CSC 273 – Data Structures

Lecture 1- Bags

Generic Data Types

- Enable you to write a placeholder instead of an actual class type
- The placeholder is
  - A generic data type
  - A type parameter
- You define a generic class
  - Client chooses data type of the objects in collection.

Interface

```
// Write The specification for the
// interface Pairable here.
public interface Pairable <T>
{
    public T getFirst();
    public T getSecond();
    public void changeOrder();
}  //end Pairable
```

Example – `OrderedPair.java`

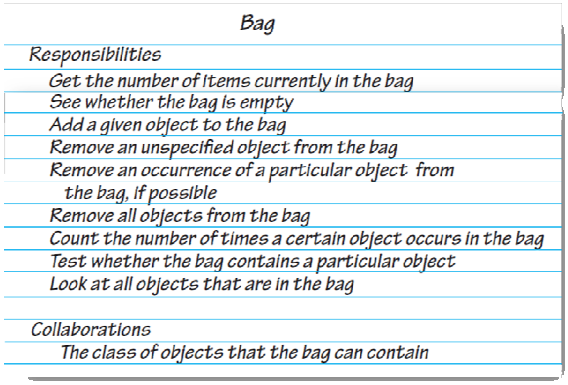
```
// OrderedPair  - an example of a class
//                implementing Pairable

public class OrderedPair <T>
                    implements Pairable<T>
{
  private T first, second;

  // NB; no <T> after the constructor name
  public OrderedPair(T firstItem, T secondItem) {
    first = firstItem;
    second = secondItem;
  }
```

```
// getFirst () - Return the first object in
//               this pair
public T getFirst() {
  return first;
}

// getSecond () - Return the second object in
//                this pair
public T getSecond() {
  return second;
}

// toString() - Returns string representation
//              of the object
public String toString() {
  return "(" + first + "< " + ")";
}
```

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

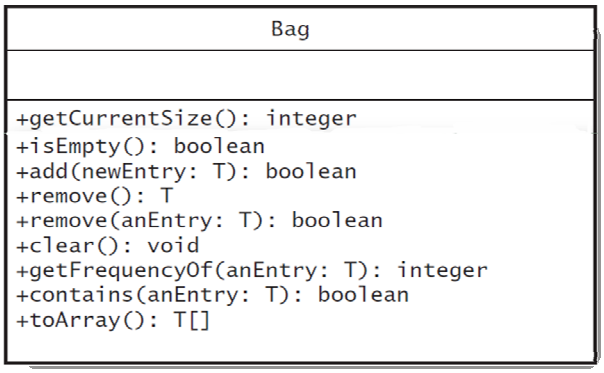
CRC (Class-Responsibility-Collaboration) Card



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

UML Notation



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

BagInterface.java

```
// An interface describing the opeations of a bag
// of objects

public interface BagInterface<T>  {
  //Gets the current number of entries in the bag
  public int getCurrentSize();

  // isEmpty() - Returns true if there's
  //             nothing in the bag
  //             Returns false if not
  public boolean isEmpty();
```

```
  // Adds the entry to the bag
  // Return true if successful, false if not
  public boolean add(T newEntry);

  // Removes one occurrence of an entry
  // to the bag
  // Returns it if successful, null if not
  public T remove();

  // Removes one occurrence of an entry
  // to the bag
  // Returns true if successful, false if not
  public boolean remove(T anEntry);

  // Removes everything from the bag
  public void clear();
```

```
  // Returns the number of such items in the bag
  public int getFrequencyOf(T anEntry);

  // Returns true if there is an item of this
  // type in the bag, false if not
  public boolean contains(T anEntry);

  // Returns everything in the bag as an array
  public T[] toArray();
}
```

Item.java

```
// A class of items for sale.

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
```

OnlineShopper.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++)     {
      // simulates getting item from shopper
      Item nextItem = items[index];
      shoppingCart.add(nextItem);
      totalCost
           = totalCost + nextItem.getPrice();
    } // end for

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

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

`OnlineShopper` Output

```
Sunflower seeds    $12.95
Bird bath  $44.99
Squirrel guard $15.47
Bird feeder$20.50
Total cost:    $93.91
```

Coin.java

```
// A class that represents a coin.
public class Coin
{
  private enum CoinSide {HEADS, TAILS}
  private CoinName myName;
  private int value; // in cents
  private int year;  // mint year
  private CoinSide sideUp;

  // Constructs an object for the coin having
  // a given value and mint year. The visible
  // side of the new coin is set at random.
```

```
  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 = CoinName.PENNY; break;
      } // end switch
```

```
        value = coinValue;
        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.
public Coin(CoinName name, int mintYear) {
    switch (name) {
       case PENNY: value = 1; break;
       case NICKEL: value = 5; break;
       case DIME: value = 10; break;
       case QUARTER: value = 25; break;
       case FIFTY_CENT: value = 50; break;
       case DOLLAR: value = 100; break;
       default: value = 1; break;
    } // end switch
```

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

```
// Returns the name of the coin.
public CoinName getCoinName() {
    return myName;
} // end getCoinName

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

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

```
// Returns "HEADS" if the coin is
// heads-side up; otherwise, returns "TAILS"
public String getSideUp() {
  return sideUp.toString();
} // end getSideUp

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

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

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

// Returns the coin as a string in the form
// value/year/side-up"
public String toString() {
    return value + "/" + year + "/" + sideUp;
} // end toString
```

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

    return result;
  } // end getToss
} // end Coin
```

Piggy Bank.java

```
// A class that implements a piggy bank by
// using a bag.
public class PiggyBank {
  private BagInterface<Coin> coins;

  public PiggyBank() {
    coins = new ArrayBag<>();
  } // end default constructor

  public boolean add(Coin aCoin) {
    return coins.add(aCoin);
  } // end add
```

```
  public Coin remove() {
    return coins.remove();
  } // end remove

  public boolean isEmpty() {
    return coins.isEmpty();
  } // end isEmpty
} // end PiggyBank
```

PiggyBankExample.java

```
// A class that demonstrates the class PiggyBank.

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

    addCoin(new Coin(1, 2010), myBank);
    addCoin(new Coin(5, 2011), myBank);
    addCoin(new Coin(10, 2000), myBank);
    addCoin(new Coin(25, 2012), myBank);
```

```
    System.out.println
            ("Removing all the coins:");
    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.getCoinName() + ".");
    else
      System.out.println("Tried to add a "
             + aCoin.getCoinName()
             + ", but couldn't");
  } // end addCoin
} // end PiggyBankExample
```

Output from PiggyBank Example

```
Added a PENNY.
Added a NICKEL.
Added a DIME.
Added a QUARTER.
Removing all the coins:
Removed a QUARTER.
Removed a DIME.
Removed a NICKEL.
Removed a PENNY.
All done. Removed 41 cents.
```

Using ADT
Like Using Vending Machine

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.

Observations about ADT Bag

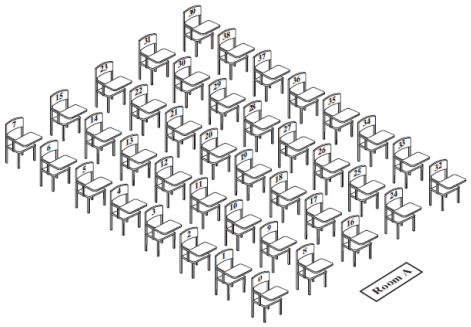
- Can perform only tasks specific to ADT
- Must adhere to the specifications of the operations of ADT
- Cannot access data inside ADT without ADT operations
- Use the ADT, even if don't know how data is stored
- Usable even with new implementation.

Set.java

```
//  An interface that specified a set of objects
public interface SetInterface <T> {
    public int getCurrentSize();
    public boolean isEmpty();


    // add() - adds an member to the set
    //         avoiding duplicates
    //         Returns true if successful
    //         Returns false if not
    public boolean add(T newEntry);
```

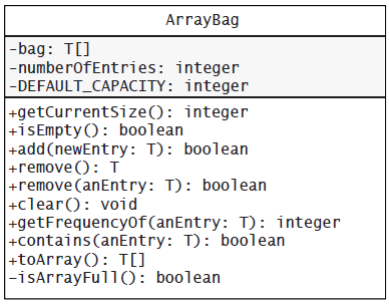
```
    // remove() - Removes anEntry from the set.
    //            Returns true if sucessful
    //            Returs if not.
    public boolean remove(T anEntry);

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

Fixed-Size Array to Implement the ADT Bag



Note – the desks are all in fixed positions!

Fixed-Size Array



BagArray.java

```
// A class of bags whose entries are stored in a
// fixed-size array.

public final class ArrayBag<T>
        implements BagInterface<T>  {
      private final T[] bag;
      private int numberOfEntries;
      private boolean initialized = false;
      private static final
                int DEFAULT_CAPACITY = 25;
      private static final
                int MAX_CAPACITY = 10000;
```

```
  // ArrayBag() - Creates an empty bag whose
  //              initial capacity is 25.
  public ArrayBag() {
    this(DEFAULT_CAPACITY);
  }
```

Making the Implementation Secure

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

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

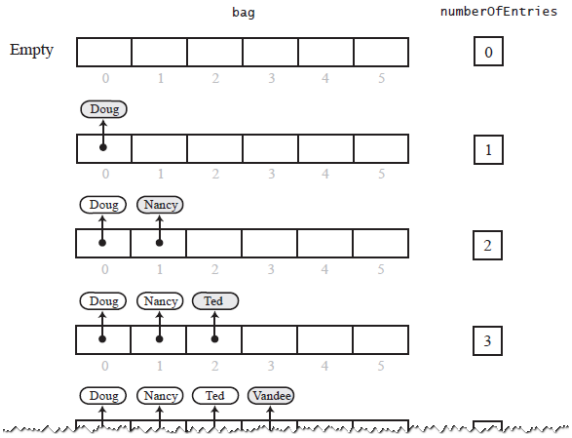
```
// ArrayBag() – Creates an empty bag having a
//              given capacity.
public ArrayBag(int desiredCapacity)    {
  if (desiredCapacity <= MAX_CAPACITY)    {
    // The cast is safe because the new array
    // contains null entries
    @SuppressWarnings("unchecked")
    // Unchecked cast
    T[] tempBag = (T[])
        new Object[desiredCapacity];
       bag = tempBag;
       numberOfEntries = 0;
       initialized = true;
  }
```

```
    else
      throw new IllegalStateException
              ("Attempt to create a bag "
                 + "whose capacity exceeds "
                 + "allowed maximum.");
  }
```

Making the Implementation Secure

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

Adding to the Fixed Array




```
// add() - Adds a new entry to this bag
public boolean add(T newEntry)     {
  checkInitialization();
  boolean result = true;

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

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

```
//  toArray() - retrieves all entries that are
//              in this bag in an array.
public T[] toArray()  {
  checkInitialization();

  // The cast is safe because the new array
  // contains null entries.
  @SuppressWarnings("unchecked")

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

```
    for (int index = 0;
             index < numberOfEntries;
                              index++)  {
            result[index] = bag[index];
    }

    return result;
  }
```

Core Methods

- Other core methods include:
  - **`clear()`**
  - **`remove()`**
  - **`isEmpty()`**
  - **`getCurrentSize()`**

Methods That Remove Items

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

Methods That Remove Items

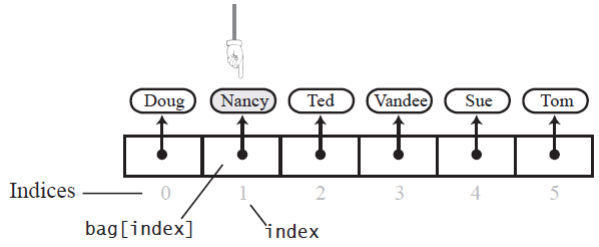
```
// remove() - removes one unspecified entry
//            from this bag, if possible.
//            Returns the removed entry, if the
//            removal was successful, or null.
public T remove()     {
  checkInitialization();
  T result = removeEntry(numberOfEntries - 1);

  return result;
} // end remove
```

Methods That Remove Items

```
// remove() - removes one occurrence of a given
//            entry from this bag.
//            Returns  True if the removal was
//            successful, or false if not.
public boolean remove(T anEntry)  {
  checkInitialization();
  int index = getIndexOf(anEntry);
  T result = removeEntry(index);

  return anEntry.equals(result);
}
```

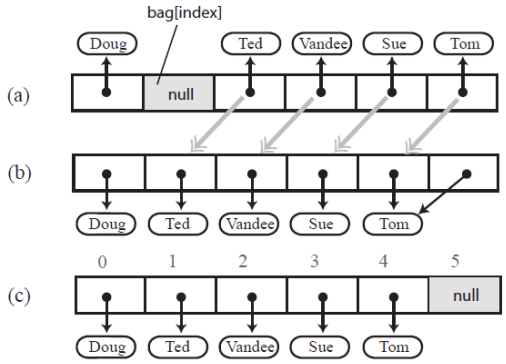
Methods That Remove Entries

Finding "Nancy"



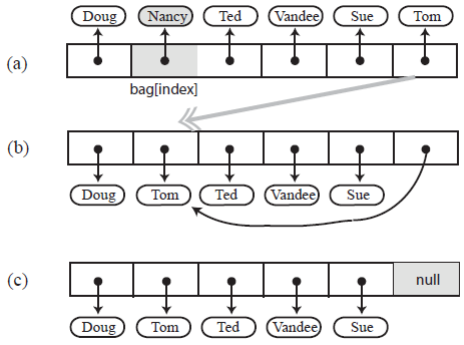
Methods That Remove Entries

Closing the gap?



Methods That Remove Entries

A Better way to close the gap



Methods That Remove Entries

```
// removeEntry - removes and returns the entry
//               at a given index within the
//               array.
//               If no such entry exists,
//               returns null.
// Precondition:
//           0 <= givenIndex < numberOfEntries.
// Precondition:
//       checkInitialization has been called.
private T removeEntry(int givenIndex)   {
  T result = null;
```

```
  if (!isEmpty() && (givenIndex >= 0))  {
    // Entry to remove
    result = bag[givenIndex];
    int lastIndex = numberOfEntries - 1;

    // Replace entry to remove with last entry
    bag[givenIndex] = bag[lastIndex];

    // Remove reference to last entry
    bag[lastIndex] = null;
    numberOfEntries--;
  }
  return result;
}
```

Methods That Remove Entries

```
// getIndexOf() - returns the index of the
//                entry, if located,
//                or -1 otherwise.
// Precondition:  checkInitialization has been
//                called.
private int getIndexOf(T anEntry)       {
  int where = -1;
  boolean found = false;
  int index = 0;
```

```
while (!found && (index < numberOfEntries))   {
    if (anEntry.equals(bag[index]))     {
      found = true;
      where = index;
    }
     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;
}
```

```
// isEmpty() - sees whether this bag is empty.
//             Returns true if this bag is
//             empty, or false if not.
public boolean isEmpty()  {
  return numberOfEntries == 0;
}


// getCurrentSize() - gets the current number
//                    of entries in this bag.
public int getCurrentSize()  {
  return numberOfEntries;
}
```

```
// getFrequency() - Counts the number of times
//                  given entry appears in this
//                  bag.
//                  Returns the number of times
//                  anEntry appears in this
//                  bag.
public int getFrequencyOf(T anEntry) {
  checkInitialization();
  int counter = 0;
```

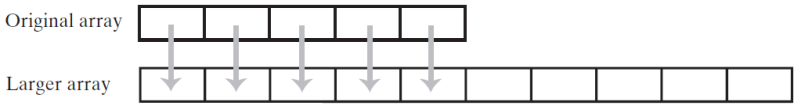
```
  for (int index = 0;
         index < numberOfEntries; index++)   {
    if (anEntry.equals(bag[index]))    {
      counter++;
    } // end if
  } // end for

        return counter;
}
```

```
// contains() - Determines if this bag contains
//              a given entry.
//              Returns true if this bag
//              contains anEntry, or false
//              otherwise.
public boolean contains(T anEntry)      {
  checkInitialization();
  return getIndexOf(anEntry) > -1; // or >= 0
}
```

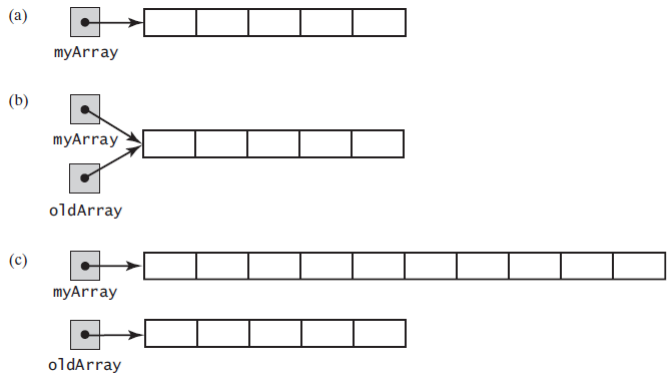
```
// checkInitialization() - throws an exception
//        if this object is not initialized.
private void checkInitialization()   {
  if (!initialized)
    throw new SecurityException
      ("ArrayBag object is not initialized "
         +   "properly.");
}
```

Using Array Resizing

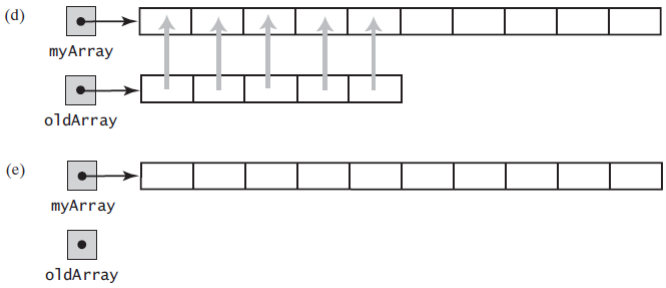
Resizing an array requires copies its contents to a larger second array



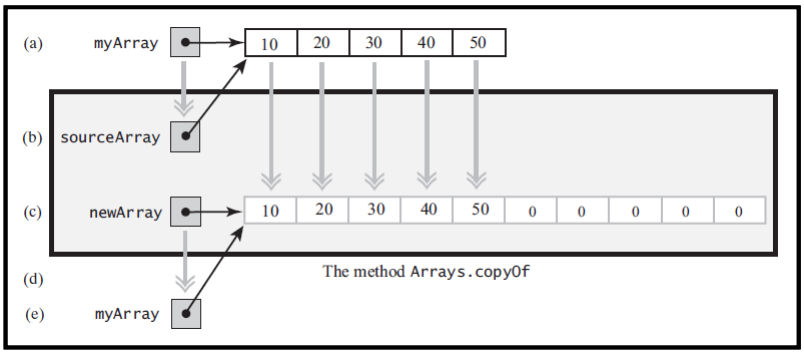
Using Array Resizing



Using Array Resizing



Using Array Resizing



`doubleCapacity()`

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

What Is an Iterator?

- An object that traverses a collection of data
- During iteration, each data item is considered once
  - Possible to modify item as accessed
- Should implement as a distinct class that interacts with the ADT

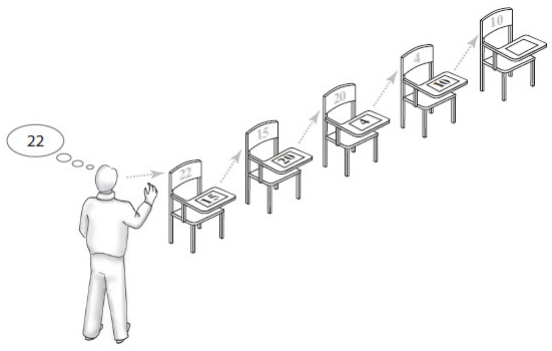
Problems with Array Implementation

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

Analogy

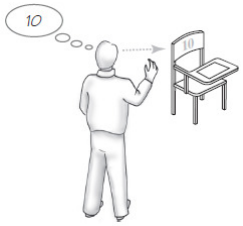
- Empty classroom
- Numbered desks stored in hallway
  - Number on **back** of desk is the "address"
- Number on **desktop** references another desk in chain of desks
- Desks are linked by the numbers

Analogy – A Chain of 5 Desks



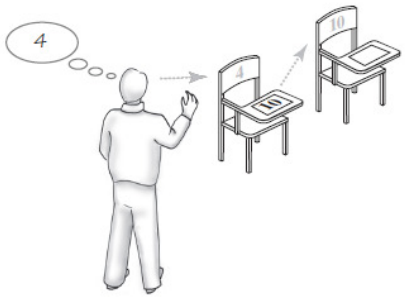
Forming a Chain by Adding to Its Beginning

One desk in the room



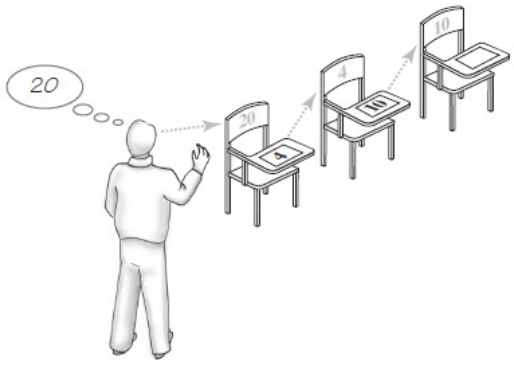
Forming a Chain by Adding to Its Beginning

Two linked desks, with the newest desk first



Forming a Chain by Adding to Its Beginning

Three linked desks, with the newest desk first.



Forming a Chain by Adding to Its Beginning

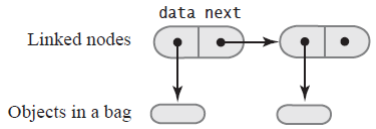
```
// Process the first student
newDesk represents the new student's desk
New student sits at newDesk
Instructor memorizes the address of newDesk
// Process the remaining students
while (students arrive)
{
    newDesk represents the new student's desk
    New student sits at newDesk
    Write the instructor's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

The Private Class **Node**

```
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;
  } // end constructor
} // end Node
```

The Private Class **Node**

Two linked nodes that each reference object data



An Outline of **LinkedBag**

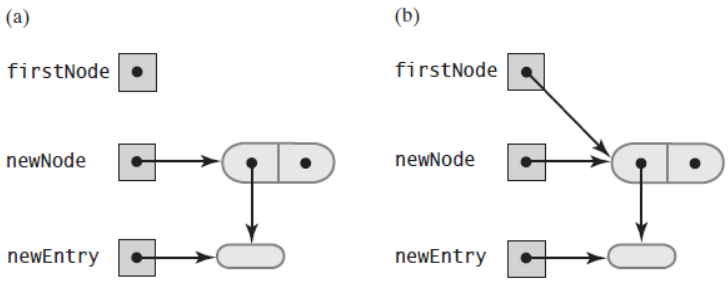
```
// A class of bags whose entries are stored in a
// chain of linked nodes.
// The bag is never full.

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

  public LinkedBag()    {
    firstNode = null;
    numberOfEntries = 0;
  }
```

An Outline of **LinkedBag**

```
  // Implementations of the public methods
  // in BagInterface go here
  … …

  private class Node {
    … …
  }
}
```

Beginning a Chain of Nodes

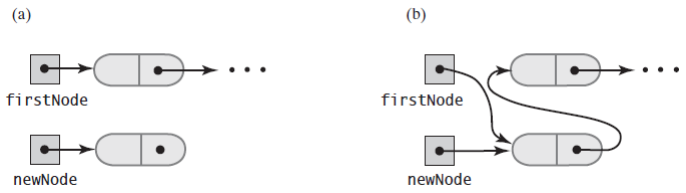
An empty chain and a new node



After adding a new node to a chain that was empty

Beginning a Chain of Nodes

Before adding a node at the beginning



After adding a node at the beginning

add()

```
// add() - Adds a new entry to this bag.
// newEntry is the object to be added
//   as a new entry
// 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);

  // Make new node reference rest of chain
  // (firstNode is null if chain is empty)
  newNode.next = firstNode;
```

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

  return true;
}
```

toArray()

```
// toArray() - Retrieves all entries that are in
//             this bag and creates 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")

    // Unchecked cast
    T[] result = (T[])new
            Object[numberOfEntries];
    int index = 0;
```

```
    Node currentNode = firstNode;
    while ((index < numberOfEntries)
           && (currentNode != null))  {
      result[index] = currentNode.data;
      index++;
      currentNode = currentNode.next;
    }
    return result;
  }
```

LinkedBagDemo1.java

```
//  A test of the constructors and the
//  methods add, toArray, isEmpty, and
//  getCurrentSize, as defined in the first draft
//  of the class LinkedBag.
public class LinkedBagDemo1 {
  // Tests on a bag that is empty
  public static void main(String[] args)  {
    System.out.println("Creating an empty bag.");
    BagInterface<String> aBag
                = new LinkedBag1<>();
    testIsEmpty(aBag, true);
    displayBag(aBag);
```

```
    // Tests on a bag that is not empty
    String[] contentsOfBag
         = {"A", "D", "B", "A", "C", "A", "D"};
    testAdd(aBag, contentsOfBag);
    testIsEmpty(aBag, false);
  }
```

```
// testIsEmpty() - tests the method isEmpty.
// Precondition: If the bag is empty, the
//          parameter empty should be true;
//          otherwise, it should be false.
private static void testIsEmpty
     (BagInterface<String> bag, boolean empty){
  System.out.print("\nTesting isEmpty with ");
  if (empty)
    System.out.println("an empty bag:");
  else
    System.out.println
           ("a bag that is not empty:");

  System.out.print("isEmpty finds the bag ");
```

```
  if (empty && bag.isEmpty())
    System.out.println("empty: OK.");
  else if (empty)
    System.out.println
       ("not empty, but it is: ERROR.");
  else if (!empty && bag.isEmpty())
    System.out.println
       ("empty, but it is not empty: ERROR.");
  else
    System.out.println("not empty: OK.");
}
```

```
// testAdd() -  tests the method add
private static void testAdd
     (BagInterface<String> aBag,
                  String[] content)    {
         … …
}

// displayBag() - tests the method toArray while
//                displaying the bag
private static void displayBag
          (BagInterface<String> aBag) {
        … …
}
```

`getFrequencyOf()` from `BagLinked`

```
// getFrequencyOf() - Counts the number of
//                    times a given entry
//                    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.data))  {
      frequency++;
    }

    counter++;
    currentNode = currentNode.next;
  } // end while

  return frequency;
}
```

contains()

```
// contains() - Returns true if this bag
//              contains a given entry.
//              Otherwise returns false
public boolean contains(T anEntry)      {
  boolean found = false;
  Node currentNode = firstNode;

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

  return found;
}
```

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.

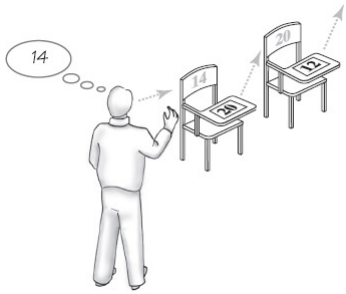
Removing an Item from a Linked Chain

Case 1

1. Locate first desk by asking instructor for its address.
2. Give address written on the first desk to instructor. This is address of second desk in chain.
3. Return first desk to hallway.

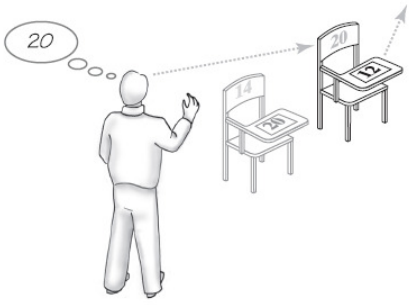
Removing an Item from a Linked Chain

A chain of desks just prior to removing its first desk



Removing an Item from a Linked Chain

A chain of desks just after removing its first desk



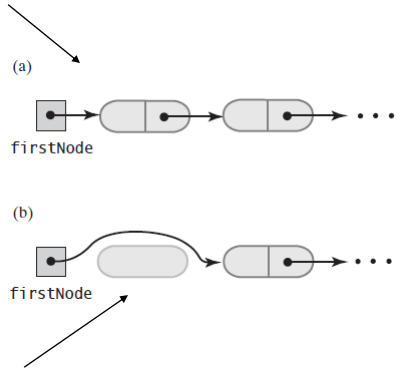
Removing an Item from a Linked Chain

Case 2

1. Move the student in the first desk to your former desk.
2. Remove the first desk using the steps described for Case 1.

Removing an Item from a Linked Chain

A chain of nodes just prior to removing the first node;



A chain of nodes just after removing the first node

getReferenceTo()

```
// getReferenceTo() - 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.data))
      found = true;
    else
      currentNode = currentNode.next;
  }

  return currentNode;
}
```

remove()

```
// remove() - Removes one unspecified entry
//            from this bag, if possible.
//            Returns  the removed entry, if
//            the removal was successful, or
//            null
public T remove()  {
  T result = null;
  if (firstNode != null)      {
  result = firstNode.data;

  // Remove first node from chain
    firstNode = firstNode.next;
    numberOfEntries--;
  }
  return result;
}
```

clear()

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

The **Node** Class

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

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

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

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

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

  private Node getNextNode() {
    return next;
  }

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

Class Within a Package

```
package BagPackage;

class Node<T> {
  private T   data;
  private Node<T> next;

  //The constructor's name is Node, not Node<T>
  Node(T dataPortion)  {
    this(dataPortion, null);
  }
```

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

  T getData() {
    return data;
  }

  void setData(T newData) {
    data = newData;
  }

  Node<T> getNextNode() {
    next = nextNode;
  }
}
```

When `Node` Is in Same Package

```
package BagPackage;
public class LinkedBag<T>
            implements BagInterface<T> {

  private Node<T> firstNode;
  … …

  public boolean add(T newEntry) {
    Node<T> new Node = new Node<T> (newEntry);
    newNode.setNextNode(firstNode);
    firstNode = newNode;
    numberOfEntries++;

    return true;
  }

 … …
}
```

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