Disclaimer: *These notes may be distributed outside this class only with the permission of the Instructor.*

5.1   Comparable and Comparator

Listing 1: Date Class

```java
/*
 * Simple Date Class
 */
public class Date implements Comparable<Date> {
    private static final int[]
        DAYS = {0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
    private final int month;   // month (between 1 and 12)
    private final int day;     // day   (between 1 and DAYS[month]
    private final int year;    // year

    /**
     * Initializes a new date from the month, day, and year.
     * @param month the month (between 1 and 12)
     * @param day the day (between 1 and 28-31, depending on the month)
     * @param year the year
     * @throws IllegalArgumentException if the date is invalid
     */
    public Date(int month, int day, int year) {
        if (!isValid(month, day, year)) {System.out.println("Invalid date");}
        this.year = year;
        this.month = month;
        this.day = day;
    }
    /**
     * @return month return the month
     */
    public int getMonth(){
        return month;
    }
    /**
     * @return year return the year
     */
    public int getYear(){
        return year;
    }
    /**
     * @return day return the day
     */
    public int getDay(){
        return day;
    }

    // is the given date valid?
    private static boolean isValid(int m, int d, int y) {
        if (m < 1 || m > 12)      {return false;}
        if (d < 1 || d > DAYS[m]) {return false;}
        if (m == 2 && d == 29 && !isLeapYear(y)) {return false;}
        return true;
    }
```

```
50
51        /**
52         * Is year y a leap year?
53         * @return true if y is a leap year; false otherwise
54         */
55        private static boolean isLeapYear(int y) {
56            if (y % 400 == 0) {return true;}
57            if (y % 100 == 0) {return false;}
58            return y % 4 == 0;
59        }
60
61        /**
62         * Compare this date to that date.
63         * @return { a negative integer, zero, or a positive integer }, depending
64         *     on whether this date is { before, equal to, after } that date
65         */
66        @Override
67        public int compareTo(Date that) {
68            if (this.year  < that.year)  {return -1;}
69            if (this.year  > that.year)  {return +1;}
70            if (this.month < that.month) {return -1;}
71            if (this.month > that.month) {return +1;}
72            if (this.day   < that.day)   {return -1;}
73            if (this.day   > that.day)   {return +1;}
74            return 0;
75        }
76
77        /**
78         * Return a string representation of this date.
79         * @return the string representation in the format MM/DD/YYYY
80         */
81        @Override
82        public String toString() {
83            return month + "/" + day + "/" + year;
84        }
85
86    }
```

Listing 2: Compare two date objects by month

```
1  import java.util.Comparator;
2  public class MonthComparator implements Comparator<Date> {
3      @Override
4      public int compare(Date d1, Date d2) {
5          if (d1.getMonth() < d2.getMonth()) {return - 1};
6          if (d1.getMonth() > d2.getMonth()) {return + 1};
7          return 0;
8      }
9  }
```

Listing 3: Compare two date objects by day

```
1  import java.util.Comparator;
2  public class DayComparator implements Comparator<Date> {
3      @Override
4      public int compare(Date d1, Date d2) {
5          if (d1.getDay() < d2.getDay()) {return -1};
6          if (d1.getDay() > d2.getDay()) {return +1};
7          return 0;
8      }
9  }
```

Listing 4: Test Date Class

```
1   /**
2    * Date class unit test
3    */
4   package date;
5   import java.util.Arrays;
6   public class DateTest {
7       public static void main(String[] args){
8           Date[] date = new Date[3];
9           date[0] = new Date(5,21,2014);
10          date[1] = new Date(10,28,2013);
11          date[2] = new Date(1,25,2011);
12          //sort the dates by non-decreasing order
13          Arrays.sort(date,new DayComparator());
14          for(Date d:date ){
15              System.out.println(d);
16          }
17          //sort the dates by month order
18          Arrays.sort(date,new MonthComparator());
19          for(Date d:date ){
20              System.out.println(d);
21          }
22          //sort the dates by day order
23          Arrays.sort(date,new DayComparator());
24          for(Date d:date ){
25              System.out.println(d);
26          }
27      }
28  }
```

Table 5.1: Output

Before Sort	After sort	Sort by month	Sort by day
5/21/2014	1/25/2011	1/25/2011	5/21/2014
10/28/2013	10/28/2013	5/21/2014	1/25/2011
1/25/2011	5/21/2014	10/28/2013	10/28/2013

5.2   Implement a Set using Bag class

A set is a container that contains no duplicate elements. We can easily extend the Bag and create a set container. We only have to override the insert method of the Bag. When we insert a new item, discard the item if it exists in the set.

Listing 5: Set Class

```
1   public class MySet<E> extends Bag<E>{
2           public void insert(E item){
3                   if(!contains(item)){
4                           super.insert(item);
5                   }
6           }
7   }
```

5.3   Cloning Objects

A clone is an identical copy of the original object.

5.3.1   Reference

Listing 6: Reference

```
1  Bag<Integer> bag1 = new Bag();
2  bag1.insert(100);
3  bag1.insert(200);
4  bag<Integer> bag2 = bag1;
```
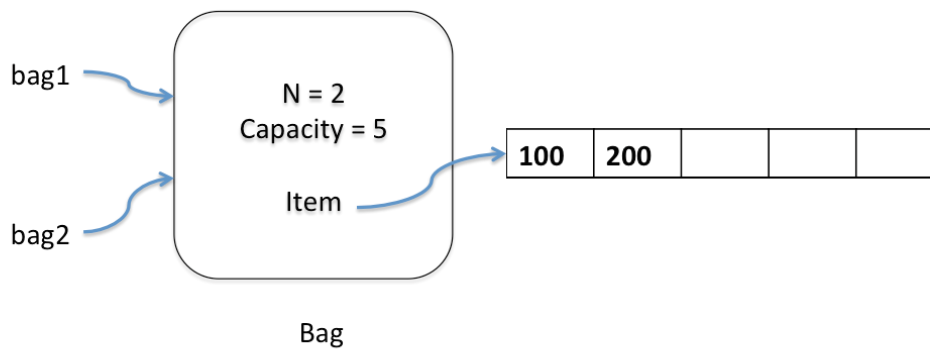

Figure 5.1: Reference Copy

bag1 and bag2 reference the same object. ba1==bag2 returns true.

5.3.2   Shallow Copy

Listing 7: Shallow Copy

```
1  @Override
2  public Bag<E> clone() throws CloneNotSupportedException{
3          return (Bag<E>) super.clone();
4  }
```

Listing 8: Shallow Copy

```
1  Bag<Integer> bag1 = new Bag();
2  bag<Integer> bag2 = bag1.clone();
3  bag1.insert(100);
4  bag1.insert(200);
```
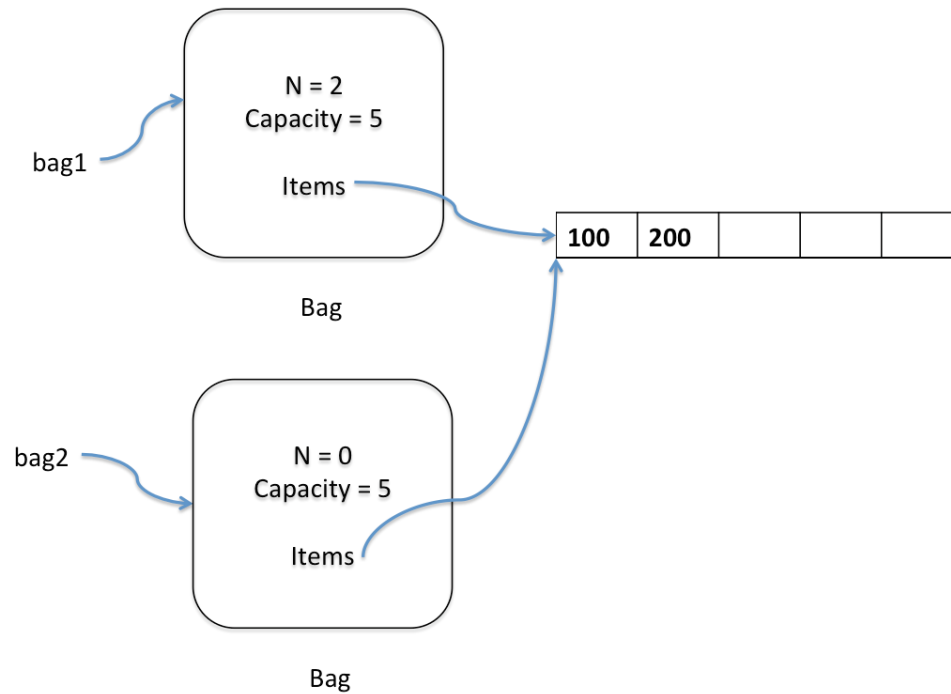
Figure 5.2: Shallow Copy

ba1 and bag2 reference different Bag objects. But those two object share the same array items. bag1 ==
bag2 return false, but bag1.equals(bag2) returns true if both have same number of items.

5.3.3 Deep Copy

Listing 9: Deep Copy

```
1  @Override
2  public Bag<E> clone() throws CloneNotSupportedException{
3          Bag<E> b = (Bag<E>) super.clone();
4          b.items = (E[]) new Object[capacity];
5          for(int i = 0; i <N; i++){
6                  b.items[i] = items[i];
7          }
8          return b;
9  }
```

Listing 10: Shallow Copy

```
1  Bag<Integer> bag1 = new Bag();
2  bag1.insert(100);
3  bag1.insert(200);
4  bag<Integer> bag2 = bag1.clone();
```
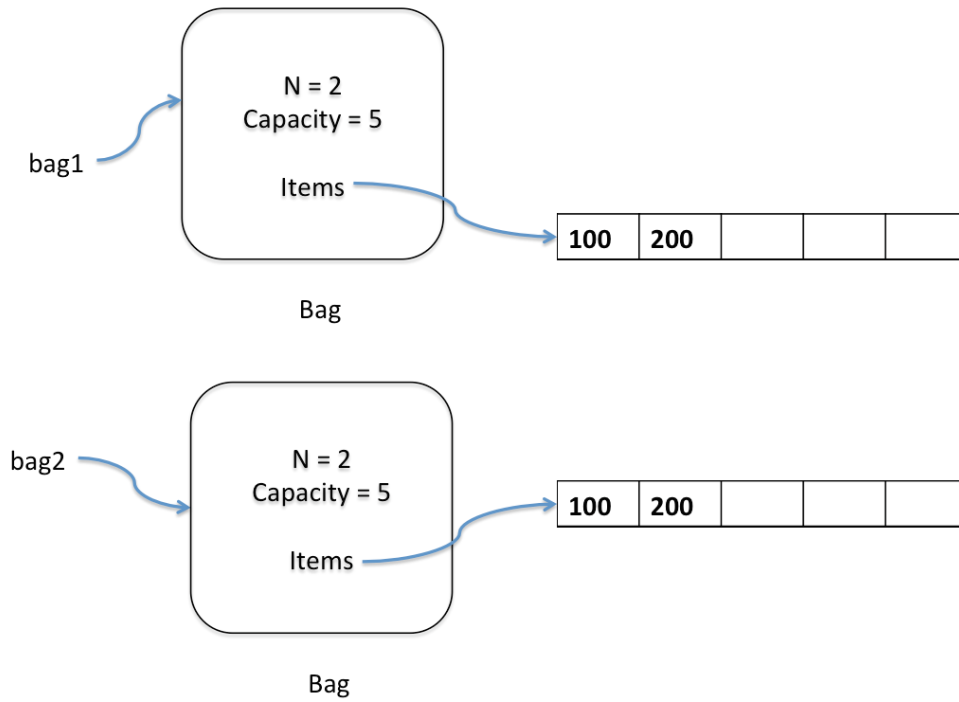
Figure 5.3: Deep Copy

bag1 and bag2 reference two different objects.