```java
/**
 * This Bag class uses both generics and interfaces!
 * Reason: you get to leave types unspecified AND get control
 * over what is baggable and what is not.
 *
 * @author Chris Fernandes
 * @version 5/3/12
 *
 ************************************************************/
package proj5;
public class GenericContainableBag<E extends Containable> {

    private int size;           //# of elements in bag
    private Object[] contents;

    public GenericContainableBag(int capacity)
    {
        contents = new Object[capacity];
        size=0;
    }

    /**
     * add item to bag
     * @param value the item to add
     */
    public void add(E value)
    {
        if(size() == capacity())
                this.growDouble();

        contents[size]=value;
        size++;
    }

    /**
     * remove item from bag
     * @param value the item to remove
     */
    public void remove(E value)
    {
```

```
41
42          int found_position = find(value);
43          if (found_position>-1)
44          {
45              // move last element to fill the hole
46              contents[found_position]=contents[size-1
    ];
47              size--;
48          }
49      }
50
51      /**
52       * does bag contain item?
53       * @param value item to search for
54       * @return true if bag contains item, else false
55       */
56      public boolean contains(E value)
57      {
58          if (find(value) == -1)
59              return false;
60          else
61              return true;
62      }
63
64      /**
65       * is bag Empty?
66       * @return true if bag empty, else false
67       */
68      public boolean isEmpty()
69      {
70          if (size()==0)
71              return true;
72          else return false;
73      }
74
75      /**
76       * empty bag of contents
77       */
78      public void clear()
79      {
80          size=0;
81      }
82
83      /**
```

```
 84      *
 85      * @return number of items in bag
 86      */
 87     public int size()
 88     {
 89         return size;
 90     }
 91 
 92     /**
 93      * return bag contents as printable string
 94      */
 95     public String toString()
 96     {
 97         String answer = "{";
 98         int currentSize=this.size();
 99         for(int i=0; i < (currentSize - 1); i++)
    // take care of all but last one
100         {
101             answer = answer + contents[i] + ", ";
102         }
103         if (currentSize>0)  // as long as not empty
104         answer = answer + contents[(currentSize - 1
    )];
105         answer+= "}";
106         return answer;
107     }
108 
109     /**
110      * Getter for bag capacity
111      * @return how many items bag can hold
112      */
113     public int capacity()
114     {
115         return contents.length;
116     }
117 
118     /**
119      * remove a random element from the bag
120      * @return the random element removed
121      */
122     public E removeRandom()
123     {
124         E toReturn = grabRandom();
125         remove(toReturn);
```

```
126         return toReturn;
127     }
128 
129     /**
130      * grab a random element from the bag
131      * @return the element grabbed
132      */
133     @SuppressWarnings("unchecked")
134     public E grabRandom()
135     {
136         int rand = (int)(Math.random()*this.size());
137         E toReturn = (E)contents[rand];
138         return toReturn;
139     }
140 
141     /**
142      *
143      * @param otherBag another bag of the same type
    of item
144      * @return true if this bag has same elements as
     otherBag.
145      * Order doesn't matter.
146      */
147     public boolean equals(GenericContainableBag<E>
    otherBag)
148     {
149         if (this.size()!=otherBag.size())
150             return false;
151         else {
152             GenericContainableBag<E> thisCopy = this
    .clone();
153             GenericContainableBag<E> otherCopy =
    otherBag.clone();
154             while (!thisCopy.isEmpty()) {
155                 E someElement = thisCopy.
    removeRandom();
156                 if (!otherCopy.contains(someElement
    ))
157                     return false;
158                 else
159                     otherCopy.remove(someElement);
160             }
161             return true;
162         }
```

```
163     }
164 
165     /**
166      * @return exact copy of this bag.  Changes to
    copy
167      * do not affect the original, and vice versa.
168      */
169     public GenericContainableBag<E> clone()
170     {
171         GenericContainableBag<E> newBag = new
    GenericContainableBag<E>(this.capacity());
172         // since you can't make new E instances
    directly,
173         // I'll let you use the array's clone method
     here.
174         newBag.contents = this.contents.clone();
175         newBag.size = this.size();
176         return newBag;
177     }
178 
179     /**
180      * makes bag capacity equal to number of items
    in bag
181      */
182     public void trimToSize()
183     {
184         int currentSize = this.size();
185         Object[] newContents = new Object[
    currentSize];
186         for (int i=0; i<currentSize; i++) {
187             newContents[i]=this.get(i);
188         }
189         contents=newContents;
190     }
191 
192     /** make new bag contain all elements from this
193      *  bag and otherBag.  Return the new bag. this
    Bag
194      *  and otherBag are not altered in the process.
195      * @param otherBag the other bag
196      * @return the union of this bag and otherBag
197      */
198     public GenericContainableBag<E> union(
    GenericContainableBag<E> otherBag)
```

```
199     {
200         GenericContainableBag<E> newBag = this.clone
    ();
201         GenericContainableBag<E> temp = otherBag.
    clone();
202         while (!temp.isEmpty()) {
203             newBag.add(temp.removeRandom());
204         }
205         return newBag;
206     }
207 
208     /**
209      * return array index where item can be found.
    Return -1 if not found.
210      * @param value the item to search for
211      * @return -1 if not found.  Else, the array
    index where first one found.
212      */
213     private int find(E value)
214     {
215         int currentSize=this.size();
216         for (int i=0; i<currentSize; i++)
217         {
218             if (this.contents[i].equals(value))
219                 return i;
220         }
221         return -1;
222     }
223 
224     /**
225      * double the size of the internal array
226      */
227     private void growDouble()
228     {
229         Object[] newArray;
230         newArray = new Object[(contents.length) * 2
    ];
231         int oldSize = this.size();
232         for(int i=0; i < oldSize; i++)
233         {
234             newArray[i] = contents[i];
235         }
236         this.contents = newArray;
237     }
```

```
238
239     /**
240      * get the element at index i
241      * @param i index of internal array where sought
     item is stored
242      * @return the sought item
243      */
244     @SuppressWarnings("unchecked")
245     private E get(int i)
246     {
247         return (E)contents[i];
248     }
249 }
250
```