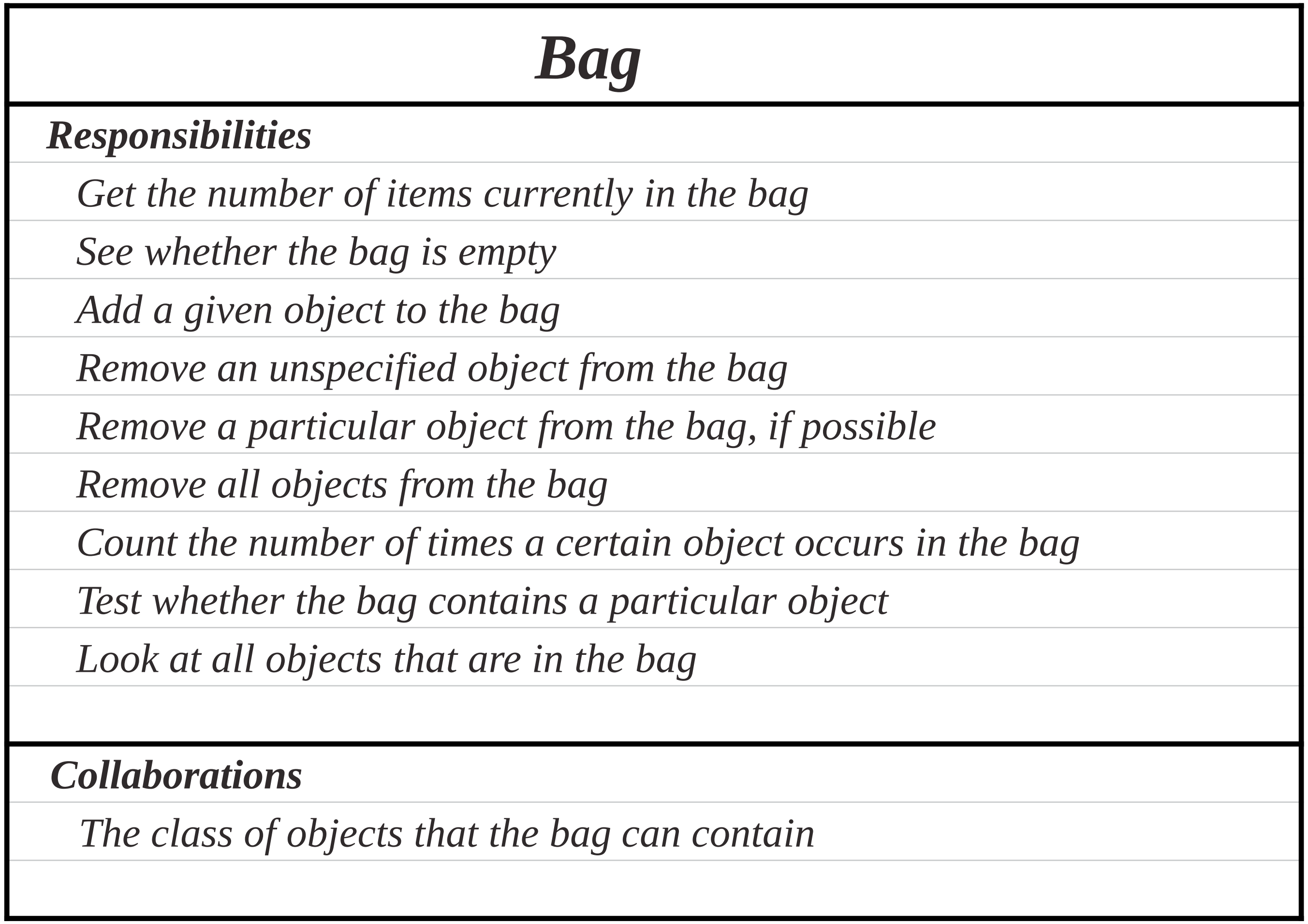
Class 02 – ADTs, Bags, and Algorithm Efficiency

CSIS 3475 Data Structures and Algorithms

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, remove objects

CRC Card



Bag

Responsibilities

Get the number of items currently in the bag

See whether the bag is empty

Add a given object to the bag

Remove an unspecified object from the bag

Remove a particular object from the bag, if possible

Remove all objects from the bag

Count the number of times a certain object occurs in the bag

Test whether the bag contains a particular object

Look at all objects that are in the bag

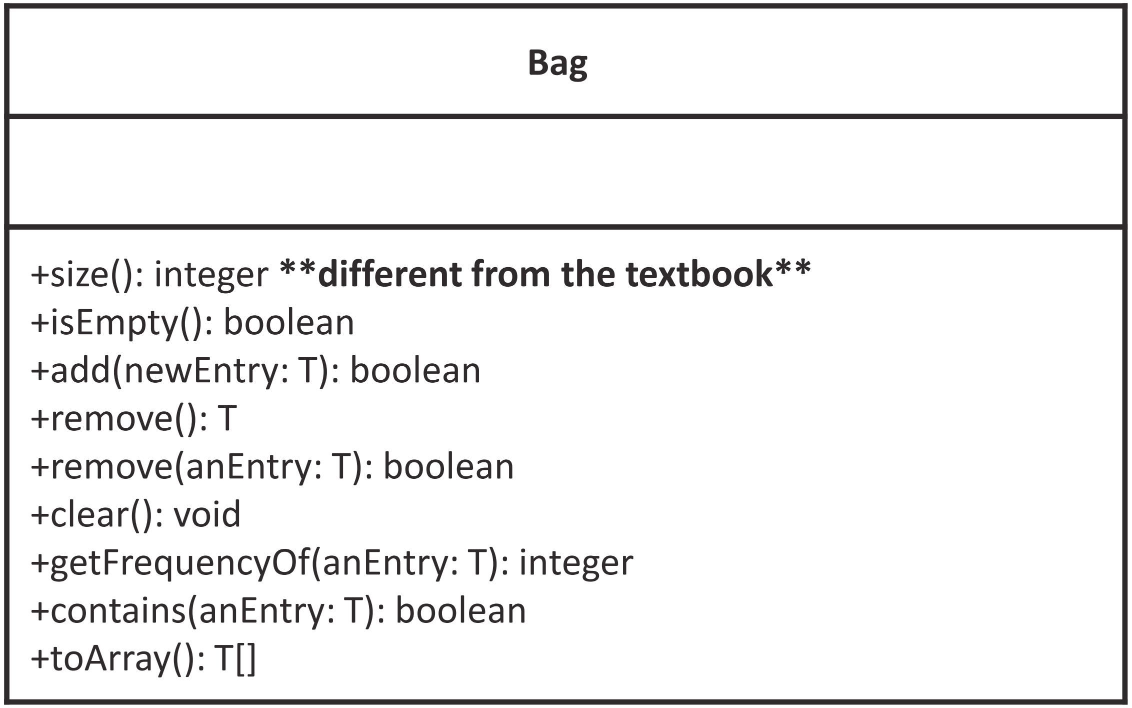
Collaborations

The class of objects that the bag can contain

Specifying a Bag

- Describe its data and specify in detail the methods
- Options that we can take when add cannot complete its task:
    - Do nothing
    - Leave bag unchanged, but signal client
- Note which methods change the object or do not

Using UML Notation to Specify a Class



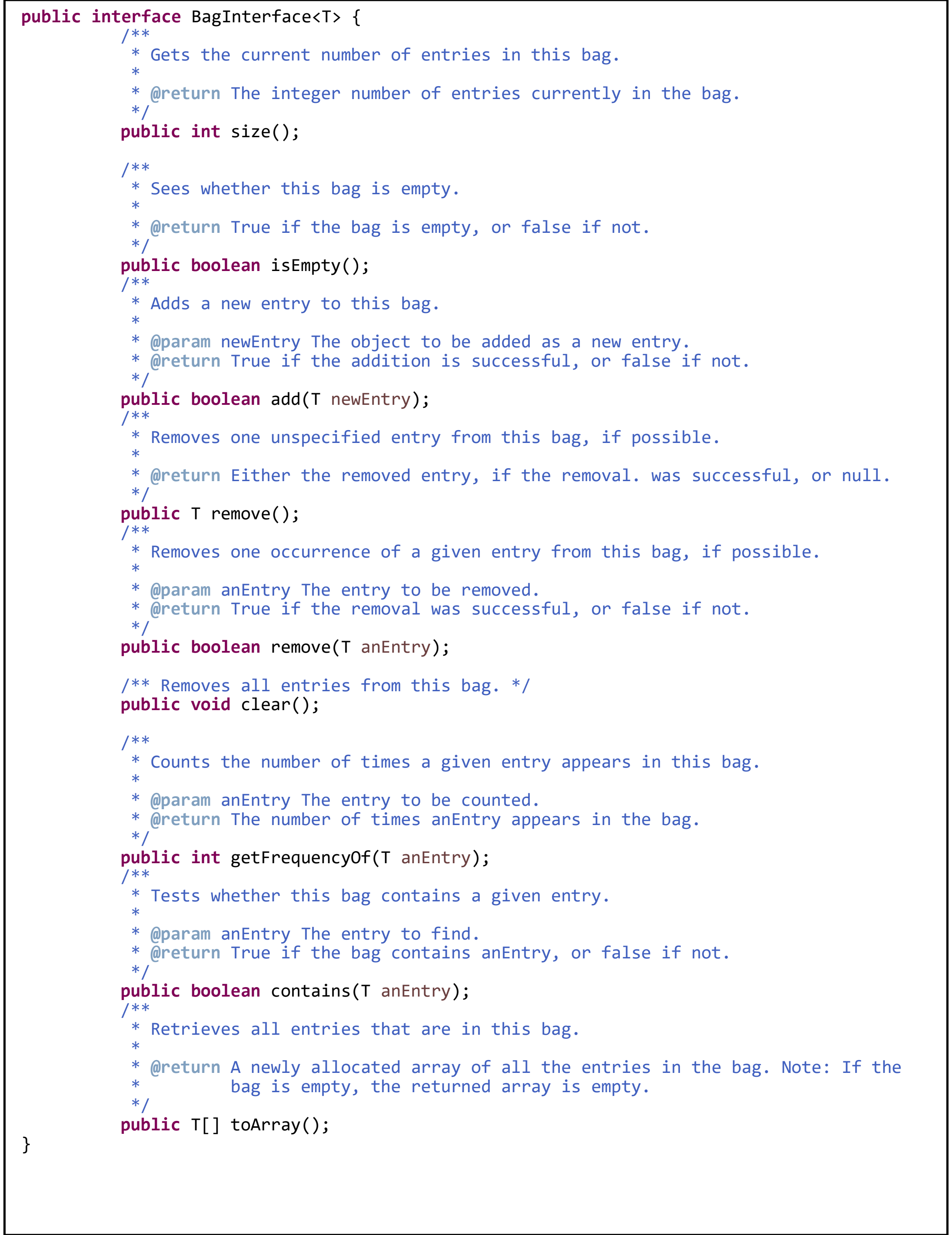
Bag

+size(): integer ****different from the textbook****
+isEmpty(): boolean
+add(newEntry: T): boolean
+remove(): T
+remove(anEntry: T): boolean
+clear(): void
+getFrequencyOf(anEntry: T): integer
+contains(anEntry: T): boolean
+toArray(): T[]

Design Decision

- What to do for unusual conditions?
- Assume it won’t happen
- Ignore invalid situations
- Guess at the client’s intention
- Return value that signals a problem
- Return a boolean
- Throw an exception

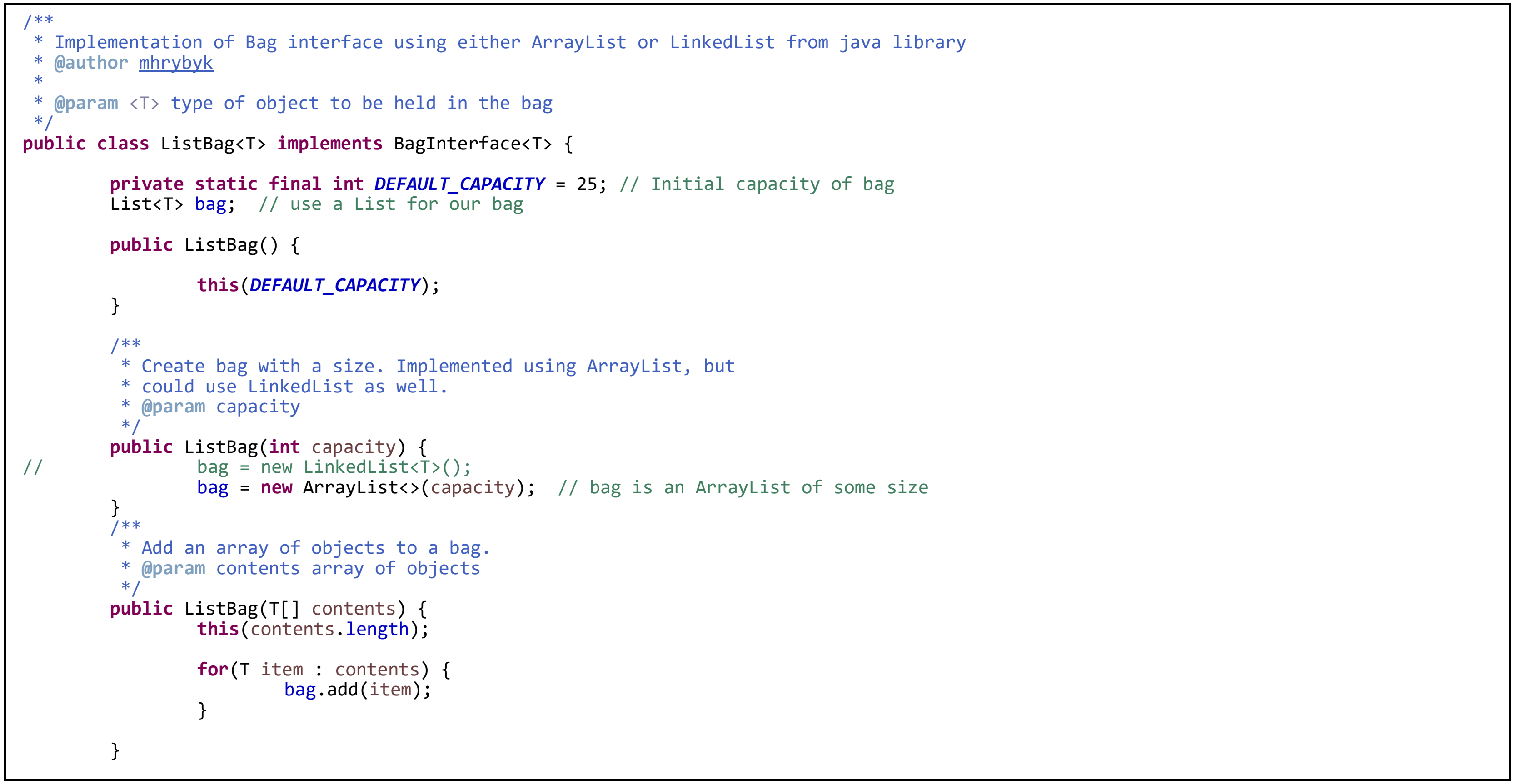
Bag Interface



```
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 size();

          /**
           * 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();
}
```

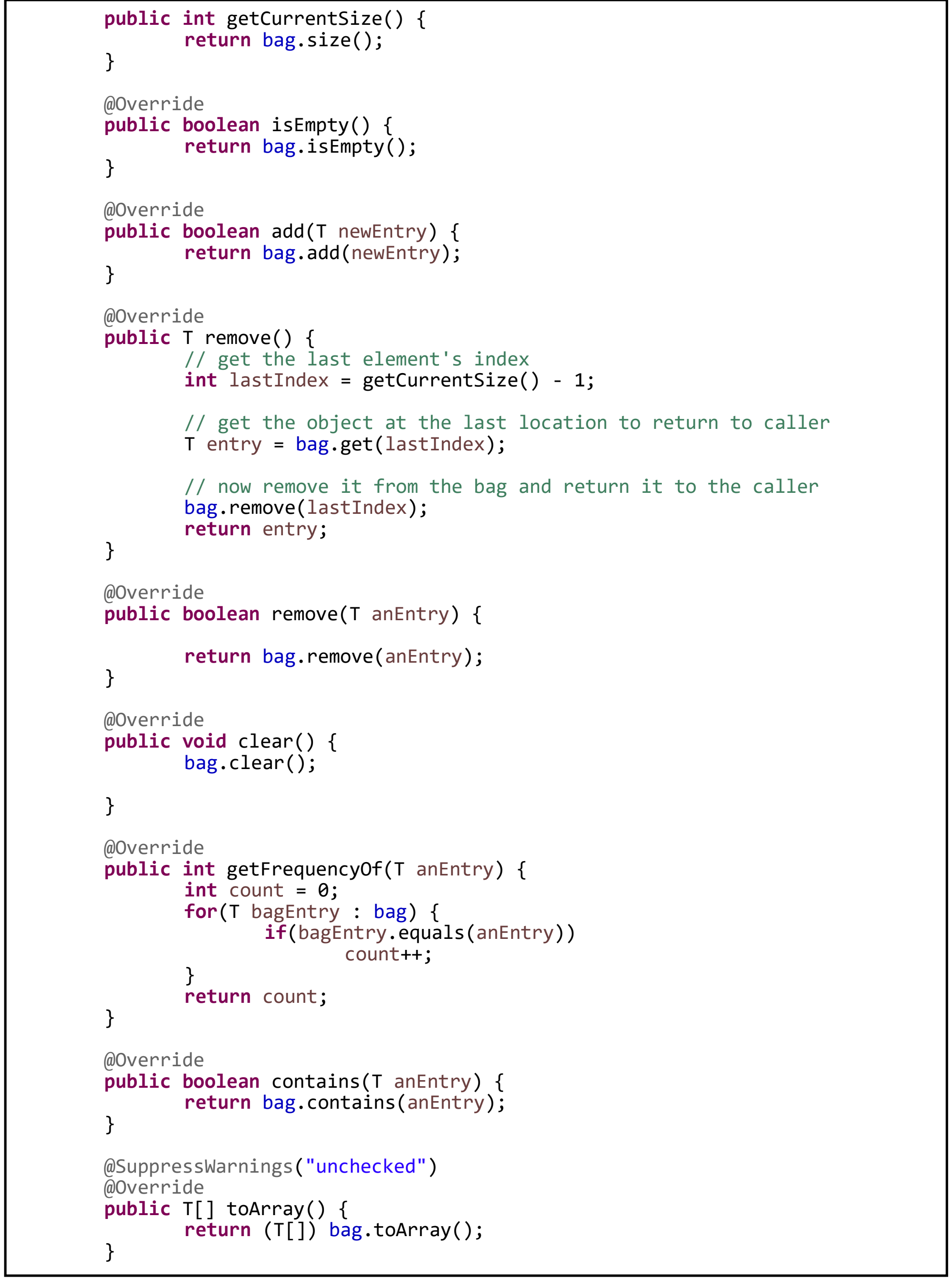
Implementing a Bag using Java Library

- LinkedBag uses an internal Java List
- List is an interface.
- Use ArrayList or LinkedList



```java
/**
 * Implementation of Bag interface using either ArrayList or LinkedList from java library
 * @author mhrybyk
 *
 * @param <T> type of object to be held in the bag
 */
public class ListBag<T> implements BagInterface<T> {

        private static final int DEFAULT_CAPACITY = 25; // Initial capacity of bag
        List<T> bag;  // use a List for our bag

        public ListBag() {

                this(DEFAULT_CAPACITY);
        }

        /**
         * Create bag with a size. Implemented using ArrayList, but
         * could use LinkedList as well.
         * @param capacity
         */
        public ListBag(int capacity) {
//              bag = new LinkedList<T>();
                bag = new ArrayList<>(capacity);  // bag is an ArrayList of some size
        }
        /**
         * Add an array of objects to a bag.
         * @param contents array of objects
         */
        public ListBag(T[] contents) {
                this(contents.length);

                for(T item : contents) {
                        bag.add(item);
                }

        }
```

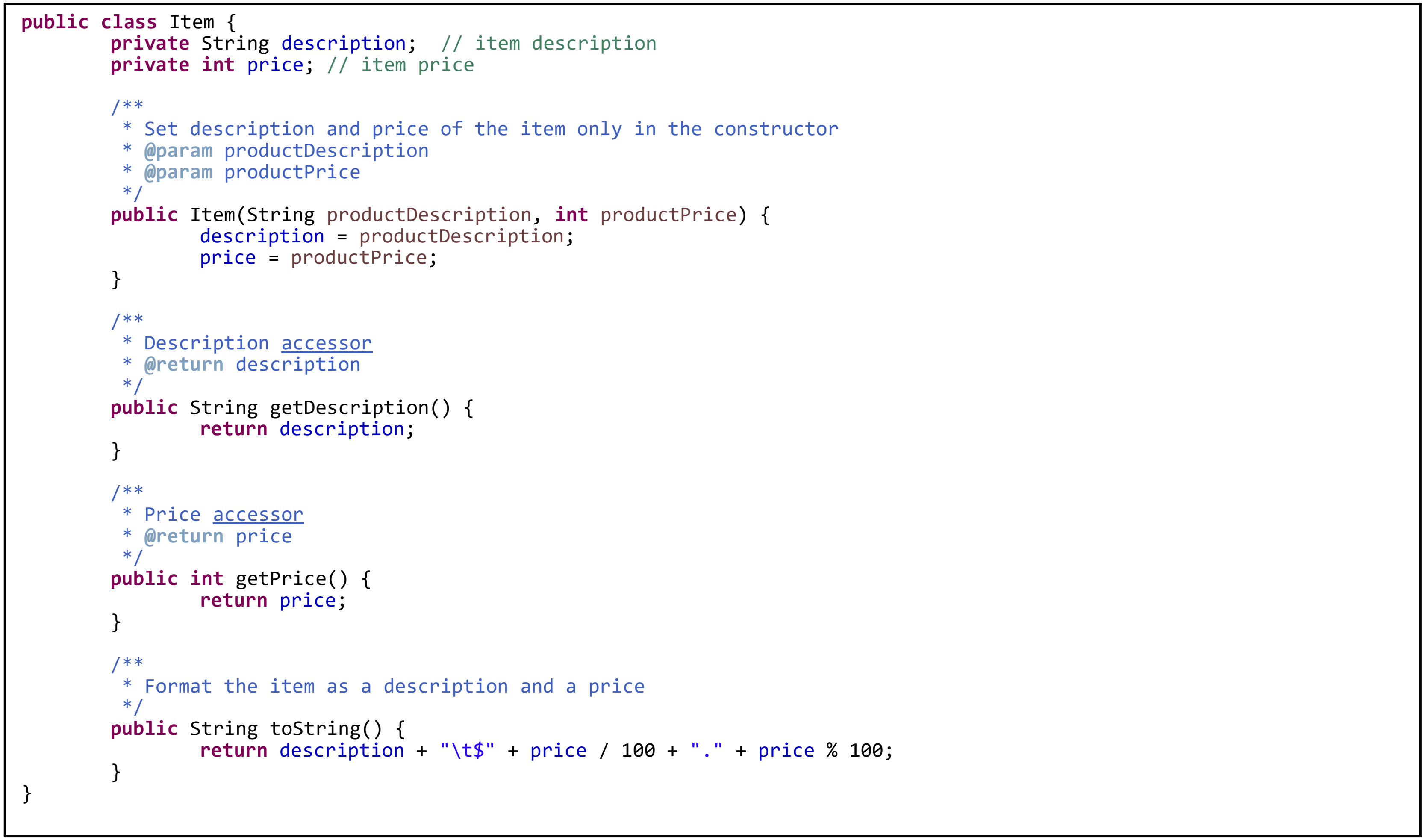
LinkedBag methods



```java
public int getCurrentSize() {
    return bag.size();
}

@Override
public boolean isEmpty() {
    return bag.isEmpty();
}

@Override
public boolean add(T newEntry) {
    return bag.add(newEntry);
}

@Override
public T remove() {
    // get the last element's index
    int lastIndex = getCurrentSize() - 1;

    // get the object at the last location to return to caller
    T entry = bag.get(lastIndex);

    // now remove it from the bag and return it to the caller
    bag.remove(lastIndex);
    return entry;
}

@Override
public boolean remove(T anEntry) {

    return bag.remove(anEntry);
}

@Override
public void clear() {
    bag.clear();

}

@Override
public int getFrequencyOf(T anEntry) {
    int count = 0;
    for(T bagEntry : bag) {
        if(bagEntry.equals(anEntry))
            count++;
    }
    return count;
}

@Override
public boolean contains(T anEntry) {
    return bag.contains(anEntry);
}

@SuppressWarnings("unchecked")
@Override
public T[] toArray() {
    return (T[]) bag.toArray();
}
```

Use of Bag – Online Shopper

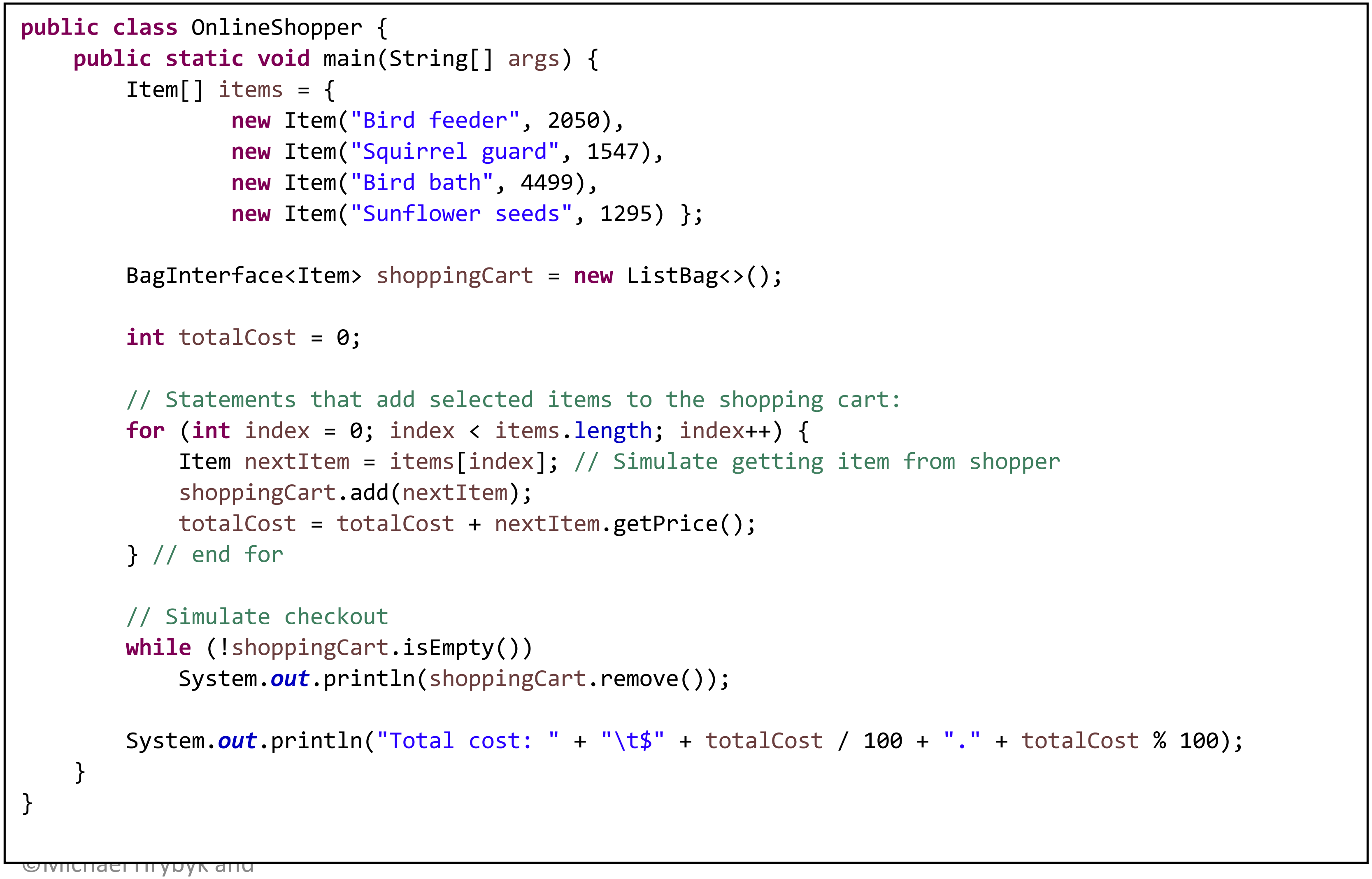
- Purchase an item (see class definition below)
- Put it in a Bag
- Check out by removing them from the bag and adding up the prices



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

        /**
         * Set description and price of the item only in the constructor
         * @param productDescription
         * @param productPrice
         */
        public Item(String productDescription, int productPrice) {
                description = productDescription;
                price = productPrice;
        }

        /**
         * Description accessor
         * @return description
         */
        public String getDescription() {
                return description;
        }

        /**
         * Price accessor
         * @return price
         */
        public int getPrice() {
                return price;
        }

        /**
         * Format the item as a description and a price
         */
        public String toString() {
                return description + "\t$" + price / 100 + "." + price % 100;
        }
}
```

OnLine Shopper

- Put items in the bag – use of ListBag class
- Checkout, removing them



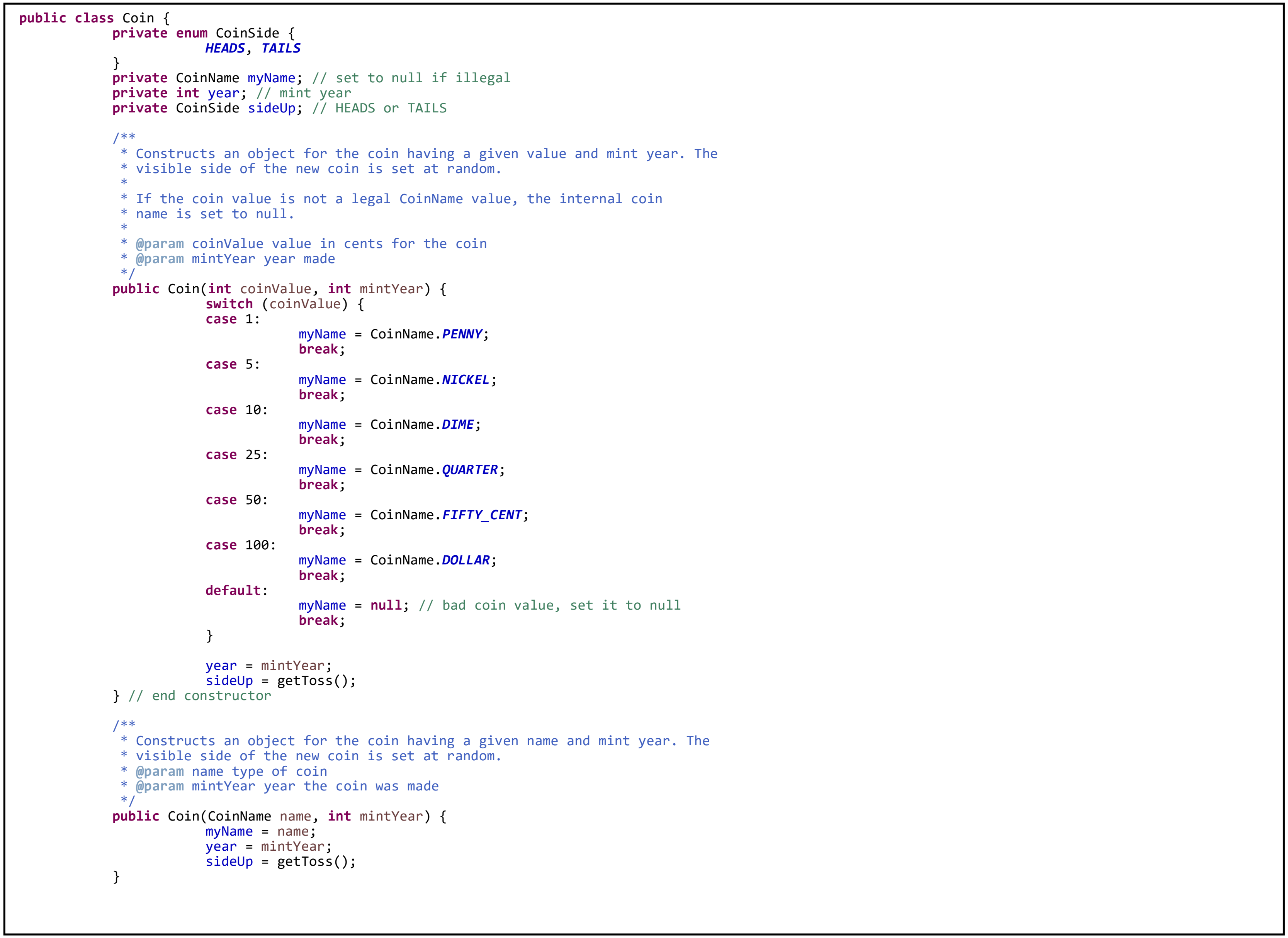
```java
public class OnlineShopper {
    public static void main(String[] args) {
        Item[] items = {
                new Item("Bird feeder", 2050),
                new Item("Squirrel guard", 1547),
                new Item("Bird bath", 4499),
                new Item("Sunflower seeds", 1295) };

        BagInterface<Item> shoppingCart = new ListBag<>();

        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);
    }
}
```

PiggyBank as a Bag – Coin and CoinName

- Add coins to a PiggyBank, then remove them
- Need a Coin and CoinName class
- CoinName is an enum with standard values
  - Note use of enum constructor – not public! – and arg that corresponds to enum value

```
public enum CoinName {
     PENNY(1), NICKEL(5), DIME(10), QUARTER(25), FIFTY_CENT(50), DOLLAR(100);

     private int coinValue = 0;

     CoinName(int value) {
          coinValue = value;
     }

     public int getValue( ) {
          return coinValue;
     }
}
```

Coin constructor



```
public class Coin {
            private enum CoinSide {
                        HEADS, TAILS
            }
            private CoinName myName; // set to null if illegal
            private int year; // mint year
            private CoinSide sideUp; // HEADS or TAILS

            /**
             * Constructs an object for the coin having a given value and mint year. The
             * visible side of the new coin is set at random.
             *
             * If the coin value is not a legal CoinName value, the internal coin
             * name is set to null.
             *
             * @param coinValue value in cents for the coin
             * @param mintYear year made
             */
            public Coin(int coinValue, int mintYear) {
                        switch (coinValue) {
                        case 1:
                                    myName = CoinName.PENNY;
                                    break;
                        case 5:
                                    myName = CoinName.NICKEL;
                                    break;
                        case 10:
                                    myName = CoinName.DIME;
                                    break;
                        case 25:
                                    myName = CoinName.QUARTER;
                                    break;
                        case 50:
                                    myName = CoinName.FIFTY_CENT;
                                    break;
                        case 100:
                                    myName = CoinName.DOLLAR;
                                    break;
                        default:
                                    myName = null; // bad coin value, set it to null
                                    break;
                        }

                        year = mintYear;
                        sideUp = getToss();
            } // end constructor

            /**
             * Constructs an object for the coin having a given name and mint year. The
             * visible side of the new coin is set at random.
             * @param name type of coin
             * @param mintYear year the coin was made
             */
            public Coin(CoinName name, int mintYear) {
                        myName = name;
                        year = mintYear;
                        sideUp = getToss();
            }
```

Coin methods

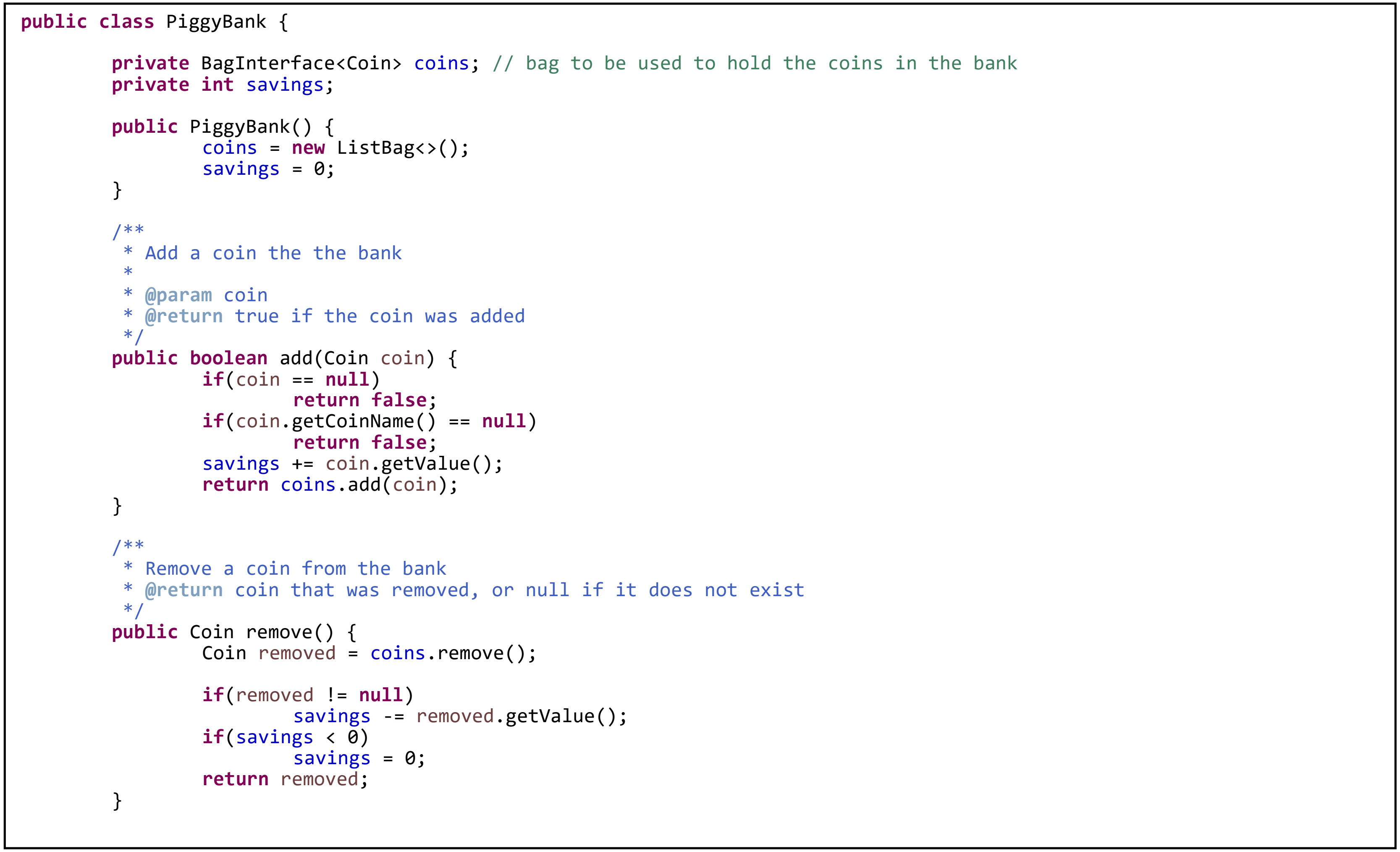
```java
/**
 * Returns name of the coin
 * @return
 */
public CoinName getCoinName() {
            return myName;
}

/**
 * Returns the value of the coin in cents
 * @return
 */
public int getValue() {
            return myName.getValue();
}

/** Returns the coin's mint year as an integer. */
public int getYear() {
            return year;
} // end getYear

/**
 * Returns "HEADS" or "TAILS"
 * @return
 */
public String getSideUp() {
            /*
             * String result = "Tails"; if (sideUp == CoinSide.HEADS) result = "Heads";
             * return result;
             */
            return sideUp.toString();
}

/**
 * Returns true if the coin is heads-side up.
 * @return
 */
public boolean isHeads() {
            return sideUp == CoinSide.HEADS;
}

/**
 * Returns true if the coin is tails-side up.
 * @return
 */
public boolean isTails() {
            return sideUp == CoinSide.TAILS;
}

/**
 * Tosses the coin; sideUp will be either HEADS or TAILS at random.
 */
public void toss() {
            sideUp = getToss();
}
```

Coin methods

```java
    @Override
    public String toString() {
        if(myName == null)
            return "[null coin]";
        return myName + " [" + myName.getValue() + ", " + year + ", " + sideUp + "]";
    }

    /**
     * Returns a random value of either HEADS or TAILS.
     * @return
     */
    private CoinSide getToss() {
        CoinSide result;
        if (Math.random() < 0.5)
            result = CoinSide.HEADS;
        else
            result = CoinSide.TAILS;

        return result;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;

        Coin other = (Coin) obj;

        if (myName != other.myName)
            return false;
        // let's not worry about the side up
//      if (sideUp != other.sideUp)
//          return false;

        if (year != other.year)
            return false;
        return true;
    }
```

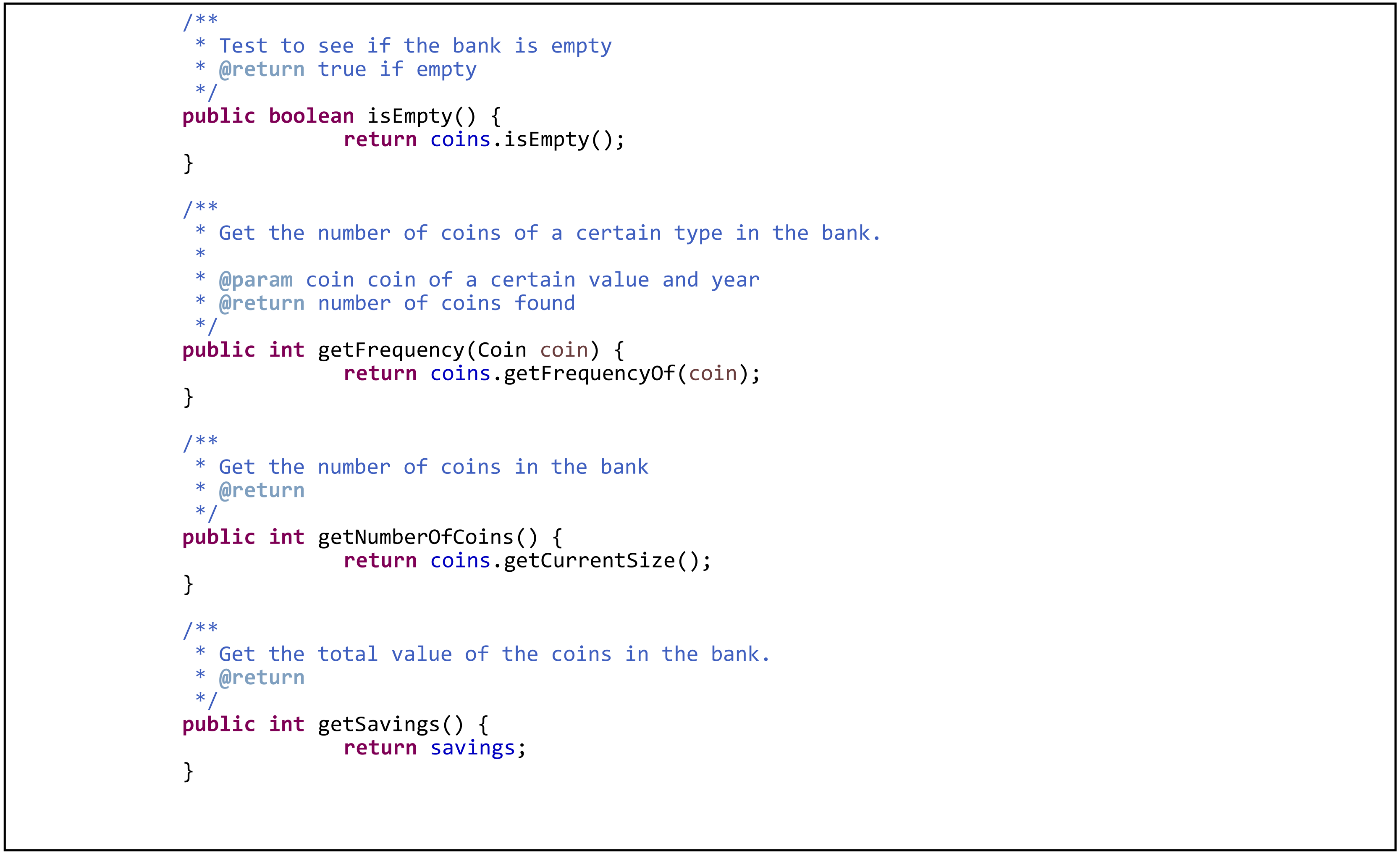
PiggyBank constructor, add(), remove()

- Add a coin to the bank, either by coin value or object



```java
public class PiggyBank {

        private BagInterface<Coin> coins; // bag to be used to hold the coins in the bank
        private int savings;

        public PiggyBank() {
                coins = new ListBag<>();
                savings = 0;
        }

        /**
         * Add a coin the the bank
         *
         * @param coin
         * @return true if the coin was added
         */
        public boolean add(Coin coin) {
                if(coin == null)
                        return false;
                if(coin.getCoinName() == null)
                        return false;
                savings += coin.getValue();
                return coins.add(coin);
        }

        /**
         * Remove a coin from the bank
         * @return coin that was removed, or null if it does not exist
         */
        public Coin remove() {
                Coin removed = coins.remove();

                if(removed != null)
                        savings -= removed.getValue();
                if(savings < 0)
                        savings = 0;
                return removed;
        }
```

PiggyBank methods



```
/**
 * Test to see if the bank is empty
 * @return true if empty
 */
public boolean isEmpty() {
             return coins.isEmpty();
}

/**
 * Get the number of coins of a certain type in the bank.
 * 
 * @param coin coin of a certain value and year
 * @return number of coins found
 */
public int getFrequency(Coin coin) {
             return coins.getFrequencyOf(coin);
}

/**
 * Get the number of coins in the bank
 * @return
 */
public int getNumberOfCoins() {
             return coins.getCurrentSize();
}

/**
 * Get the total value of the coins in the bank.
 * @return
 */
public int getSavings() {
             return savings;
}
```

PiggyBank app

- Add coins to bank, then remove them

```java
public class PiggyBankExample {
        public static void main(String[] args) {
                PiggyBank myBank = new PiggyBank();

                System.out.println(">>> Adding coins to the bank");

                addCoin(new Coin(1, 2010), myBank);
                addCoin(new Coin(5, 2011), myBank);
                addCoin(new Coin(10, 2000), myBank);
                addCoin(new Coin(25, 2012), myBank);
                addCoin(new Coin(25, 2012), myBank);
                addCoin(new Coin(-1, 2000), myBank);  // this is illegal, so should fail

                System.out.println("Number of quarters should be 2, is: "
                                + myBank.getFrequency(new Coin(25, 2012)));
                System.out.println("Bank should have 5 coins, value of 66, has: "
                                + myBank.getNumberOfCoins()
                                + " value: " + myBank.getSavings());

                System.out.println(">>> Removing all the coins from the bank:");

                int amountRemoved = 0;

                while (!myBank.isEmpty()) {
                        Coin removedCoin = myBank.remove();
                        System.out.println("Removed a " + removedCoin.getCoinName() + ".");
                        amountRemoved = amountRemoved + removedCoin.getValue();
                } // end while

                System.out.println(">>> All done. Removed " + amountRemoved + " cents.");
        } // end main

        private static void addCoin(Coin aCoin, PiggyBank aBank) {
                if (aBank.add(aCoin))
                        System.out.println("Added a " + aCoin + ".");
                else
                        System.out.println("Tried to add a " + aCoin + ", but couldn't");
        }
}
```

PiggyBank app output

- Note what happens when an illegal coin is added

```
>>> Adding coins to the bank
Added a PENNY [1, 2010, TAILS].
Added a NICKEL [5, 2011, TAILS].
Added a DIME [10, 2000, TAILS].
Added a QUARTER [25, 2012, TAILS].
Added a QUARTER [25, 2012, HEADS].
Tried to add a [null coin], but couldn't
Number of quarters should be 2, is: 2
Bank should have 5 coins, value of 66, has: 5 value: 66
>>> Removing all the coins from the bank:
Removed a QUARTER.
Removed a QUARTER.
Removed a DIME.
Removed a NICKEL.
Removed a PENNY.
>>> All done. Removed 66 cents.
```

Observations about Vending Machines

- Can perform only tasks machine's interface presents.
- You must understand these tasks
- Cannot access the inside of the machine
- You can use the machine even though you do not know what happens inside.
- Usable even with new insides.

I'm really thirsty— what looks good?
© 2019 Pearson Education, Inc.

FIGURE 1-3
A vending machine

Observations about ADT Bag

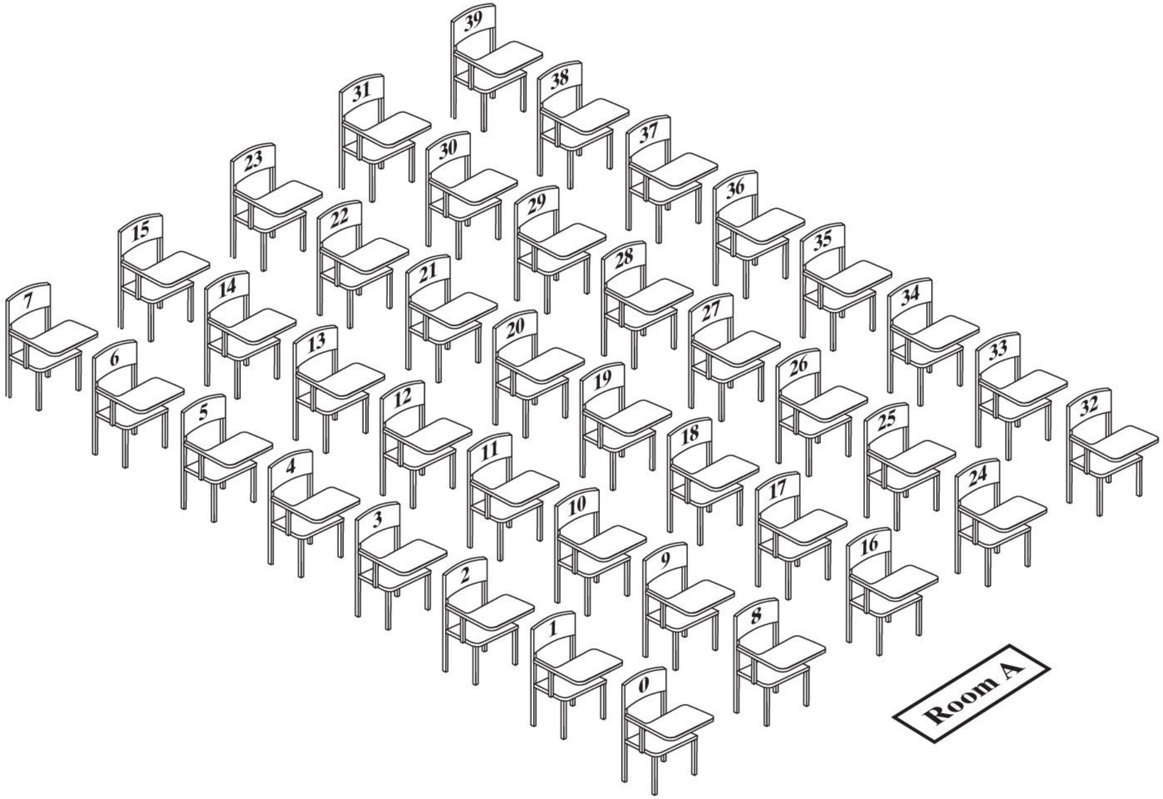
- Can perform only tasks specific to ADT
- Must adhere to the specifications of the operations of ADT
- Cannot access data inside ADT without ADT operations
- Use the ADT, even if don't know how data is stored
- Usable even with new implementation.

Java Class Library: The Interface Set

```
/** 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
```

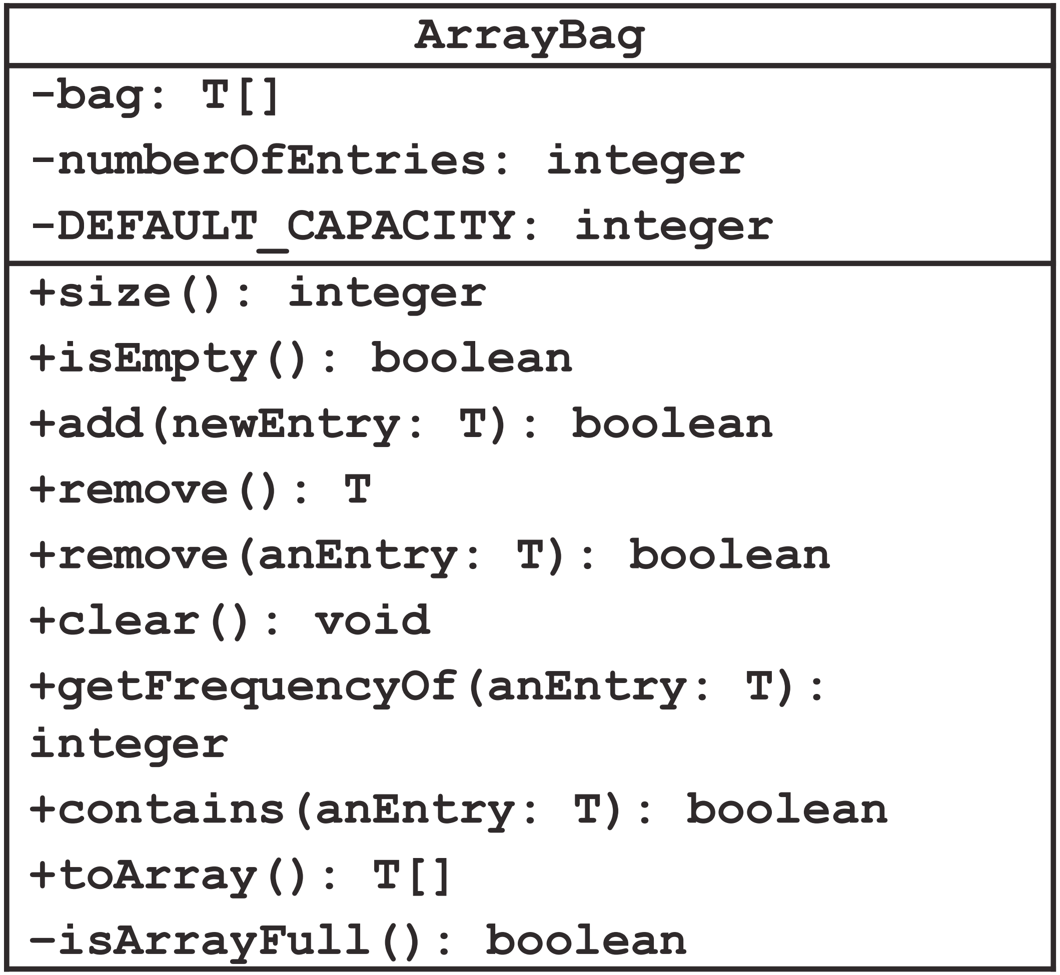
Fixed-Size Array to Implement the ADT Bag

- **A classroom that contains desks in fixed positions**



© 2019 Pearson Education, Inc.

UML for a fixed size `ArrayBag`



```
ArrayBag
-bag: T[]
-numberOfEntries: integer
-DEFAULT_CAPACITY: integer
+size(): 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
```

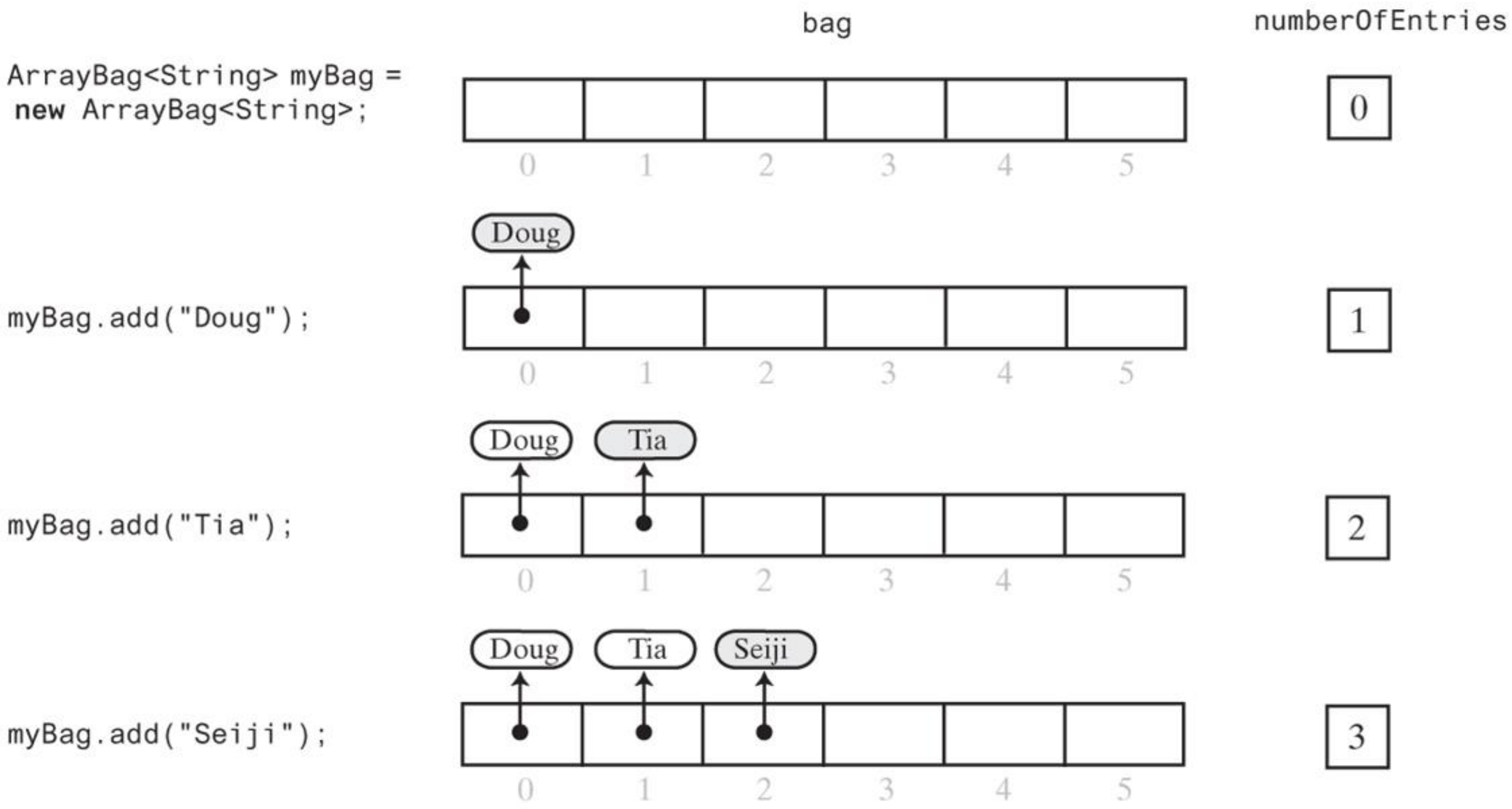
FixedSizeArrayBag

- Implement Bag using a fixed size generic array.
- Constructor creates an array then casts it to the generic type
- Need other methods such as add(), remove(), toArray()

```java
public final class CompletedFixedSizeArrayBag<T> implements BagInterface<T> {
     private final T[] bag;  // array to hold bag of objects
     private int numberOfEntries;  // count of objects in the bag

     private static final int DEFAULT_CAPACITY = 25;
     private static final int MAX_CAPACITY = 10000;

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

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

Adding to a fixed-size `ArrayBag` (Part 1)

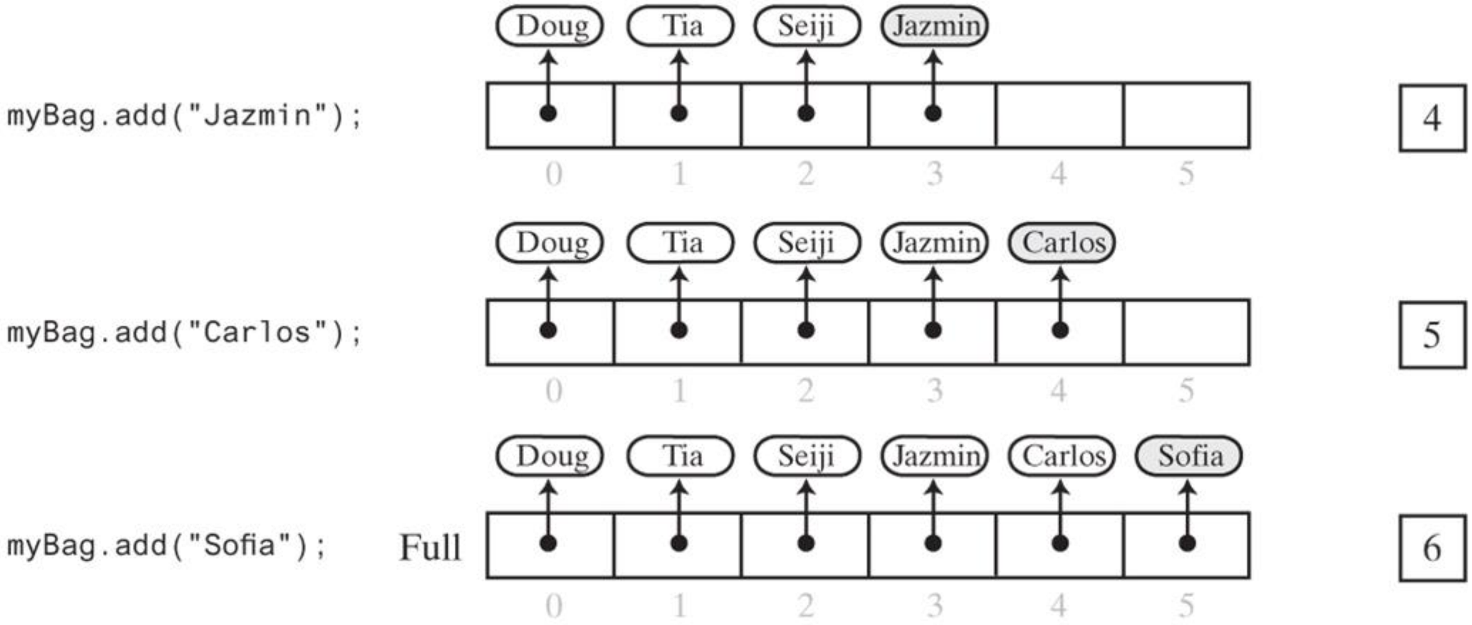
- **Adding entries to an array that represents a bag, whose capacity is six, until it becomes full**



```
ArrayBag<String> myBag =
 new ArrayBag<String>;

myBag.add("Doug");

myBag.add("Tia");

myBag.add("Seiji");
```

Adding to a fixed-size `ArrayBag` (Part 2)

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



```
myBag.add("Jazmin");
myBag.add("Carlos");
myBag.add("Sofia");
```

© 2019 Pearson Education, Inc.

add() method

- Add entry to the next free position (at end)
- Notice this is at the numberOfEntries slot, not at the end of the actual array

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

       return result;
}

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

toArray() – gets a copy of the bag

- Two ways of implementing this
  - Iterate through the internal array, copying to a new array
  - Use Arrays.copyOf() from the java library

```
    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];
//      }
//      return result;

        // Note: The body of this method could consist of one return statement,
        // if you call Arrays.copyOf

        return Arrays.copyOf(bag, numberOfEntries);
    }
```

Testing Bag implementations

- See BagDemo
  - Includes testing for each Bag implementation method
- Simply uncomment the implementation to be tested.

```
public class BagDemo {
    public static void main(String[] args) {
        // A bag that is not full
//      BagInterface<String> aBag = new CompletedFixedSizeArrayBag<>();
        BagInterface<String> aBag = new CompletedLinkedBagWithNodeMethods<>();
//      BagInterface<String> aBag = new CompletedLinkedBag<>();
//      BagInterface<String> aBag = new FixedSizeArrayBag<>();
//      BagInterface<String> aBag = new LinkedBagWithNodeMethods<>();
//      BagInterface<String> aBag = new LinkedBag<>();
        System.out.println("Testing an initially empty bag:");

        // Removing a string from an empty bag:
        String[] testStrings1 = { "", "B" };
        testRemove(aBag, testStrings1);
```

BagDemo displayBag()

- Tests toArray()
- Gets copy of bag items, and iterates through the array to display.

```
/**
 * Display the bag using toArray method
 * 
 * @param aBag
 */
private static void displayBag(BagInterface<String> aBag) {
    System.out.println("The bag contains " + aBag.size() + " string(s), as follows:");
    Object[] bagArray = aBag.toArray();
    for (int index = 0; index < bagArray.length; index++) {
        System.out.print(bagArray[index] + " ");
    }

    System.out.println();
}
```

Other methods

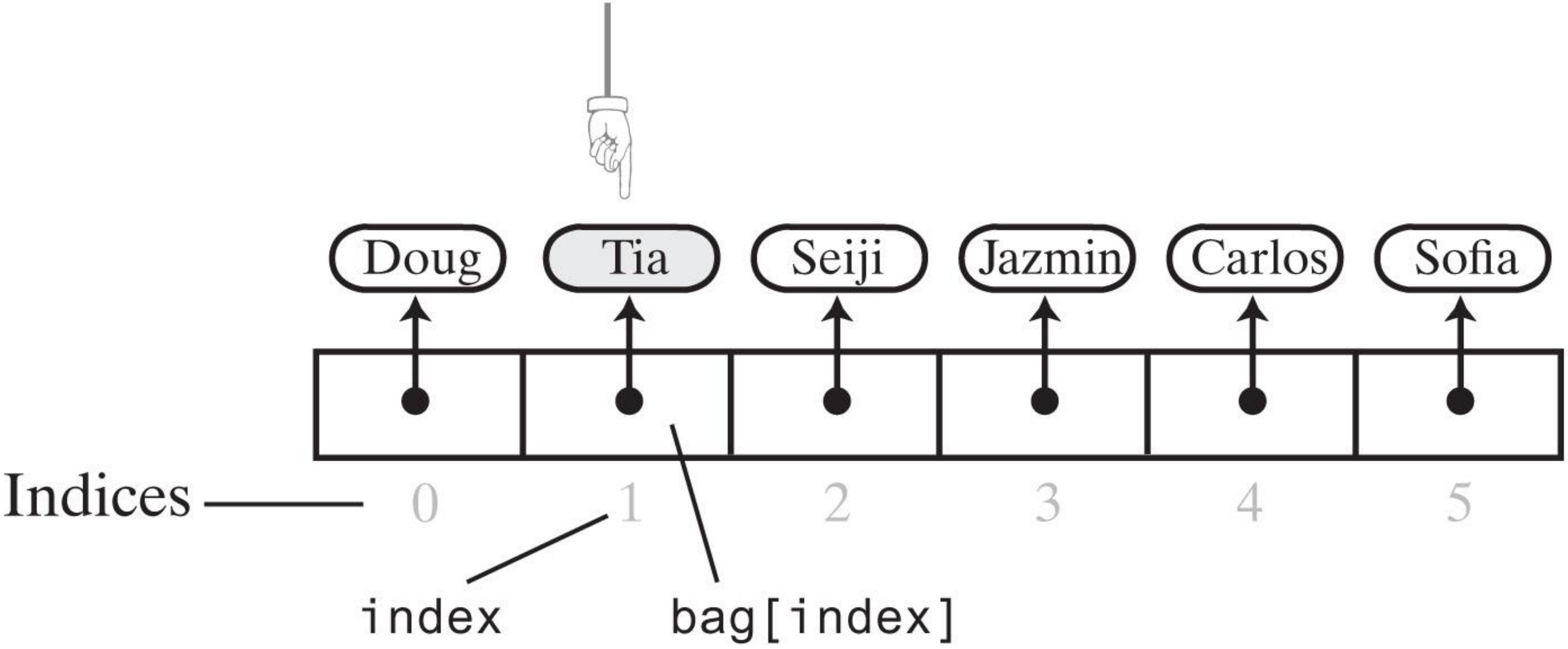
```
public boolean isEmpty() {
    return numberOfEntries == 0;
}

public int getCurrentSize() {
    return numberOfEntries;
}

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

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

    return counter;
}

public void clear() {
    while (!isEmpty())
        remove();
}
```

contains() and getIndexOf()

- contains() just calls getIndexOf()
- getIndexOf() iterates through the array until entry found.
    - returns -1 if not found

```java
public boolean contains(T anEntry) {
    return getIndexOf(anEntry) > -1; // or >= 0
}

/**
 * Locates a given entry within the array bag.
 * @param anEntry entry to locate
 * @return index of the entry, if located, or -1 otherwise
 */
private int getIndexOf(T anEntry) {
    int where = -1;  // location of entry
    boolean found = false;   // sentinel
    int index = 0;

    // look through all entries until we find a matching one

    while (!found && (index < numberOfEntries)) {
        // if we find it, set the flag, and
        // save the index where found
        if (anEntry.equals(bag[index])) {
            found = true;
            where = index;
        }
        index++;
    }

    return where;
}
```

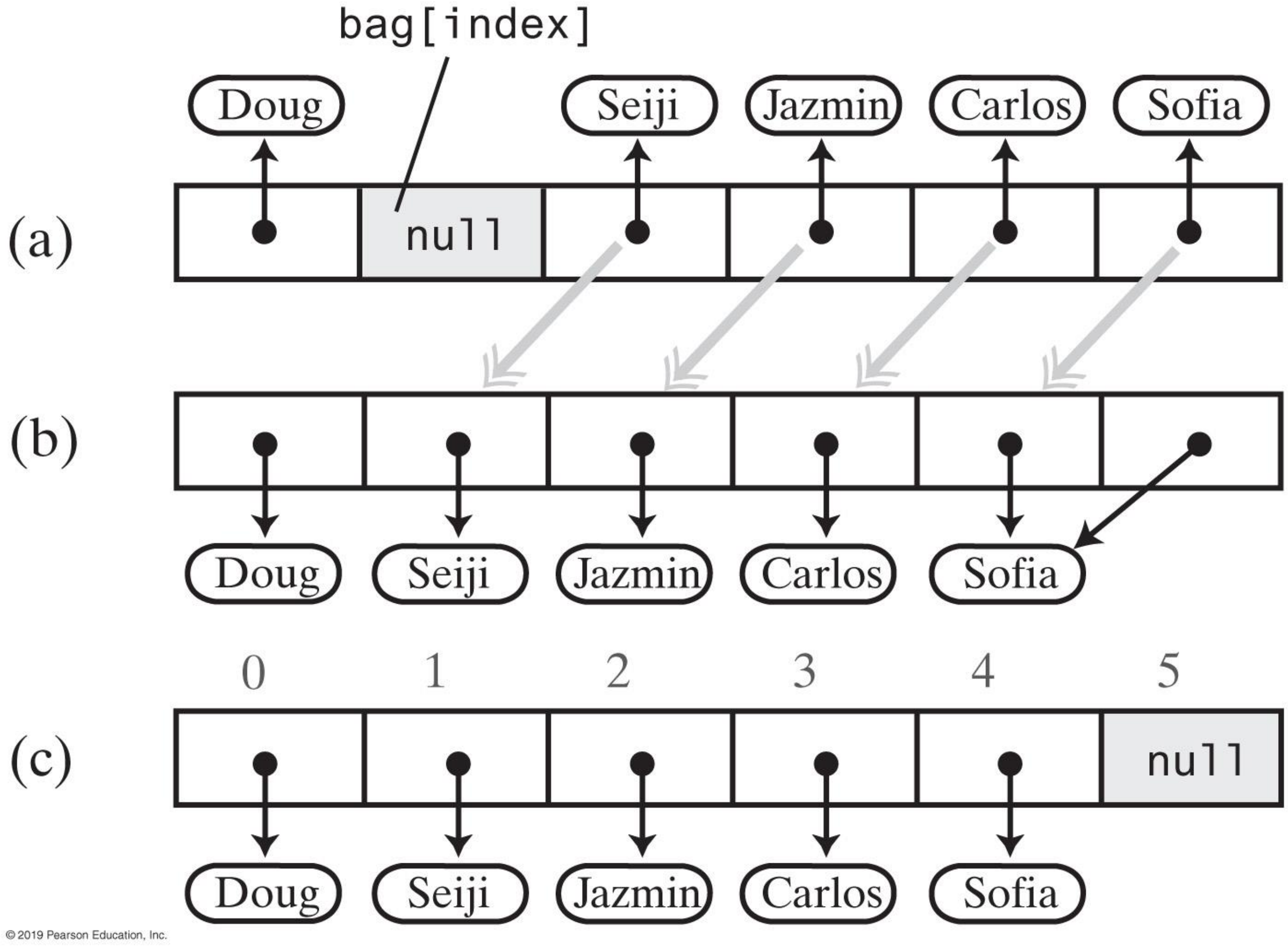
Methods That Remove Entries



© 2019 Pearson Education, Inc.

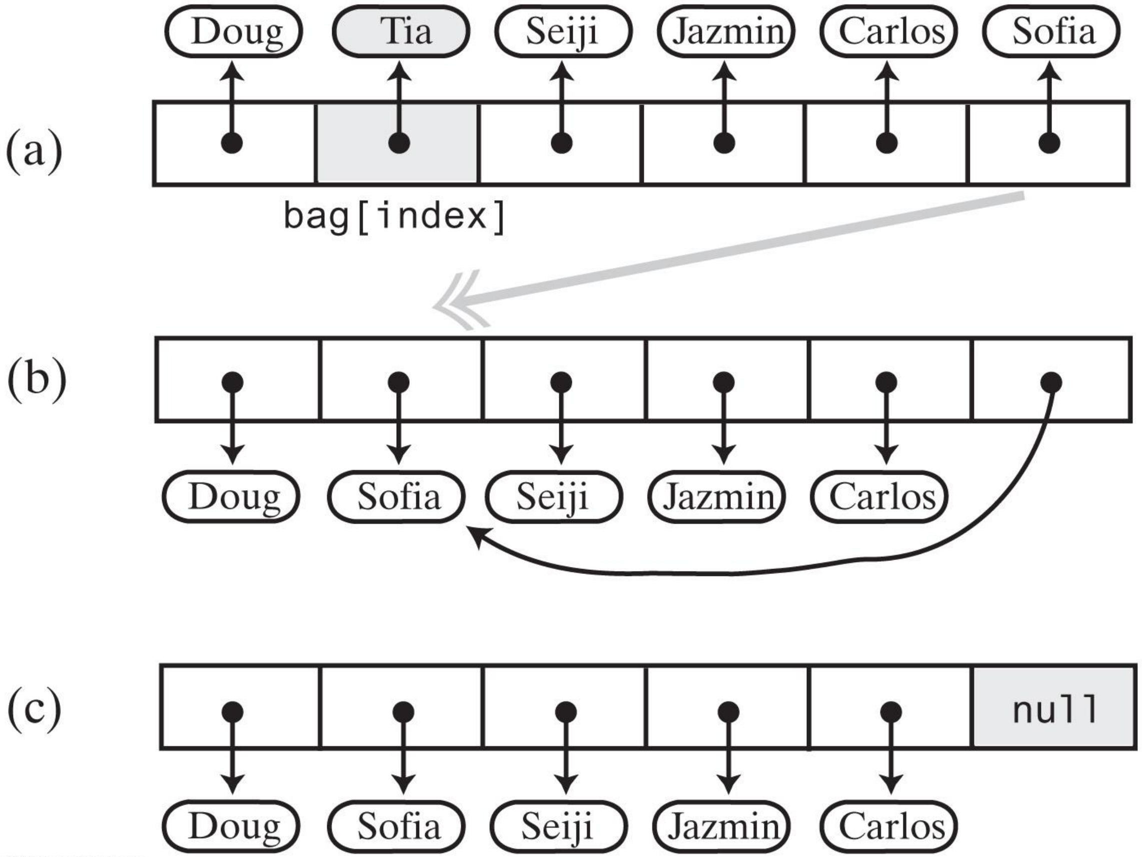
Methods That Remove Entries

- **Shifting entries from back to front to avoid a gap when removing an entry**



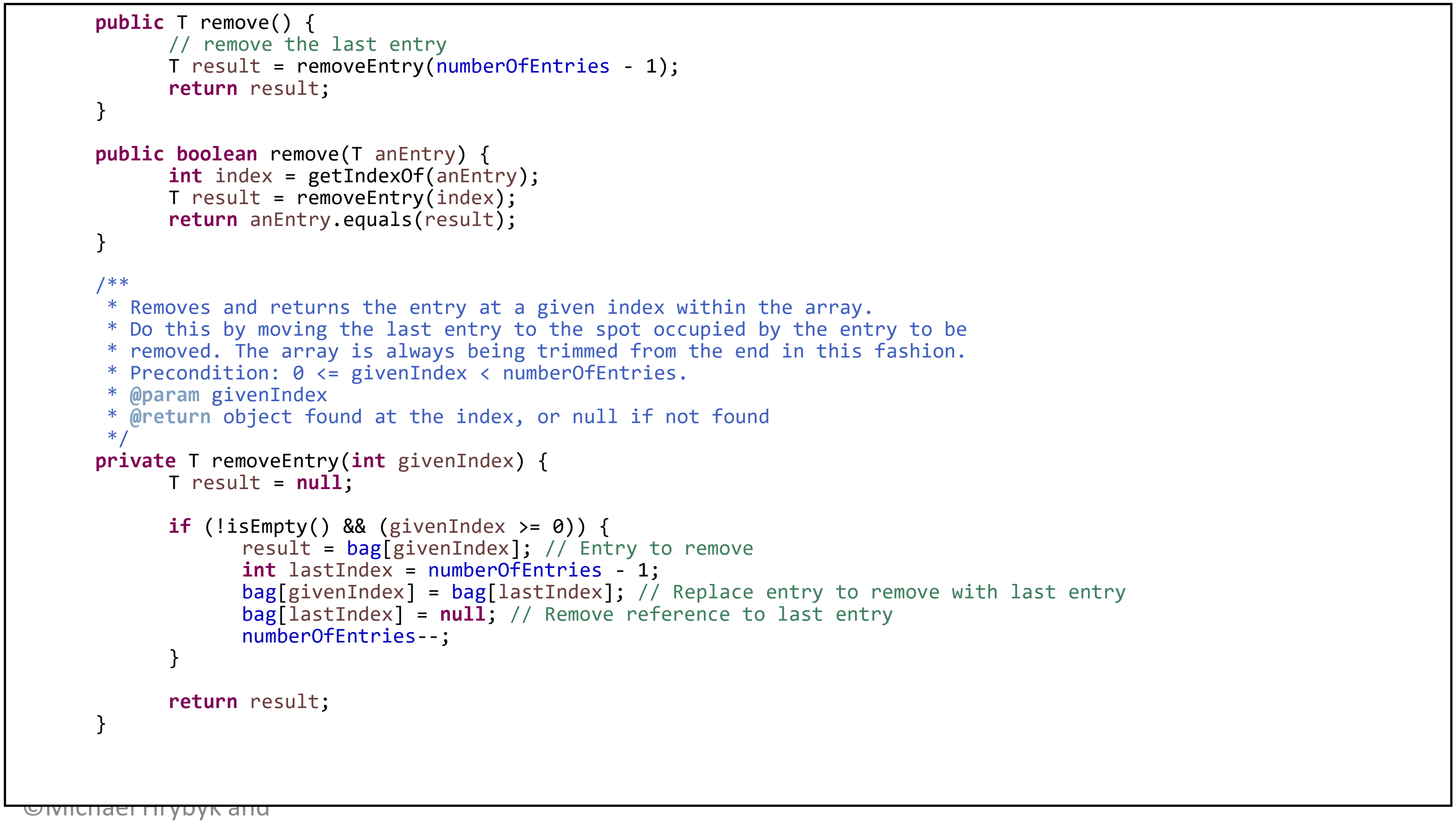
Methods That Remove Entries

- **Swap the last entry into the slot where the entry is removed**



remove()

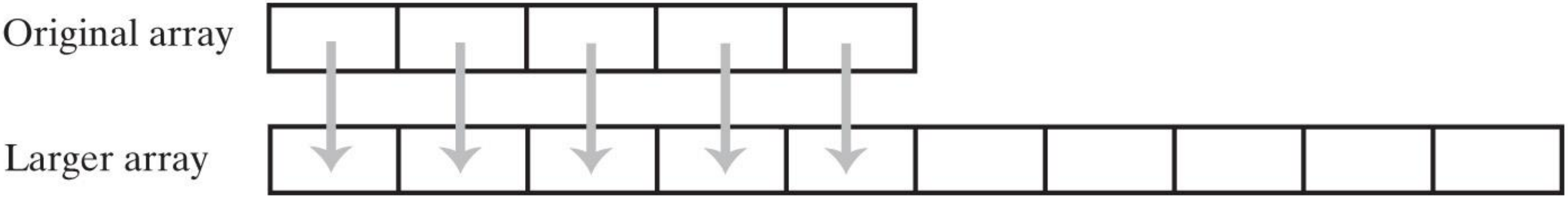
- Removing items from anything is always tricky
- remove() with no args just removes the last entry and returns the data.
    - Note call to removeEntry(), which removes an item at a position
    - In this case, the last position
- remove(T anEntry) first has to find it and get its index, then remove the position.
- removeEntry() works by swapping the entry to be removed with the last entry in the bag.



```
public T remove() {
    // remove the last entry
    T result = removeEntry(numberOfEntries - 1);
    return result;
}

public boolean remove(T anEntry) {
    int index = getIndexOf(anEntry);
    T result = removeEntry(index);
    return anEntry.equals(result);
}

/**
 * Removes and returns the entry at a given index within the array.
 * Do this by moving the last entry to the spot occupied by the entry to be
 * removed. The array is always being trimmed from the end in this fashion.
 * Precondition: 0 <= givenIndex < numberOfEntries.
 * @param givenIndex
 * @return object found at the index, or null if not found
 */
private T removeEntry(int givenIndex) {
    T result = null;

    if (!isEmpty() && (givenIndex >= 0)) {
        result = bag[givenIndex]; // Entry to remove
        int lastIndex = numberOfEntries - 1;
        bag[givenIndex] = bag[lastIndex]; // Replace entry to remove with last entry
        bag[lastIndex] = null; // Remove reference to last entry
        numberOfEntries--;
    }

    return result;
}
```

Resizing an Array

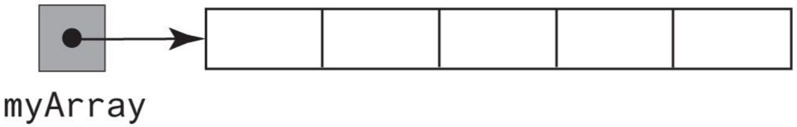
- **Resizing an array copies its contents to a larger second array**



© 2019 Pearson Education, Inc.

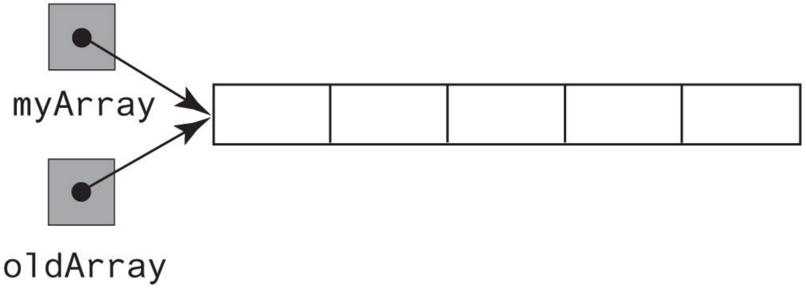
Steps to Resize an Array (Part 1)

(a) An array



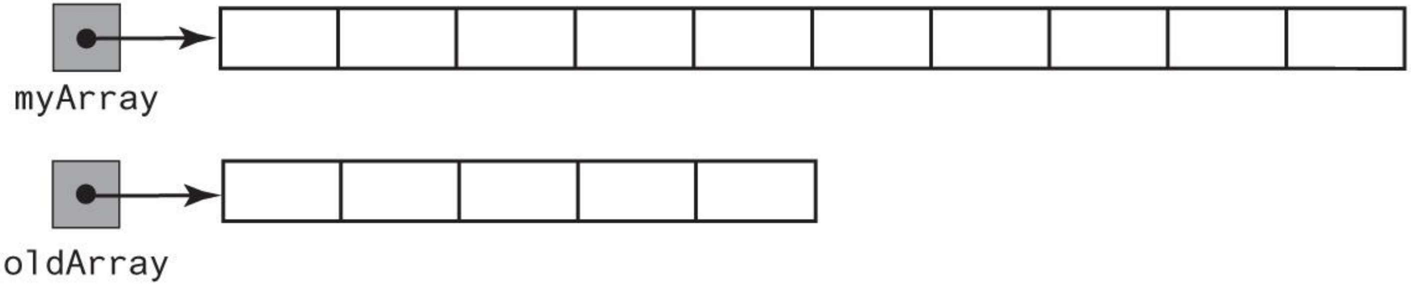
© 2019 Pearson Education, Inc.

(b) Two references to the same array



© 2019 Pearson Education, Inc.

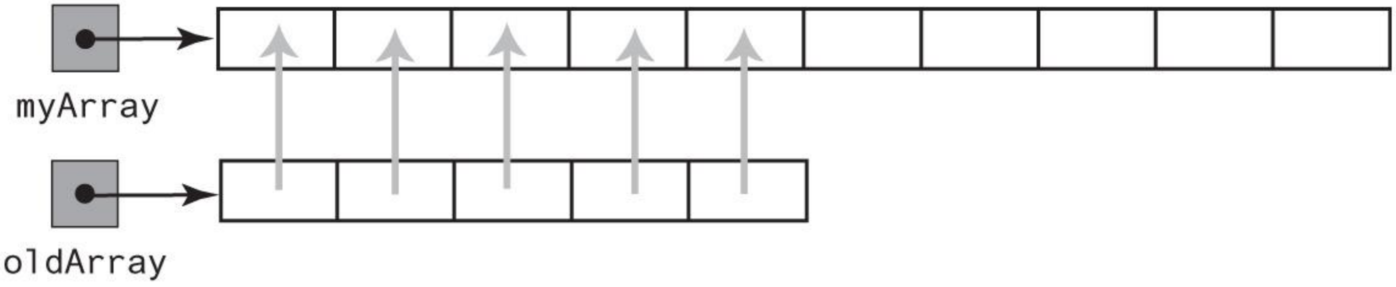
(c) The original array variable now references a new, larger array



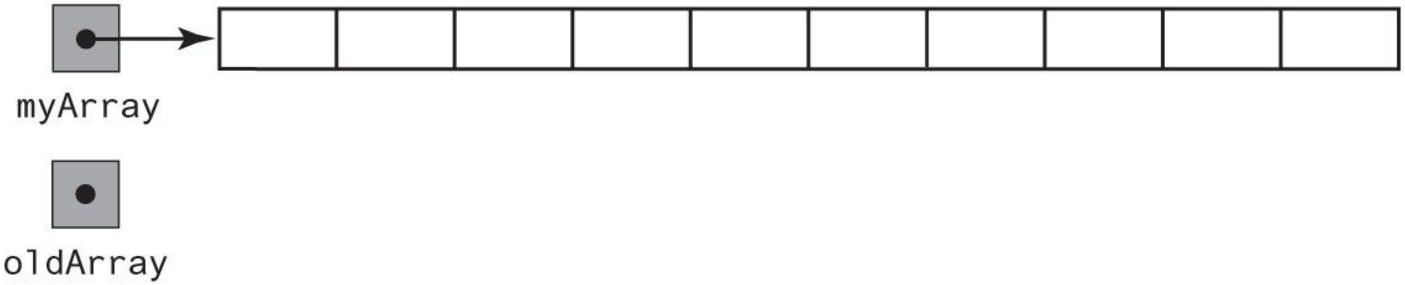
© 2019 Pearson Education, Inc.

Steps to Resize an Array (Part 2)

(d) The entries in the original array are copied to the new array

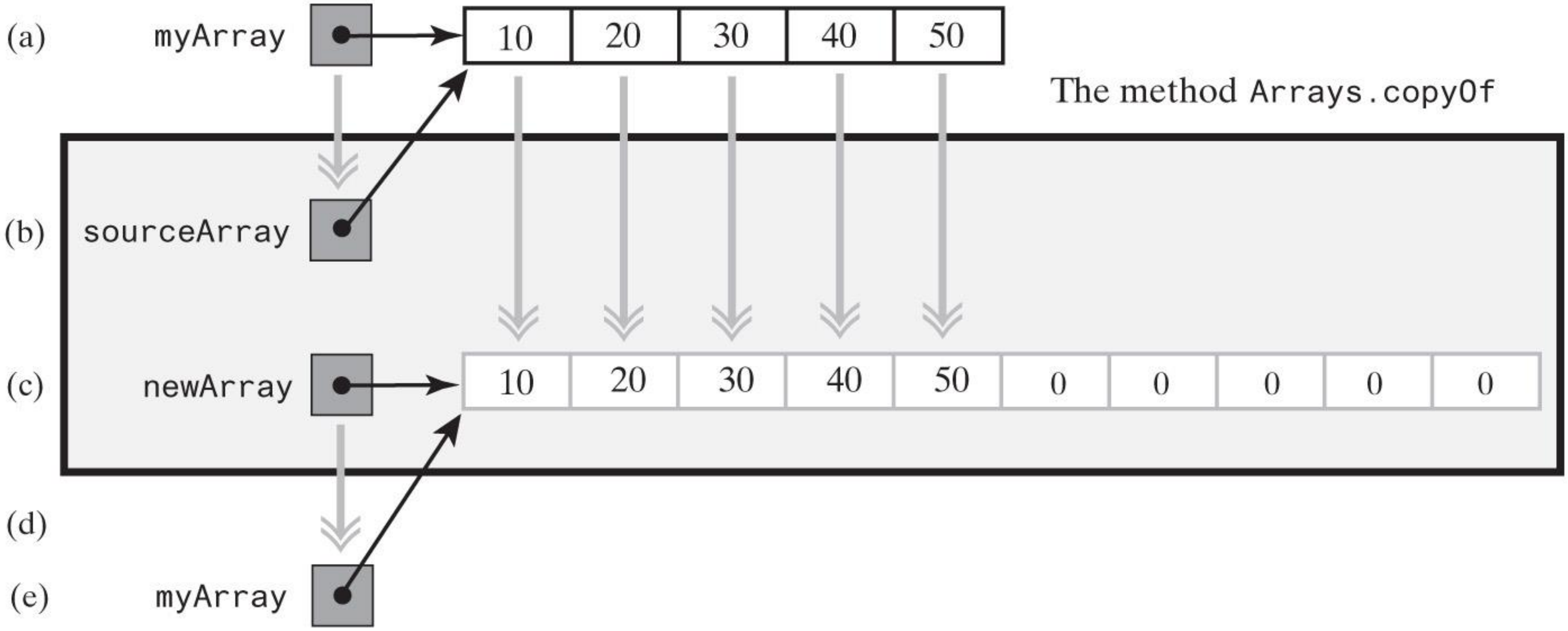


(e) The original array is discarded



Resizing Using `Arrays.copyOf`

- **Alternative steps to resize an array**



© 2019 Pearson Education, Inc.

ResizableArrayBag

- Double the size when full in add()

```
    public boolean add(T newEntry) {
       // double the capacity if full then add the entry

       if (isArrayFull()) {
          doubleCapacity();
       }

       bag[numberOfEntries] = newEntry;
       numberOfEntries++;

       return true;
    }

    /**
     * Doubles the size of the bag
     */
    private void doubleCapacity() {
       int newLength = 2 * bag.length;
       checkCapacity(newLength);
       bag = Arrays.copyOf(bag, newLength);
    }

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

Pros and Cons of Using an Array

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

Problems with Array Implementation

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

Classroom Analogy

- Empty classroom
- Numbered desks stored in hallway
  - Number on back of desk is the "address"
- Number on desktop references another desk in chain of desks
- Desks are linked by the numbers

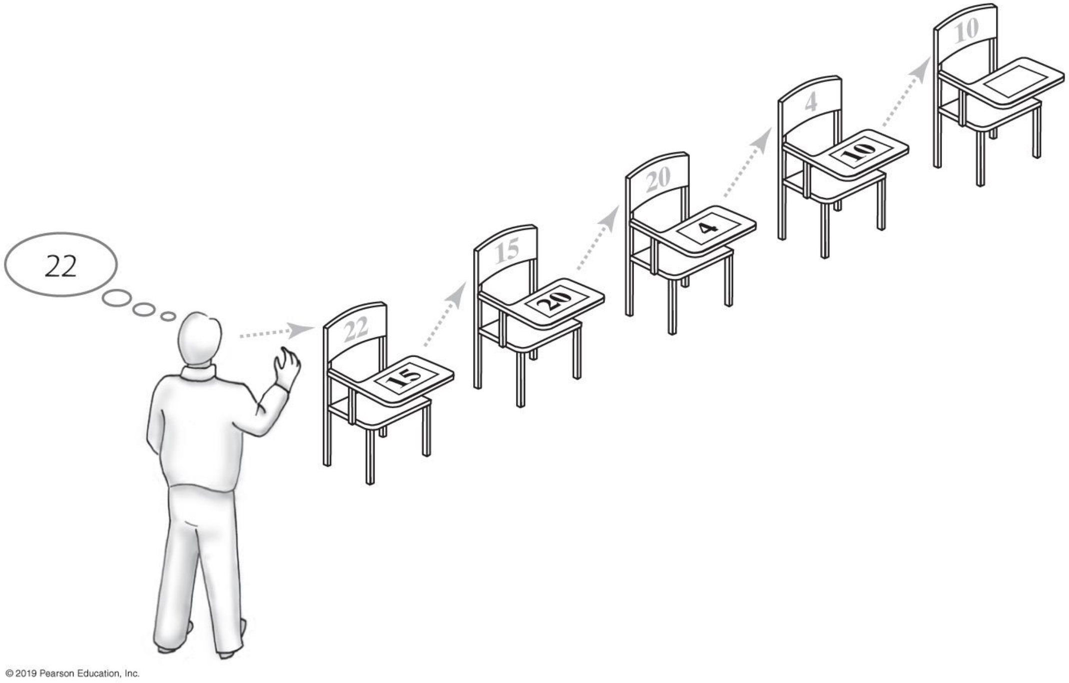
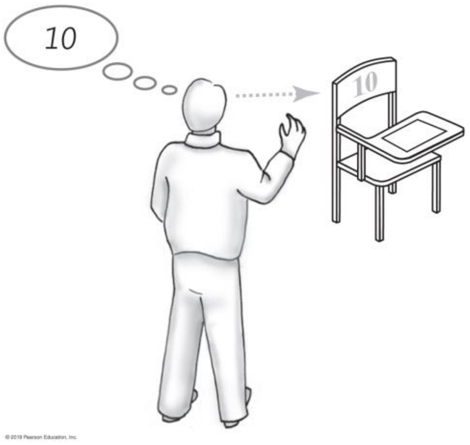


FIGURE 3-1 A chain of five desks

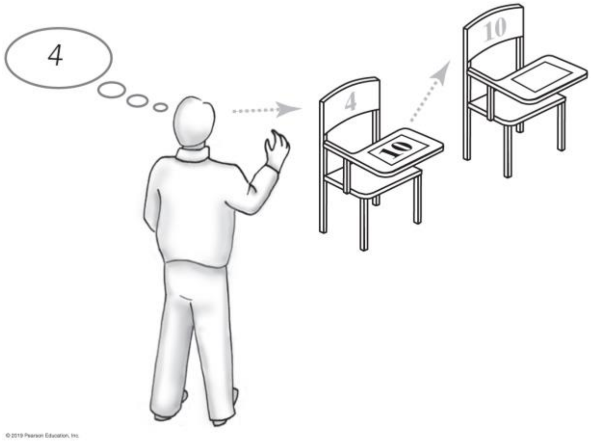
Forming a Chain by Adding to Its Beginning



© 2019 Pearson Education, Inc.

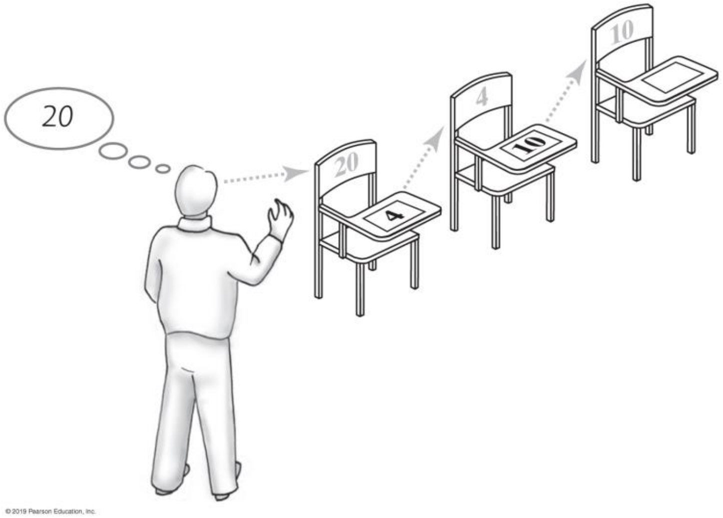
FIGURE 3-2

One desk in the room



© 2019 Pearson Education, Inc.

FIGURE 3-3
Two linked desks, with the newest desk first



© 2019 Pearson Education, Inc.

FIGURE 3-4
Three linked desks, with the newest desk first

Forming a Chain by Adding to Its Beginning

- **Pseudocode detailing steps taken to form a chain of desks**

```
//Process the first student
newDesk represents the new student's desk New student sits at newDesk
Instructor memorizes the address of newDesk
// Process the remaining students
while (students arrive)
{
    newDesk represents the new student's desk New student sits at newDesk
    Write the instructor's memorized address on newDesk
    Instructor memorizes the address of newDesk
}
```

Using linked Nodes to build a chain

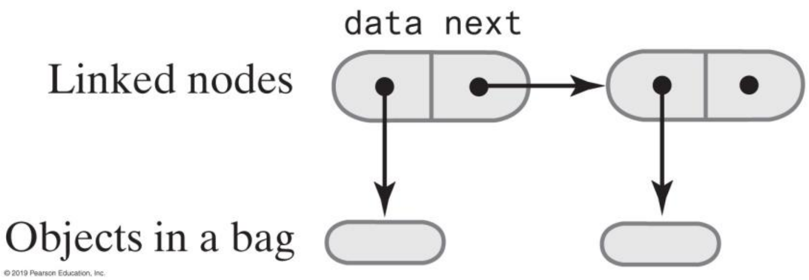


FIGURE 3-5
Two linked nodes that each reference object data

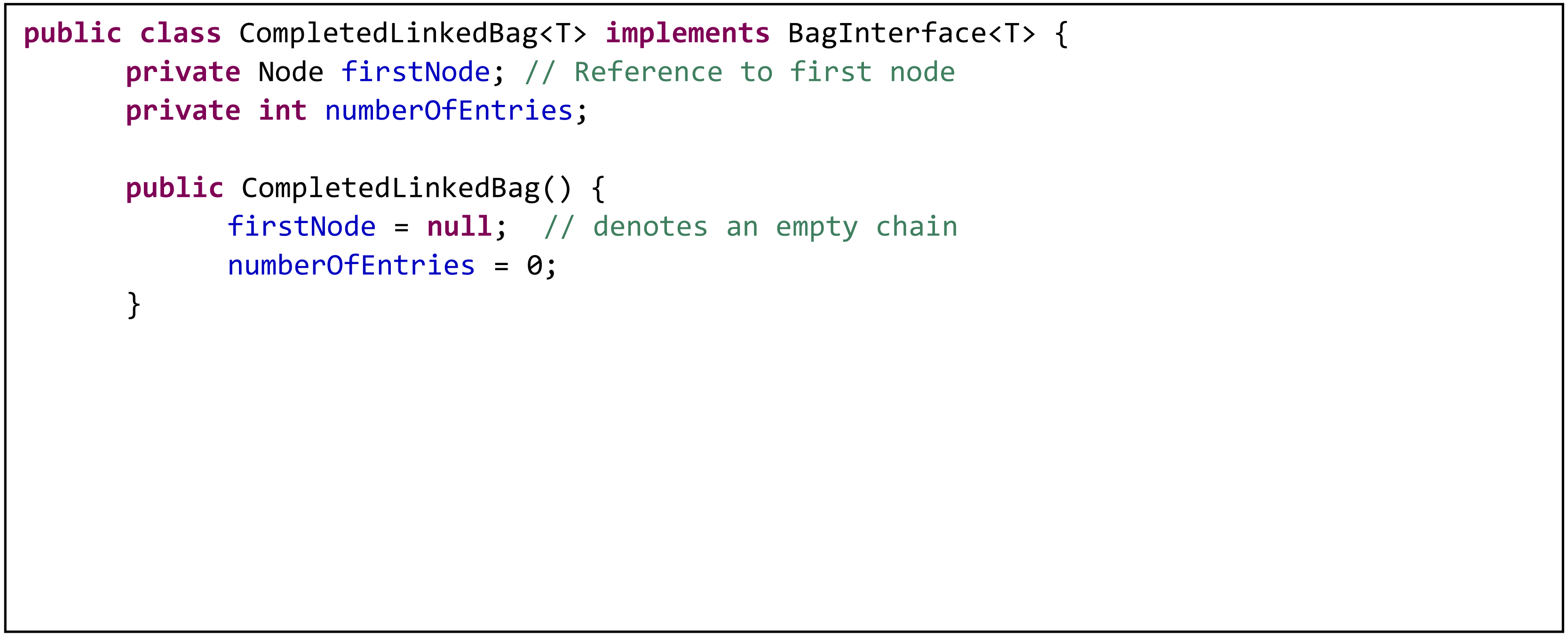
Private Node class

- used in an implementation to link data
- enclosing class can set and get next and data directly without getter/setter
- Might be better to use a public Node class instead

```java
private class Node {
      private T data; // Entry in bag
      private Node next; // Link to next node

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

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

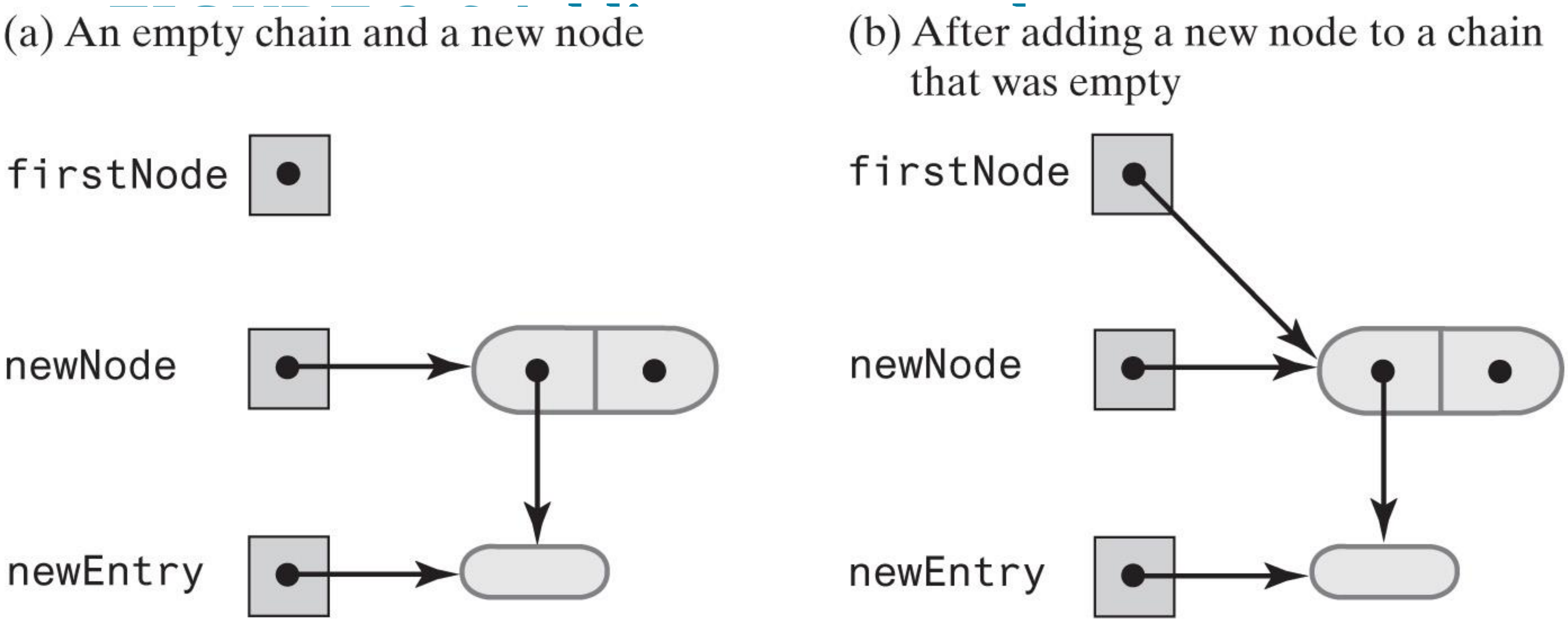
LinkedBag

- A bag consisting of a set of linked nodes
- Keep a reference to the head of the list
- Notice that there are no size limits



```
public class CompletedLinkedBag<T> implements BagInterface<T> {
      private Node firstNode; // Reference to first node
      private int numberOfEntries;

      public CompletedLinkedBag() {
            firstNode = null;  // denotes an empty chain
            numberOfEntries = 0;
      }
```

Beginning a Chain of Nodes

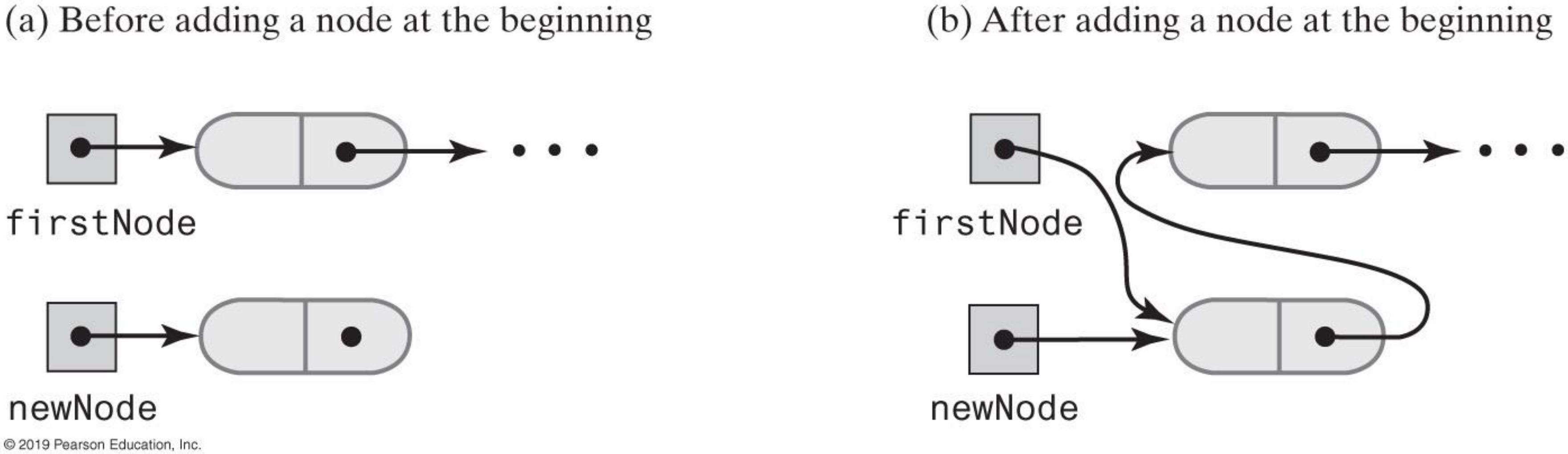


(a) An empty chain and a new node

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

Beginning a Chain of Nodes

- A chain of nodes just before and just after adding a node at



(a) Before adding a node at the beginning

(b) After adding a node at the beginning

© 2019 Pearson Education, Inc.

add()

```
	public boolean add(T newEntry) // OutOfMemoryError possible
	{
		// Add to beginning of chain:
		Node newNode = new Node(newEntry);
		newNode.next = 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;
	}
```

toArray()

- need to walk through chain to get each node, then copy data to an array slot

```java
    public T[] toArray() {
        // The cast is safe because the new array contains null entries
        @SuppressWarnings("unchecked")
        T[] result = (T[]) new Object[numberOfEntries]; // Unchecked cast

        // walk through the chain
        int index = 0;
        Node currentNode = firstNode;  // start at beginning of chain
        while ((index < numberOfEntries) && (currentNode != null)) {
            result[index] = currentNode.data; // copy the data
            index++;
            currentNode = currentNode.next;  // go to the next link on the chain
        } // end while

        return result;
    }
```

contains()

- calls getReferenceTo(), which gets the Node that contains the desired data
- getReferenceTo() walks through the chain until data found
    - Always sets current Node to next

```
public boolean contains(T anEntry) {
    return getReferenceTo(anEntry) != null ? true : false;
}

/**
 * Locates a given entry within this bag.
 * Returns a reference to the node containing the entry, if located,
 * or null otherwise.
 * @param anEntry the data to look for in the list of nodes
 * @return the node containing the data
 */
private Node getReferenceTo(T anEntry) {
    boolean found = false;
    Node currentNode = firstNode;

    while (!found && (currentNode != null)) {
        if (anEntry.equals(currentNode.data))
            found = true;
        else
            currentNode = currentNode.next;
    }

    return currentNode;
}
```

getFrequencyOf()

- like getReferenceTo(), walks through the chain
- counts the number of times data is found

```
public int getFrequencyOf(T anEntry) {
    int frequency = 0;
    int loopCounter = 0;
    Node currentNode = firstNode;

    while ((loopCounter < numberOfEntries) && (currentNode != null)) {
        if (anEntry.equals(currentNode.data)) {
            frequency++;
        }

        loopCounter++;
        currentNode = currentNode.next;
    }

    return frequency;
}
```

Removing an Item from a Linked Chain

- **Case 1:**
  - Your desk is first in the chain of desks.
- **Case 2:**
  - Your desk is not first in the chain of desks.

Removing an Item from a Linked Chain

- **Case 1 – removing first item**
  - Locate first desk by asking instructor for its address.
  - Give address written on the first desk to instructor.
    - This is address of second desk in chain.
  - Return first desk to hallway.

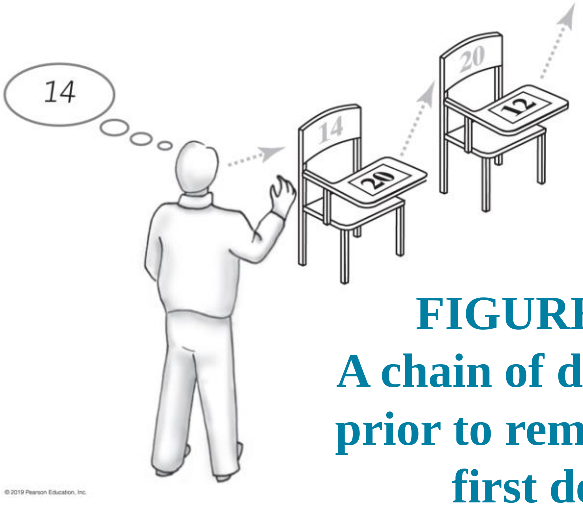


FIGURE 3-8
A chain of desks just prior to removing its first desk

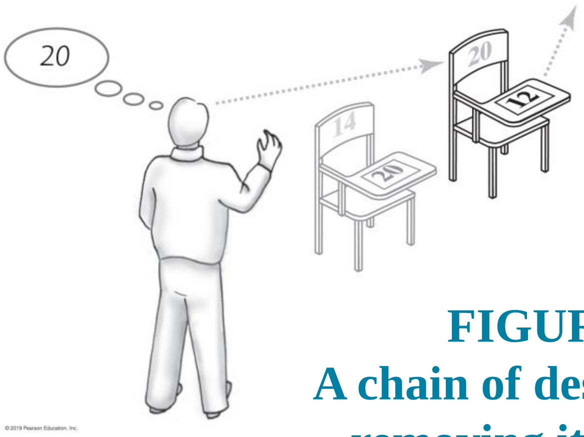
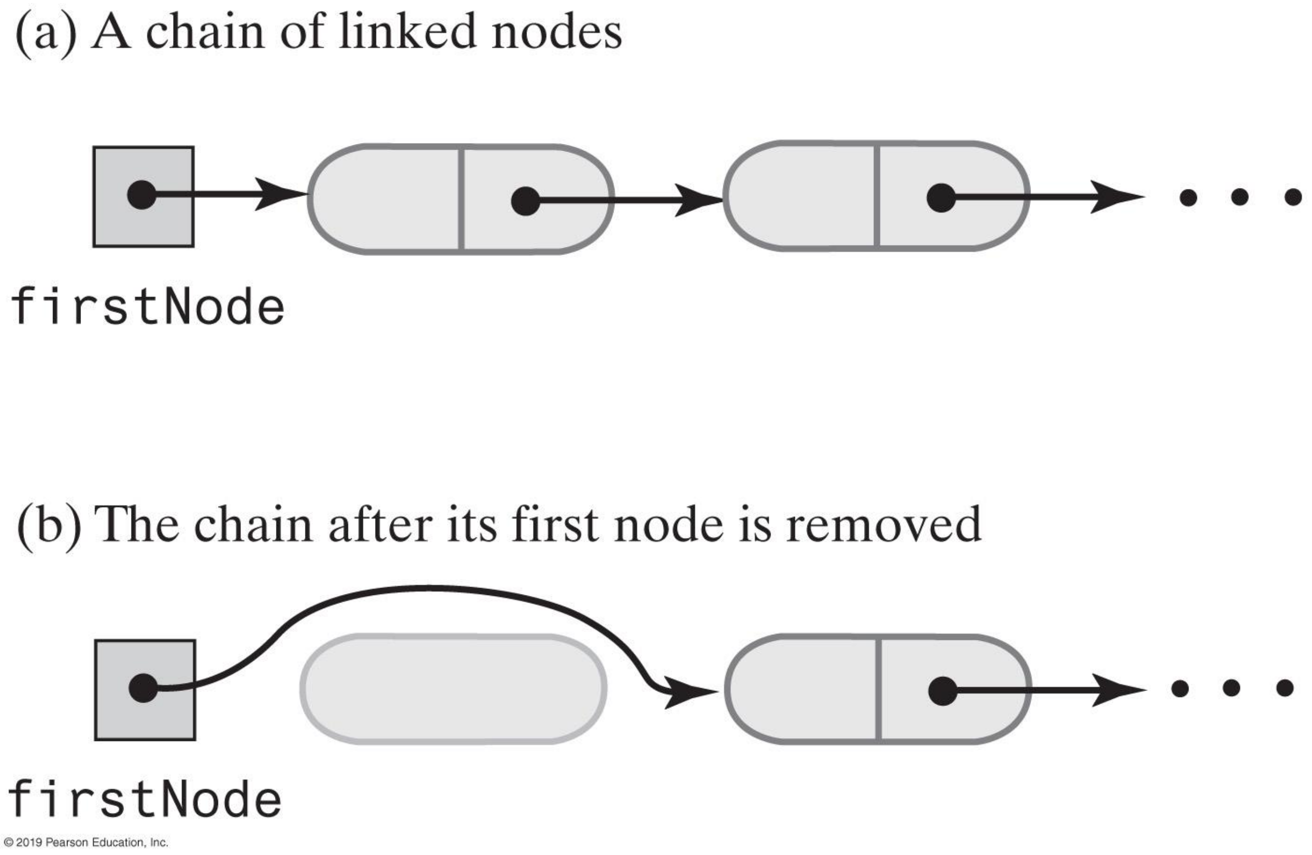


FIGURE 3-9
A chain of desks just after removing its first desk

Removing an Item from a Linked Chain

- **FIGURE 3-10 A chain of nodes just before and just after its first node is removed**


(a) A chain of linked nodes
firstNode
(b) The chain after its first node is removed
firstNode
© 2019 Pearson Education, Inc.

Removing an Item from a Linked Chain

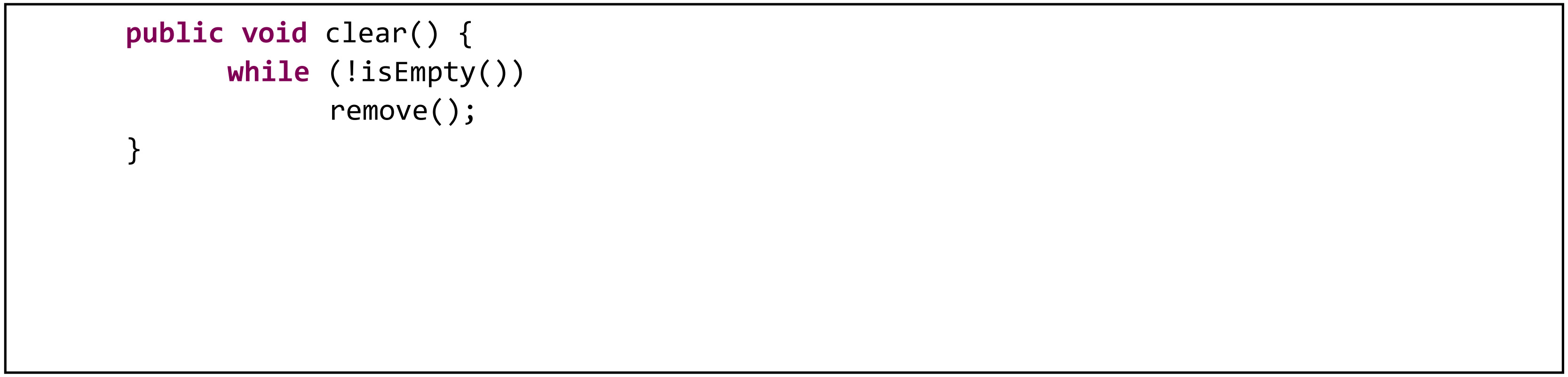
- **Case 2 – removing other items**
  - o Move the student in the first desk to your former desk.
  - o Remove the first desk using the steps described for Case 1.

remove()

- remove() with no args just removes the first entry
- remove() with an arg
    - find the node
    - place data from first node in the found node (duplicates it)
    - remove the first node
    - No need to save the data as it was passed from the caller

```java
    public T remove() {
        T result = null;
        if (firstNode != null) {
            result = firstNode.data;
            firstNode = firstNode.next; // Remove first node from chain
            numberOfEntries--;
        }

        return result;
    }

    public boolean remove(T anEntry) {
        boolean result = false;
        Node entryNode = getReferenceTo(anEntry);

        if (entryNode != null) {
            // Replace located entry with entry in first node
            entryNode.data = firstNode.data;
            firstNode = firstNode.next; // Remove first node
            numberOfEntries--;
            result = true;
        } // end if

        return result;
    }
```

clear() – can use remove()!!



```
public void clear() {
      while (!isEmpty())
            remove();
}
```

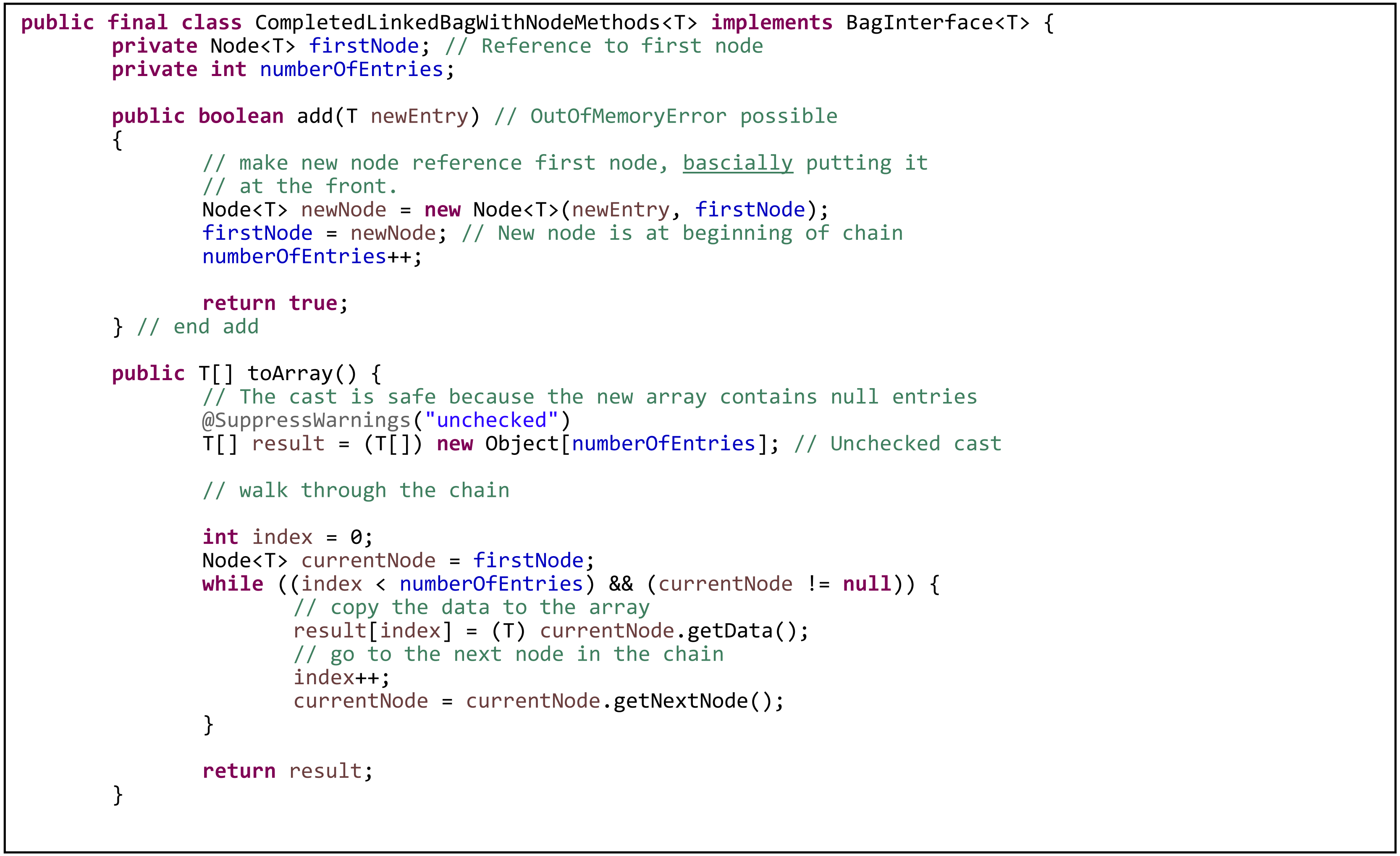
Public class Node

- Need getters/setters as data/next is private

```java
/**
 * Node in a linked list. Each node
 * contains data and a link to the next node in the
 * list.
 */
public class Node<T> {
    private T data; // Entry in bag
    private Node<T> next; // Link to next node

    /**
     * Create a new node containing data
     * @param dataPortion
     */
    public Node(T dataPortion) {
        this(dataPortion, null);
    }
    /**
     * Create a new node containing data
     * and set the next node.
     * @param dataPortion
     * @param nextNode
     */
    public Node(T dataPortion, Node<T> nextNode) {
        data = dataPortion;
        next = nextNode;
    }
    /**
     * Get the data from the node
     * @return
     */
    public T getData() {
        return data;
    }

    /**
     * Set the data in the node
     * @param newData
     */
    public void setData(T newData) {
        data = newData;
    }

    /**
     * Get the next node
     * @return
     */
    public Node<T> getNextNode() {
        return next;
    }

    /**
     * Set the next node
     * @param nextNode
     */
    public void setNextNode(Node<T> nextNode) {
        next = nextNode;
    }
}
```

LinkedBag methods using public Node Class

- Substitute get/set methods when accessing data/next
- Apply changes to other methods such as remove()



```
public final class CompletedLinkedBagWithNodeMethods<T> implements BagInterface<T> {
        private Node<T> firstNode; // Reference to first node
        private int numberOfEntries;

        public boolean add(T newEntry) // OutOfMemoryError possible
        {
                // make new node reference first node, bascially putting it
                // at the front.
                Node<T> newNode = new Node<T>(newEntry, firstNode);
                firstNode = newNode; // New node is at beginning of chain
                numberOfEntries++;

                return true;
        } // end add

        public T[] toArray() {
                // The cast is safe because the new array contains null entries
                @SuppressWarnings("unchecked")
                T[] result = (T[]) new Object[numberOfEntries]; // Unchecked cast

                // walk through the chain

                int index = 0;
                Node<T> currentNode = firstNode;
                while ((index < numberOfEntries) && (currentNode != null)) {
                        // copy the data to the array
                        result[index] = (T) currentNode.getData();
                        // go to the next node in the chain
                        index++;
                        currentNode = currentNode.getNextNode();
                }

                return result;
        }
```

Pros 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

Cons of Using a Chain

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

Why Efficient Code?

- Computers are faster, have larger memories
  - So why worry about efficient code?
- And … how do we measure efficiency?

Importance of Efficiency

- Consider the problem of summing

- Compare three algorithms

$$\sum_{k=1}^{n} k = 1 + 2 + 3 + \ldots + n$$

Algorithlm A	Algorithm B	Algorithm C

long sum = 0;
for (long i = 1; i <= n; i++)
  sum = sum + i;
	sum = 0;
	for (long i = 1; i <= n; i++)
	{
	  for (long j = 1; j <= i; j++)
	    sum = sum + 1;
	} // end for
		sum = n * (n + 1) / 2;

What is “best”?

- An algorithm has both time and space constraints – that is complexity
  - Time complexity
  - Space complexity
- This study is called analysis of algorithms

Counting Basic Operations

- A basic operation of an algorithm
  - Most significant contributor to its total time requirement

	Algorithm A	Algorithm B	Algorithm C

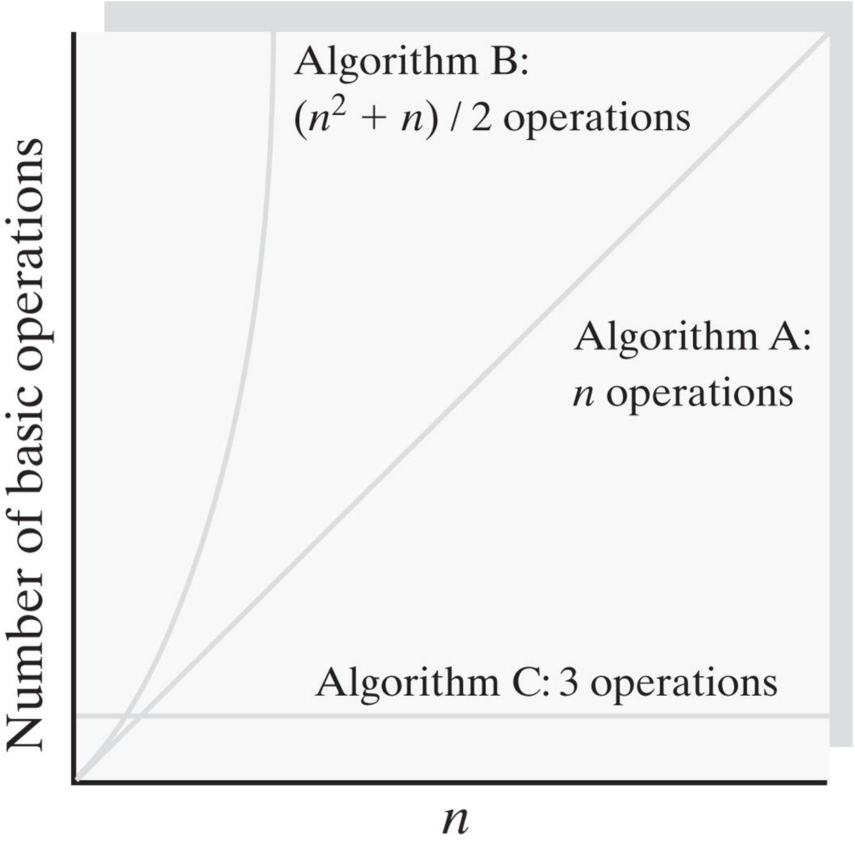
	long sum = 0;
	for (long i = 1; i <= n; i++)
	   sum = sum + i;
		sum = 0;
		for (long i = 1; i <= n; i++)
		{
		   for (long j = 1; j <= i; j++)
		      sum = sum + 1;
		} // end for
			sum = n * (n + 1) / 2;

Additions	n	$n(n + 1)/2$	1
Multiplications	0	0	1
Divisions	0	0	1
Total Basic Operations	n	$(n^2 + n)/2$	3

FIGURE 4-2 The number of basic operations required by the algorithms

Counting Basic Operations

- **Number of basic operations required by the algorithms as a function of *n***



© 2019 Pearson Education, Inc.

Counting Basic Operations

- **Typical growth-rate functions evaluated at increasing values of *n***

	(log(log n)	log n	$\log^2 n$		$n \log n$	n^2	n^3	2^n	$n!$
10	2	3	11	10	33	10^{2}	10^{3}	10^{3}	10^{5}
10^{2}	3	7	44	100	664	10^{4}	10^{6}	10^{30}	10^{94}
10^{3}	3	10	99	1,000	9,966	10^{6}	10^{9}	10^{301}	10^{1435}
10^{4}	4	13	177	10,000	132,877	10^{8}	10^{12}	10^{3010}	$10^{19,335}$
10^{5}	4	17	276	100,00	1,660,964	10^{10}	10^{15}	$10^{30,103}$	$10^{243,338}$
10^{6}	4	20	397	1,000,000	19,931,569	10^{12}	10^{18}	$10^{301,301}$	$10^{2,933,369}$

Best, Worst, and Average Cases

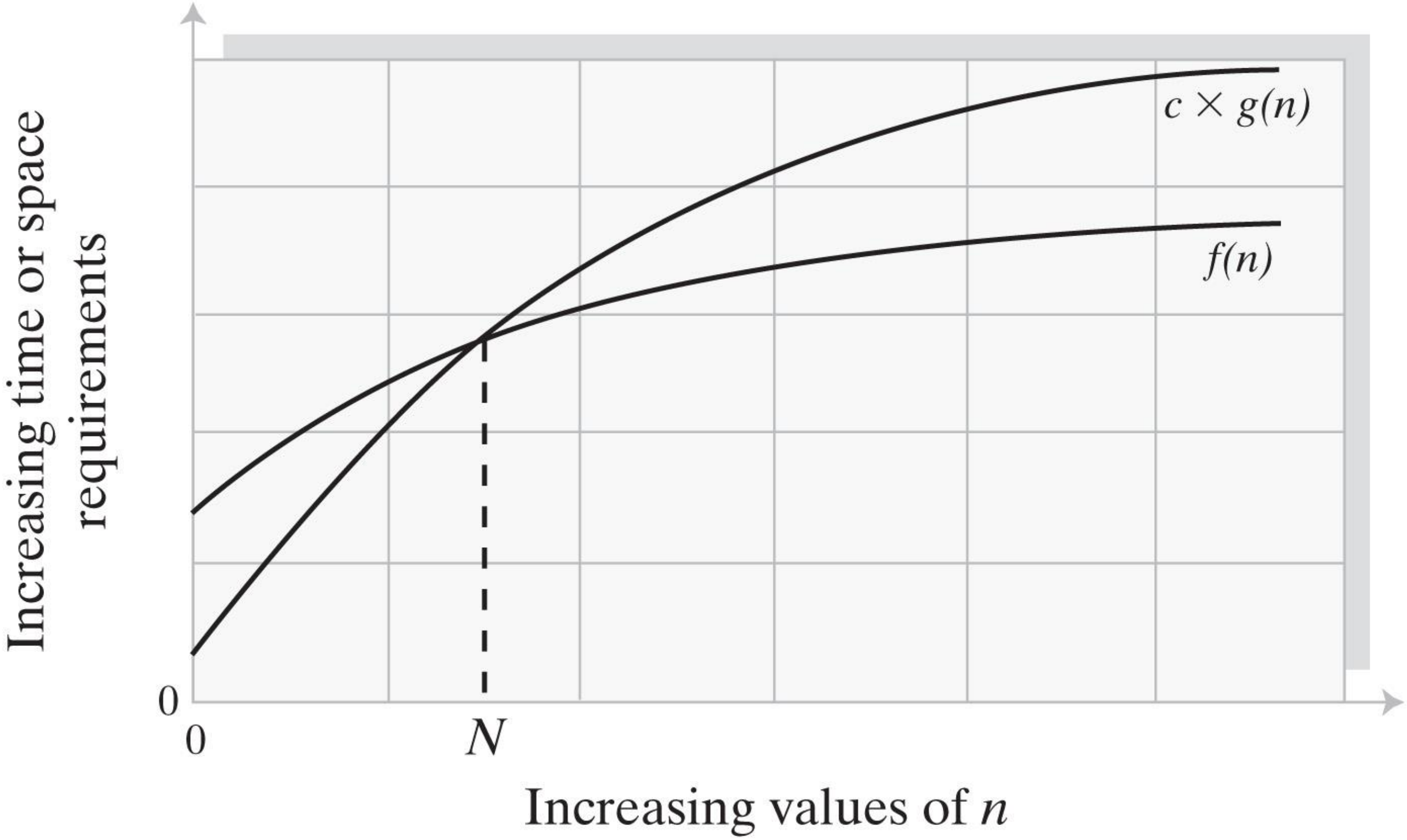
- For some algorithms, execution time depends only on size of data set
- Other algorithms depend on the nature of the data itself
    - Goal is to know best case, worst case, average case

Big Oh Notation

- A function $f(n)$ is of order at most $g(n)$
- That is, $f(n)$ is $O(g(n))$ — if
  - A positive real number c and positive integer N exist …
  - Such that $f(n) \leq c\ x\ g(n)$ for all $n \geq N$
  - That is:
    - c x $g(n)$ is an upper bound on $f(n)$ when n is sufficiently large

Big Oh Notation

- **An illustration of the values of two growth-rate functions**



Identities for Big Oh Notation

$O(k\ g(n)) = O(g(n))$ for a constant k

$O(g\ 1\ (n)) + O(g\ 2\ (n)) = O(g\ 1\ (n) + g\ 2\ (n))$

$O(g\ 1\ (n)) * O(g\ 2\ (n)) = O(g\ 1\ (n) * g\ 2\ (n))$

$O(g\ 1\ (n) + g\ 2\ (n) + \ldots + g\ m\ (n)) =$

$O(max(g\ 1\ (n), g\ 2\ (n), \ldots, g\ m\ (n))$

$O(max(g\ 1\ (n), g\ 2\ (n), \ldots, g\ m\ (n)) =$

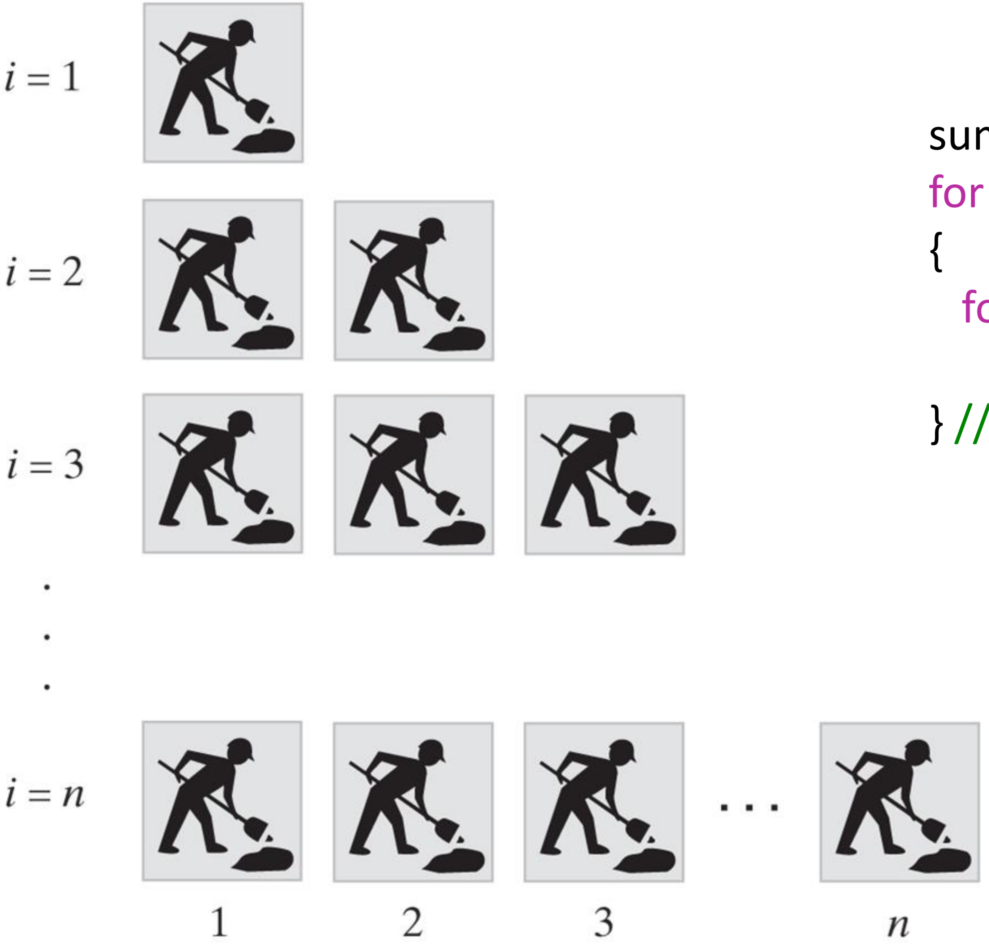
$max(O(g\ 1\ (n)), O(g\ 2\ (n)), \ldots, O(g\ m\ (n)))$

Picturing Efficiency of O(n) algorithm

```
long sum = 0;
for (long i = 1; i <= n; i++)
  sum = sum + i;
```

© 2019 Pearson Education, Inc.

Picturing Efficiency O(n^2)

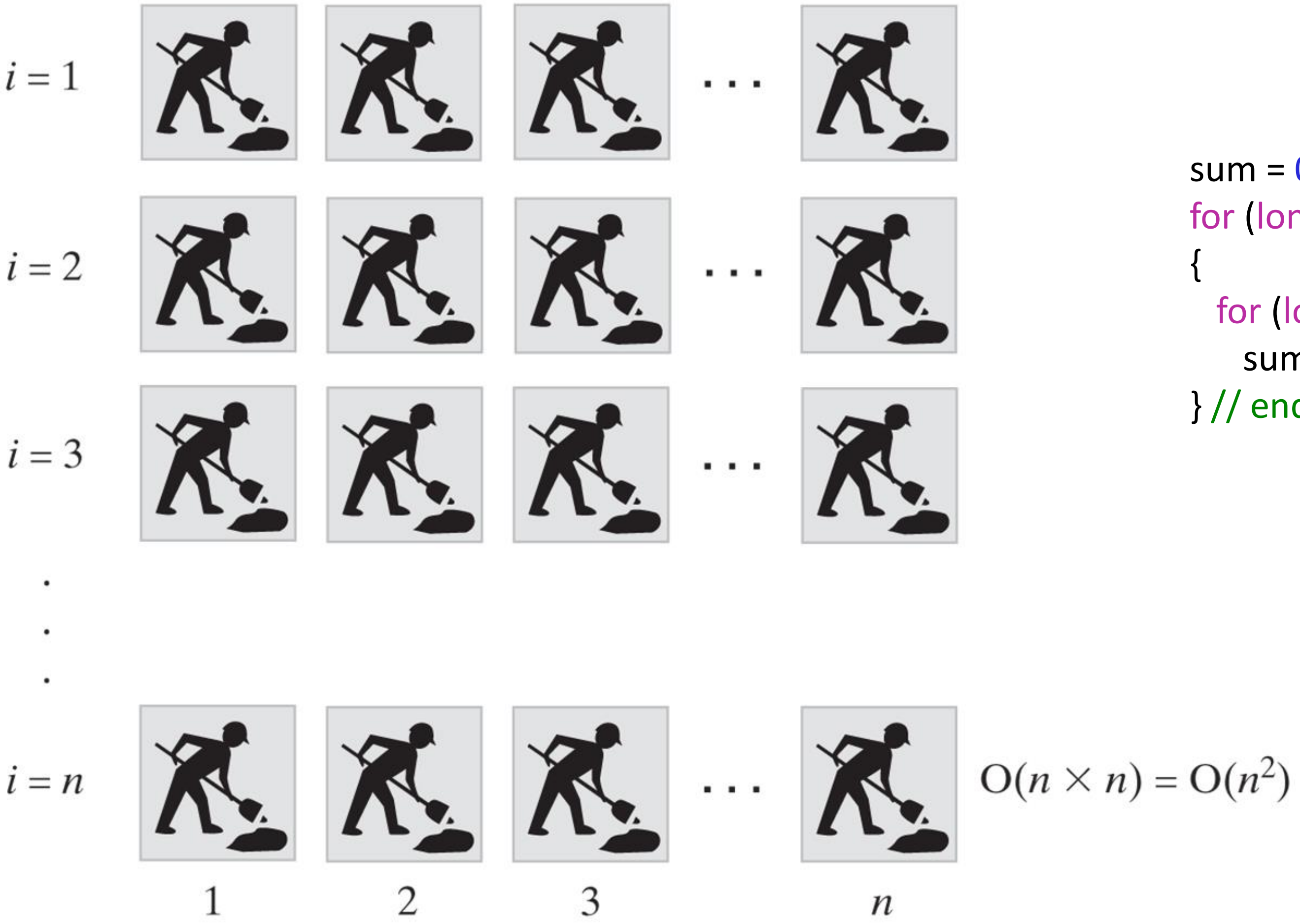


© 2019 Pearson Education, Inc.

```
sum = 0;
for (long i = 1; i <= n; i++)
{
   for (long j = 1; j <= i; j++)
      sum = sum + 1;
} // end for
```

$$O(1 + 2 + \ldots + n) = O(n^2)$$

Picturing Efficiency $O(n^2)$



© 2019 Pearson Education, Inc.

```
sum = 0;
for (long i = 1; i <= n; i++)
{
  for (long j = 1; j <= n; j++)
    sum = sum + 1;
} // end for
```

Picturing Efficiency

- **The effect of doubling the problem size on an algorithm's time requirement**

Growth-Rate Function for Size *n* Problems	Growth-Rate Function for Size 2*n* Problems	
1	1	None
log *n*	1 + log *n*	Negligible
n	2*n*	Doubles
n log *n*	2*n* log *n* + 2*n*	Doubles and then adds 2*n*
n^2	$(2n)^2$	Quadruples
n^3	$(2n)^3$	Multiples by 8
2^n	2^{2n}	Squares

Picturing Efficiency

- **The time required to process one million items by algorithms of various orders at the rate of one million operations per second**
- **See GrowthRateFunctionDemo in MiscellaneousPackage for another example**

Growth-Rate Function *g*	$g(10^6) / 10^6$
log *n*	0.0000199 seconds
n	1 second
n log *n*	19.9 seconds
n^2	11.6 days
n^3	31,709.8 years
2^n	$10^{301,016}$ years

Efficiency of ADT Bag Implementations

- **The time efficiencies of the ADT bag operations for two implementations, expressed in Big Oh notation**

Operation	Fixed-Size Array	**Linked**
`add(newEntry)`	O(1)	O(1)
`remove()`	O(1)	O(1)
`remove(anEntry)`	O(1), O(*n*), O(*n*)	O(1), O(*n*), O(*n*)
`clear()`	O(*n*)	O(*n*)
`getFrequencyOf(anEntry)`	O(*n*)	O(*n*)
`contains(anEntry)`	O(1), O(*n*), O(*n*)	O(1), O(*n*), O(*n*)
`toArray()`	O(*n*)	O(*n*)
`getCurrentSize(), isEmpty()`	O(1)	O(1)

In class activities

- Implement the following using BagInterface
  - FixedSizeArrayBag
  - LinkedBag
  - LinkedBagWithNodeMethods
    - Use Node.java – already completed
  - ResizableArrayBag
- All have been started, just need to complete methods
- Test using BagDemo.java
  - Comment/uncomment the lines to test various implementations
- Use code from the textbook if necessary, but note some changes to BagInterface