer>();
        a.addAll(Arrays.asList(new Integer[] {1, 3, 2, 4, 8, 9, 0}));
        Set<Integer> b = new HashSet<Integer>();
        b.addAll(Arrays.asList(new Integer[] {1, 3, 7, 5, 4, 0, 7, 5}));

        // To find union
        Set<Integer> union = new HashSet<Integer>(a);
        union.addAll(b);
        System.out.print("Union of the two Set");
        System.out.println(union);

        // To find intersection
        Set<Integer> intersection = new HashSet<Integer>(a);
        intersection.retainAll(b);
        System.out.print("Intersection of the two Set");
        System.out.println(intersection);

        // To find the symmetric difference
        Set<Integer> difference = new HashSet<Integer>(a);
        difference.removeAll(b);
        System.out.print("Difference of the two Set");
        System.out.println(difference);
    }
}
```

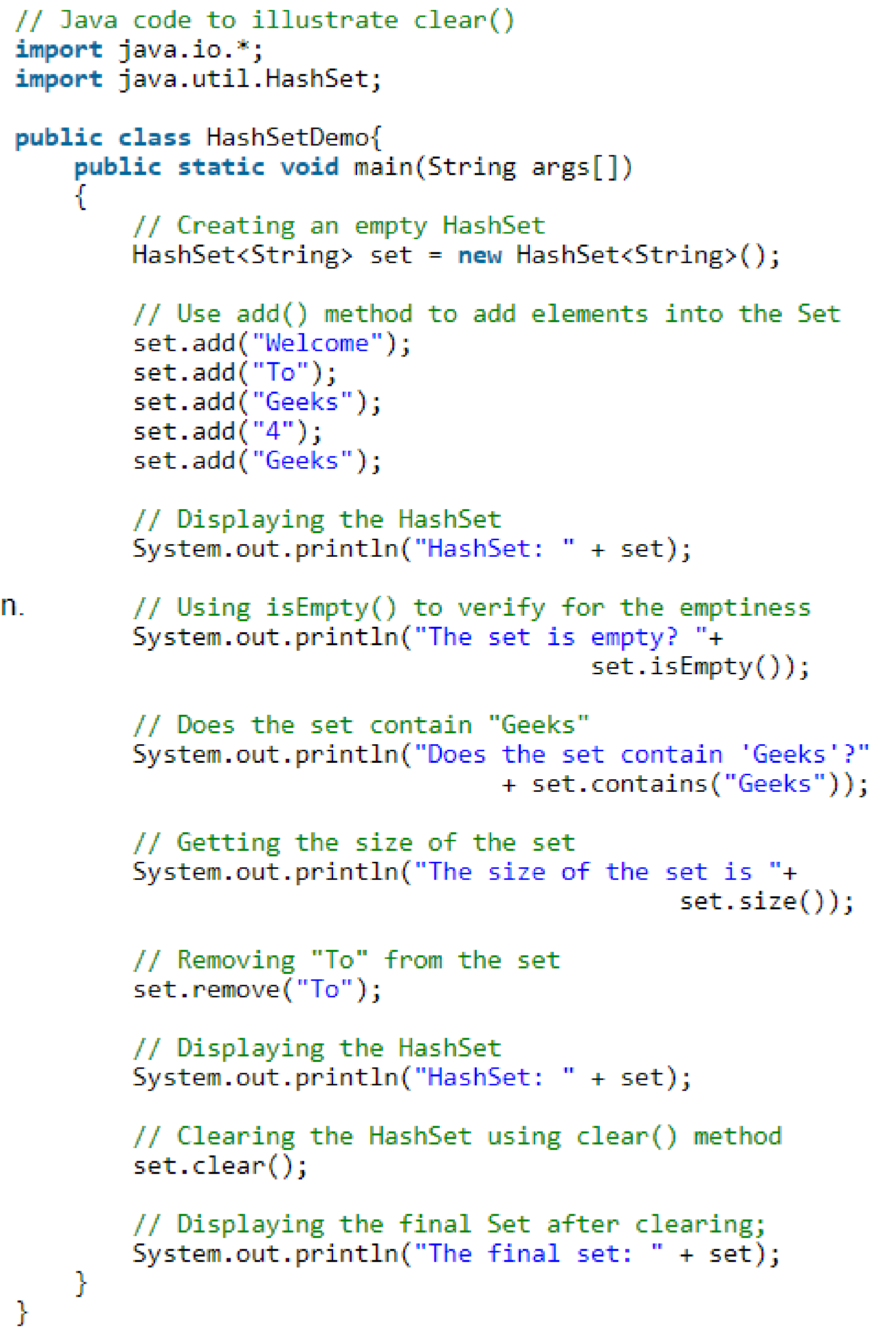
https://www.geeksforgeeks.org/set-in-java/

The Interface Set

Methods in Set Interface:

1. **add():** This method is used to add one object to the collection at a time.
2. **clear():** This method is used to remove all elements from the collection.
3. **contains():** This method is used to verify whether a specified element is present in the collection or not.
4. **isEmpty():** This method is used to check whether the collection is empty or not.
5. **iterator():** This is used to return an Iterator object, which may be used to retrieve an object from the collection.
6. **remove():** This is used to removes a specified object from the collection.
7. **size():** This is used to know the size or the number of elements present in the collection.

Readings: https://www.geeksforgeeks.org/set-in-java/



```
// Java code to illustrate clear()
import java.io.*;
import java.util.HashSet;

public class HashSetDemo{
    public static void main(String args[])
    {
        // Creating an empty HashSet
        HashSet<String> set = new HashSet<String>();

        // Use add() method to add elements into the Set
        set.add("Welcome");
        set.add("To");
        set.add("Geeks");
        set.add("4");
        set.add("Geeks");

        // Displaying the HashSet
        System.out.println("HashSet: " + set);

        // Using isEmpty() to verify for the emptiness
        System.out.println("The set is empty? "+
                                set.isEmpty());

        // Does the set contain "Geeks"
        System.out.println("Does the set contain 'Geeks'?"
                                + set.contains("Geeks"));

        // Getting the size of the set
        System.out.println("The size of the set is "+
                                      set.size());

        // Removing "To" from the set
        set.remove("To");

        // Displaying the HashSet
        System.out.println("HashSet: " + set);

        // Clearing the HashSet using clear() method
        set.clear();

        // Displaying the final Set after clearing;
        System.out.println("The final set: " + set);
    }
}
```

Summary

- ADT Bag
- Implementations of a Bag

What I Want You to Do

- Review class slides
- Review Chapter 1, Chapter 2, and Chapter 3

- Next Topic
  - Efficiency of Algorithms