Introduction to Computer Science with Java

Community College of Philadelphia Computer Science

Chapter 19 – Iterative Sorting Techniques; Sorting Objects

This chapter discusses iterative techniques for sorting data, including the Bubble Sort, Selection Sort and Insertion Sort, along with techniques for comparing and sorting objects in Java. Object and array serialization for object stream I/O is also discussed.

Chapter Learning Outcomes

Upon completion of this chapter students should be able to:

- describe the transitive property of equality and inequality and how this is related to the notion of a comparable set and a comparable data typs in computer programming.
- describe how a bubble sort works and create Java software to implement a bubble sort.
- describe how a selection sort works and create Java software to implement a selection sort.
- describe how an insertion sort works and create Java software to implement an insertion sort.
- describe the concepts of temporal complexity and spatial complexity of an algorithm
- Roughly describe the temporal complexity and spatial complexity of a bubble sort, selection sort and insertion sort and how they compare to one another.
- describe how to compare instances of a class of objects to one another and the concept of the natural ordering of a class of objects.
- describe how to write an instance of an object as a line in a CSV file, and create Java code to do so.
- describe the concept of object serialization and file I/O for objects based on object serialization, including serializing an array.
- create simplified java software for object file I/O, including file I/O that handles array serialization and deserialization using the *ObjectOutputStream* class *writeObject()* method and the *ObjectInputStream* class *readObject()* method.

19.1 Sorting Data: The Bubble Sort

All methods for sorting data are based on the transitive nature of equality and inequality relationships:

If A <= B and B <= C, then A <= C.

The statement above combines two transitive properties:

- the **transitive property of equality** (If A = B and B = C, then A = C)
- the **transitive property of inequality** (if A < B and B < C, then A < C).

If we have a set of values and a way to compare the values in a *less than*, *equal to*, or *greater than* manner, then the values have a transitive relationship. We call such a set of data a **comparable set**. In Java, primitive data types are comparable, as are any class of objects that implements the *Comparable* interface, such as the String class. They are known as **comparable data types**. This does not mean that the data types can be compared to one another, but that instances of each of these data types can be compared to other instances of the same data type.

We can use the transitive nature of comparable data to see if a list is in the correct order. If the first and second items in a list are in the correct order, and the second and third items are in the correct order, and so on, throughout the entire list, then by the transitive nature of the data, all items in the list must be in the correct order relative to one another, so the entire list is in the correct order.

We can also use the transitive nature of comparable data as the basis for a technique to sort a list. We begin by comparing the first two items in the list. If they are not in the correct order we swap them, if they are in the correct order we leave them alone. We then move on to the next pair, which would be the second and third items in the list, and repeat the process. We continue comparing consecutive pairs through the entire list.

When we make it all the way through the list, we have completed one *pass* through the list. We continue to make passes through the list until we can make one complete pass without swapping anything. When this happens, we will know that each pair of items in the list is in the correct order, and therefore, by the transitive nature of the data, the entire list is in the correct order.

This technique for sorting – making passes through the list and putting consecutive pairs of items in the correct order as we do so – is known as a **bubble sort**. The name comes from the fact that items in the list bubble into the correct position, getting closer to where they belong in the list with each pass through the list.

We saw the bubble sort in chapter 6 in CSCI 111, and again in chapter 15 in CSCI 112. Included here is here the example of a bubble sort from chapter 6.

```
Pseudocode for a Bubble Sort on an Array of Data
Start Bubble Sort – sort in ascending order (lowest to highest)

a[n] is an array with n elements. The data type of the array must be a comparable data type.

boolean swapped;   // a boolean variable to keep track of when array values are swapped

do    //the outer post-test loop will  repeat another pass through the list when swapped in true
{
    set swapped to false before each pass
    for ( i=0; i < a.length – 1; i++)    // a pass through the array to the second to last element
    {
        if ( a[i+1] < a[i])          // if the two items are out of order
        {
            swap the two items     (a[i] and a[i+1])
            set swapped to true
        }  // end if

    } // end for

} while (swapped);   // the outer loop will repeat if swapped is true – another pass

Stop Bubble Sort
```

The table below illustrates a bubble sort on an array of integers:

{3 7 1 9 4 8 2 5 6}

On the left, we can see what happens during one pass through the array. Starting with i=0, A[i+1] is compared to A[i]. If A[i+1] < A[i], they are swapped, otherwise, nothing happens.

On the right, we see what the list looks like after each pass. The sort stops when a pass is made without swapping anything, which means the list is in order.

Illustration of a Bubble Sort

Inside one pass through the array		After each pass through the array
3 7 1 9 4 8 2 5 6 ↑ ↑	**i=0** 7 > 3 — nothing happens	
3 7 1 9 4 8 2 5 6 ↑ ↑	**i=1** 1 < 7 — swapped	3 7 1 9 4 8 2 5 6 swapped = true
3 1 7 9 4 8 2 5 6 ↑ ↑	**i=2** 9 > 7 — nothing happens	3 1 7 4 8 2 5 6 9 swapped = true
3 1 7 9 4 8 2 5 6 ↑ ↑	**i=3** 4 < 9 — swapped	1 3 4 7 2 5 6 8 9 swapped = true
3 1 7 4 9 8 2 5 6 ↑ ↑	**i=4** 8 < 9 — swapped	1 3 4 2 5 6 7 8 9 swapped = true
3 1 7 4 8 9 2 5 6 ↑ ↑	**i=5** 2 < 9 — swapped	1 3 2 4 5 6 7 8 9 swapped = true
3 1 7 4 8 2 9 5 6 ↑ ↑	**i=6** 5 < 9 — swapped	1 2 3 4 5 6 7 8 9 swapped = false the list is in order
3 1 7 4 8 2 5 9 6 ↑ ↑	**i=7** 6 < 9 — swapped	
3 1 7 4 8 2 5 6 9	result of first pass	

The bubble sort is a correct sorting technique, but it is not a very efficient one. We can roughly measure the efficiency of a sorting technique by the number of comparisons it needs to make to sort the list.

If a list has 100 items, then a bubble sort needs to make 99 comparisons in each pass through the list. If the item that should be first in the properly sorted list is last in the original list, then we would need to make 100 passes until the list is in the correct order. Of course, if we start with a list in the correct order, then we only need to make one pass through the list.

In general, if a list has *n* items, we might need to make *n*(n -1)* comparisons to complete a bubble sort. On the average, we will need to make $\frac{n*(n-1)}{2}$ comparisons to complete a bubble sort.

Next we will look at two other sorting techniques, *insertion sort* and *selection sort*, and see how they compare to the bubble sort.

19.2 Selection Sort

Imagine that you are holding a group of five playing cards in your hand. The cards are not in any particular order. We wish to put them in order according to their rank – all of the twos, then all of the threes, and so on. We can do this by selecting the lowest card in the hand and placing it on the table, then selecting the lowest card from the remaining cards in the hand and placing it on table after the first card, and so on, until we have a set of cards on the table in the correct order. This technique is known as a selection sort. It illustrated on the next page.

In a **selection sort**, we repeatedly find the minimum item in a list, take it out of the list, and add it as the next item in a new list.

The process of finding the minimum item in a list requires us to go through the entire list, making *n-1* comparisons as we do so. In the selection sort, each time we do this the list is one item shorter. The first time **we go through the list, it will have n items. The next time it will have *n-1* items, then *n-2* items, and so on, until there is only one item in the list. On the average, there will be $\frac{n}{2}$ items** in the list each time we need to select an item to be moved. So, selection sort requires $(n-1) * \frac{n}{2}$ comparison operations to complete the sort. This equals $\frac{n*(n-1)}{2}$, the same as the average case for the bubble sort.

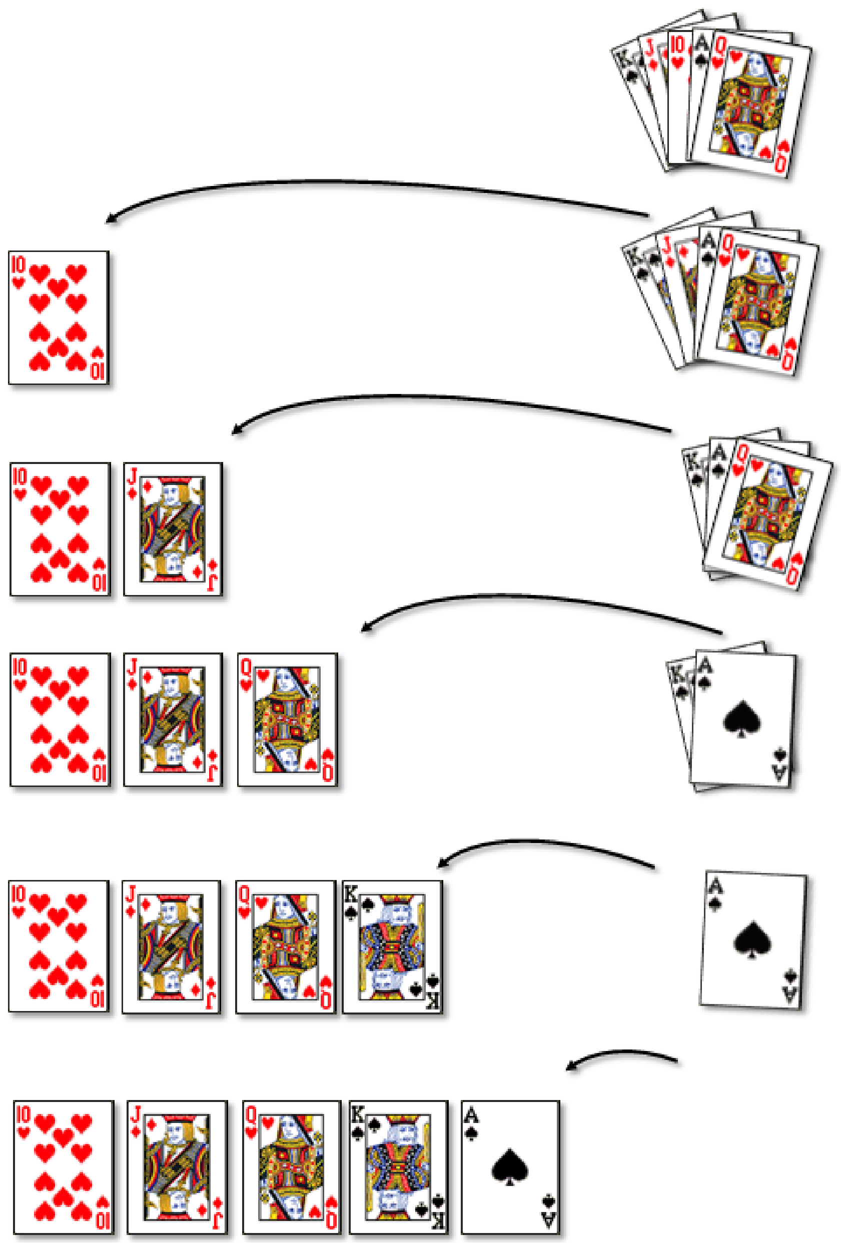
Algorythmics

There are many Web sites with ways to illustrate various sorting algorithms visually, such as http://www.sorting-algorithms.com One very interesting attempt to illustrate sorting algorithms is the award winning **AlgoRythmics Project** at *Sapientia University*, in Tirgu Meres, Romania. They have created a series of folk dance performances to illustrate different sorting algorithms, which are available on the Web.

- Bubble Sort with Hungarian ("Csángó") folk dance:
  http://www.youtube.com/watch?v=lyZQPjUT5B4
- Insertion sort with Romanian folk dance:
  http://www.youtube.com/watch?v=ROalU379l3U
- Selection Sort with Gypsy folk dance:
  http://www.youtube.com/watch?v=Ns4TPTC8whw
- The complete AlgoRythmics YouTube Channel
  http://www.youtube.com/user/AlgoRythmics
- The AlgoRythmics website:
  http://algo-rythmics.ms.sapientia.ro

Image of a Selection Sort

The image on this page image shows a selection sort for a hand of playing cards. We repeatedly select the lowest card, then move it to the next spot in the new list of cards. The *work* of the sort happens when we select the lowest card by searching the original list of cards. Compare this to the *selection sort*, above, to understand the difference between the two.



19.3 Algorithmic Complexity

So, which is more efficient— the bubble sort or the selection sort? In the best case, the bubble sort only needs one pass, which means *n-1* comparisons. In the worst case, the bubble sort needs *n*(n -1)* comparisons. The selection sort always requires $\frac{n*(n-1)}{2}$ comparisons. The average case for the bubble sort is $\frac{n*(n-1)}{2}$ comparisons, the same as the selection sort. In general, they have roughly the same efficiency, although occasionally the bubble sort is faster and occasionally the bubble sort is slower.

The time that it takes an algorithm to run as its dataset gets bigger is known as the **temporal complexity** of the algorithm. The bubble sort and the selection sort have roughly the same temporal complexity.

However the selection sort described above requires twice as much space as the bubble sort. In the selection sort as described above, the data is moved from one copy of the list to another copy of the last. Each copy of the list requires its own storage space. If we are using arrays, for example, then the selection sort requires two arrays.

The bubble sort moves data around within the original array. This is known as an in-place sorting technique, which does not require duplicate storage space.

The amount of space an algorithm uses is known as the algorithm's **spatial complexity**. The bubble sort demonstrates similar temporal complexity to the selection sort shown above, but exhibits better spatial complexity. We will examine algorithmic complexity in more detail in the next chapter.

In-Place Selection Sort

It is possible to complete a selection sort in-place, as described in the pseudo code below. The in-place version of the selection sort has the same spatial complexity as the bubble sort and is the most common version of a selection sort.

In a selection sort, the original list gets smaller as the new list gets bigger, so, to complete an in-place selection sort, we keep the new list and the old list both in the original array, using the space left over from the shrinking old list to store the growing new list. We do this by moving the first item in the array into the place where we find the minimum item in the first pass through the old list, and putting the minimum in the first place in the array; then moving the second item in the array into the place where we find the minimum item in the second pass through the old list, and putting the new minimum in the second place in the array; and so on, until the sort is complete. We start with the first spot in the array, and iteratively swap the data in each spot with the minimum in the remainder of the array.

Pseudocode for an In-Place Selection Sort on an Array of Data
Start Selection Sort – sort in ascending order (lowest to highest)

a[n] is an array with n elements. The data type of the array must be a comparable data type.

```
int spot; // location in the array where we will insert the minimum from the remainder of the list
int minimum; // location of the minimum value in the remainder of the list
for ( spot=0; spot < a.length; spot++)       // the outer loop – if the list has n items, we  make n passes
 {
        // find the minimum value in the remainder of the list

        minimum = spot;   // initialize the minimum to be the first value in the remainder of the list

        for (i = spot+1; i < a.length; i++)    // i iterates from spot to the end of the list – one pass
        {
           if  a[i] < a[minimum]
              minimum = i;   // location of the minimum so far for this pass through the remainder of the list
        } // end for i – end of one pass through the remainder of the list

 swap  a[spot] and a[minimum]; swap the value in spot with this pass's minimum

 } // end for spot

Stop Selection Sort
```

Illustration of a Selection Sort

	spot = 0 (the first position)
3 7 1 9 4 8 2 5 6 ↑	find the minimum spot = 0, min is in a[2]
1 7 3 9 4 8 2 5 6 ↑	swap the minimum with the value in A[spot]
1 **7** 3 9 4 8 2 5 6 ↑	increment spot and find the new minimum spot = 1, min is in a[6]
1 **2** 3 9 4 8 7 5 6 ↑	swap the minimum with the value in a[spot]
1 **2** 3 9 4 8 7 5 6 ↑	increment spot and find the new minimum spot = 2, min is in a[2]

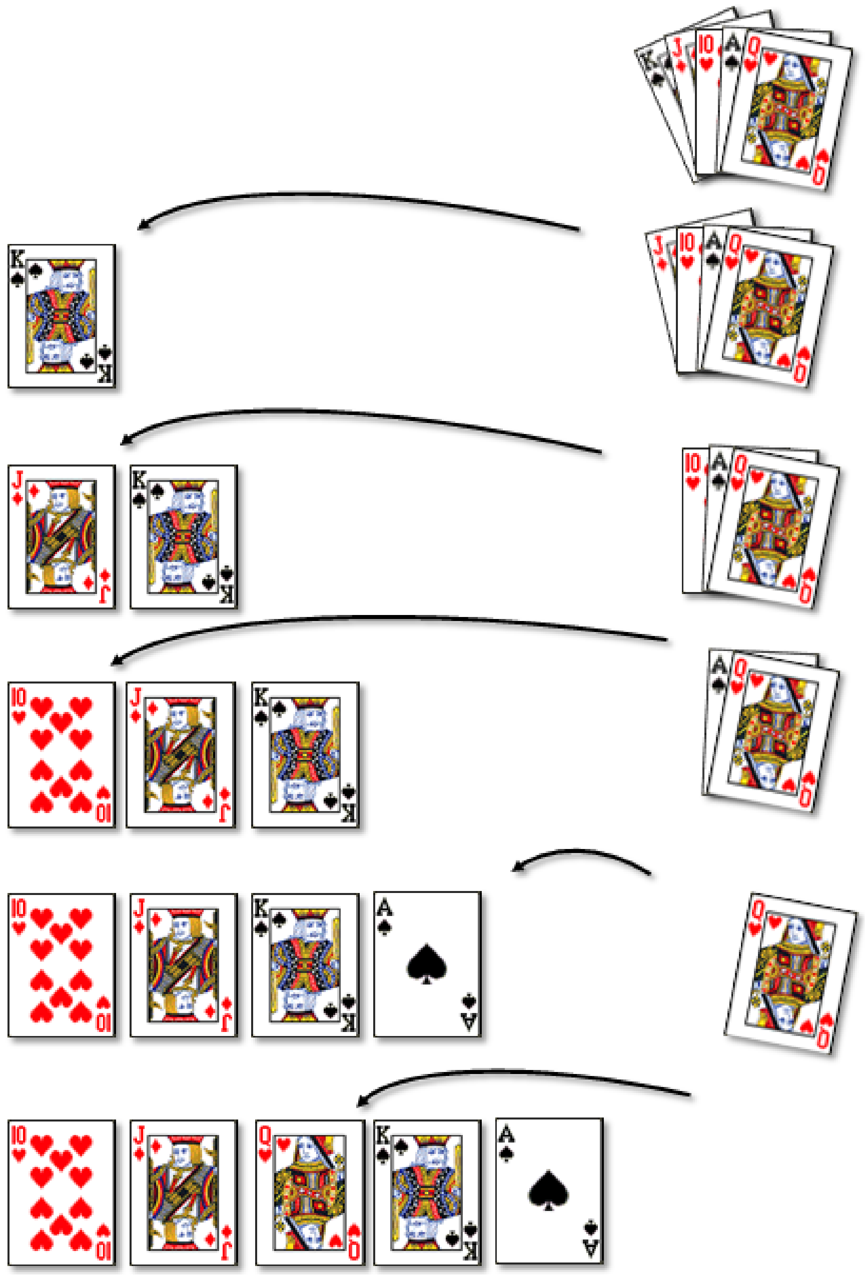
```
1  2  3  9  4  8  7  5  6        no swap needed; the minimum is in a[spot]
      ↑
1  2  3  9  4  8  7  5  6        increment spot and find the new minimum
         ↑                       spot = 3,  min is in a[4]
1  2  3  4  9  8  7  5  6        swap
         ↑
1  2  3  4  9  8  7  5  6        increment spot and find the new minimum
            ↑                    spot = 4,  min is in a[7]
1  2  3  4  5  8  7  9  6        swap
            ↑
1  2  3  4  5  8  7  9  6        increment spot and find the new minimum
               ↑                 spot = 5,  min is in a[8]
1  2  3  4  5  6  7  9  8        swap
               ↑
1  2  3  4  5  6  7  9  8        increment spot and find the new minimum
                  ↑              spot = 6,  min is in a[6]
1  2  3  4  5  6  7  9  8        no swap needed; the minimum is in a[spot]

                  ↑
1  2  3  4  5  6  7  9  8        increment spot and find the new minimum
                     ↑           spot = 7,  min is in a[8]
1  2  3  4  5  6  7  8  9         swap

                     ↑
1  2  3  4  5  6  7  8  9        increment spot and find the new minimum
                        ↑        spot = 7,  min is in a[8]
1  2  3  4  5  6  7  8  9        done – end of list reached
```

19.4 Insertion Sort

Insertion sort is a sorting technique similar to selection sort: both techniques involve searching a dataset. In selection sort, we searched the original list to find the minimum item in the list, then added it as the next item in a new list. In the insertion sort, we remove the next item from a list and search a new list to find its correct place in the new list. In one sense, selection sort and insertion sort are the opposite of each other: selection sort makes comparisons when removing an item from the original list, whereas insertions sort makes comparisons when adding an item to the new list.

Image of an Insertion Sort

The image on this page image shows an insertion sort for a hand of playing cards. We simply select the next card, then insert it in the correct spot in the new list of cards. The *work* of the sort happens when we search the new list of cards to find the right spot to insert the card being moved. Compare this to the *selection sort*, above, to understand the difference between the two.



In-Place Insertion Sort

Just as with selection sort, the original list shrinks as the new list grows, so we can use this phenomena to complete an insertion sort in-place, just as we did for selection sort. The pseudo code for an in-place insertion sort is included below.

An in-place Insertion sort has the original list in the right side of an array and the sorted list in the left side of the array. To start with, the first element in the array (a[0]fro example) is considered a one element sorted list and the entire rest of the array (a[1] to a[n]) is considered the original unsorted list. The algorithm works through the array from a[1} upward, (in order from left to right), removing each item from the original list one at a time. It then works backwards through the old list in the left part of the array, moving each item one spot to the right as it does so, until it find where the item removed from the original list belongs. It then inserts the item in that spot, which has been emptied by moving things to the right. As the algorithm progresses, the original unsorted list gets smaller and the new sorted list gets larger, until everything is in the new list.

```
Pseudocode for an In-Place Insertion Sort on an Array of Data
Start Insertion Sort – sort in ascending order (lowest to highest)

a[n] is an array with n elements. The data type of the array must be a comparable data type.

int value ;    // the next value from the unsorted list to be inserted into the sorted list
int i;         // i is a pointer to an item in the unsorted list
int j;         // j is a pointer to an item in the sorted list;  originally the sorted list is just a[0]

for ( i=1; i < a.length; i++)      iterate the entire list
  {
  value = a[i];  // the next item from the original list, which will be added to the new list
  j = i -1;       // set the pointer to the beginning of the unsorted list. We will work down (left) from here

  while ( j => 0 and a[j] > value)  // go from the current item backwards in the array to find where it goes
     {                              // keep going while a[j] > value, in other words, stop when [j] <=
                      value, // this is the spot where the value should be inserted.
    a[j+1] = A[ j]    //move the current item to the right; this is not the insertion spot yet
     j =j-1;          // go left to the next item in the array

     } // end while   -- the loop continues, moving each array element to the right
      // until the spot for insertion is reached

  a[i+1] = value;     \\ insert value in the correct spot.
                      \\ We add one because the empty correct spot is after this

  } // end for loop   Process the next item from the original list each time through the loop.

Stop Insertion Sort
```

Illustration of an Insertion Sort

```
                          i = 1;  unsorted list starts at a[1]; a[1] = 7;
3 7 1 9 4 8 2 5 6         7 is in the correct spot, no movement needed
  ↑
3 7 1 9 4 8 2 5 6         increment i
    ↑                     i = 2;  A[2 ] = 1;
1→3→7 9 4 8 2 5 6         move values and put 1 in the correct spot in the new list;
↑
1 3 7 9 4 8 2 5 6         increment i
      ↑                   i = 3;  A[3 ] = 9;
1 3 7 9 4 8 2 5 6         9 is in the correct spot, no movement needed
      ↑
1 3 7 9 4 8 2 5 6         increment i
        ↑                 i = 4;  A[4 ] = 4;
1 3 4→7→9 8 2 5 6         move values and put 4 in the correct spot in the new list;
    ↑
1 3 4 7 9 8 2 5 6         increment i
          ↑               i = 5;  A[5 ] = 8;
1 3 4 7 8→9 2 5 6         move values and put 8 in the correct spot in the new list;
        ↑
1 3 4 7 8 9 2 5 6         increment i
            ↑             i = 6;  A[6 ] = 2;
1 2→3→4→7→8→9 5 6         move values and put 2 in the correct spot in the new list;
  ↑
1 2 3 4 7 8 9 5 6         increment i
              ↑           i = 7;  A[7 ] = 5;
1 2 3 4 5→7→8→9 6         move values and put 5 in the correct spot in the new list;
        ↑
1 2 3 4 5 7 8 9 6         increment i
                ↑         i = 8;  A[8 ] = 6;
1 2 3 4 5 6→7→8→9         move values and put 6 in the correct spot in the new list;
          ↑
1 2 3 4 5 6 7 8 9         done – end of list reached
```

A variation of the insertion sort is useful for putting data into sorted order as it is read in from an external source. As unsorted data items come into an array, we can use the same technique as the insertion part of an insertion sort to put the items in the correct position in the array.

How does the speed of insertion sort compare to that of selection sort and bubble sort? The in-place version of Insertion sort requires n-1 passes, but the number of comparisons in each pass depends on the entropy of the data. **Data entropy** is a general measure of the randomness of a dataset – how much it is in order or out of order. There are specialized forms of data entropy, such as *Shannon entropy*, beyond a study of data entropy is beyond what we cover in this course. In the worst case, the insertion sort will need to perform the same number of comparisons as the selection sort, ——— which is the average number of comparisons in a bubble sort.

There are variations of the bubble sort that could speed it up a little, but among the techniques we have seen, insertion sort will, on the average, have fewer comparisons than bubble sort or selections sort. However, insertion sort moves data items as it searches for a correct spot in an array, which selection sort does not do. The amount of movement needed depends on the data entropy.

So, in the final analysis, all three of these iterative sorting techniques – bubble sort, selection sort, and insertion sort – seem to exhibit similar temporal and spatial complexity. In the next chapter, after we see some recursive sorting techniques, we will return to the question of how the temporal complexity of sorting methods compares to one another.

19.5 Sorting Objects

You should be able to read data in from a data file, sort the data, then write it out again to another data file, or to the same file. This is relatively easy for integer data or for String data, but what about sorting an array of objects, each with a collection of several properties? What about saving and retrieving an array of objects? Are there simplified ways to do these things, or do we need to manually work with each property of each object in an array?

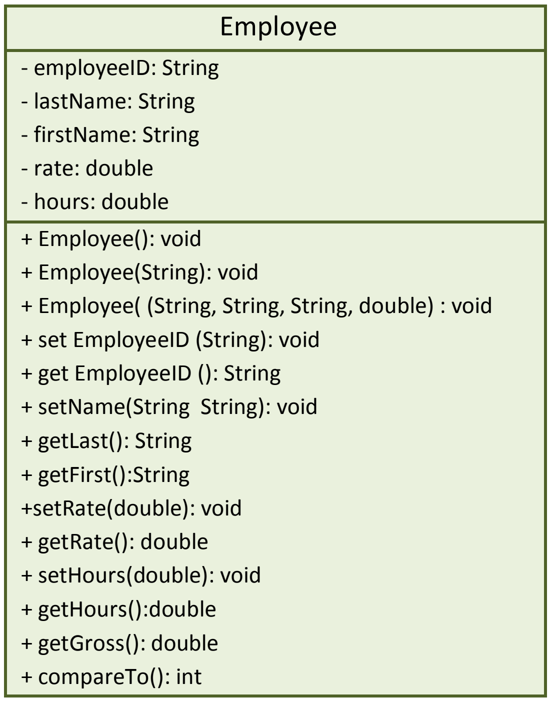
It turns out that Java has some features to help us work with arrays of objects, including implementing the Comparable Interface to sort objects as we wish them to be sorted, and simplified file I/O for files of objects, including reading and writing an array as a single object. We will examine these features of Java in the remainder of this chapter.

Comparing Instances of a Class of Objects

We have discussed the Comparable interface in previous chapters. We are re-examining it here because of its importance in sorting.

In order to sort objects, we need to be able to compare them in some way. The person who creates an object can implement the comparable interface to define how instances of the object can be compared. For example, consider the Employee class shown on the right. We could sort instances of the class several different ways:

- by any of its properties
  - *employeeID*
  - *lastName*
  - *firstName*
  - *rate*
  - *hours*
- by a combination of properties, such as *lastName* and *firstName* concatenated
- by a derived property, such as gross pay, which would be *hours * rate*.



Employee

- employeeID: String
- lastName: String
- firstName: String
- rate: double
- hours: double

+ Employee(): void
+ Employee(String): void
+ Employee((String, String, String, double) : void
+ set EmployeeID (String): void
+ get EmployeeID (): String
+ setName(String String): void
+ getLast(): String
+ getFirst():String
+setRate(double): void
+ getRate(): double
+ setHours(double): void
+ getHours():double
+ getGross(): double
+ compareTo(): int

The comparable interface will only be designed for one of these possibilities, most likely the object's key property, if it has one. Recall that a **key property** is a property that has a unique value for each instance of the object – no two instances can have the same value for key property. A *key property* is known as a *key field* in a database management system.

Instances of an object that have a *compareTo()* method implementing the Comparable interface can be placed in order by the property or properties used by the compareTo() method. According to the official definition of the Comparable interface[1], this ordering is referred to as the class's **natural ordering**, and the class's *compareTo()* method is referred to as its **natural comparison method**. We should be careful to choose the most logical or most natural way to compare instances of an object when we implement the Comparable interface for a class of objects. We can always create other methods that sort the instance by other properties.

In the example above, the Employee class has a *compareTo()* method that compares instances according to *employeeID,* so it can be used to sort according to *employeID.* Other methods can be created to sort according to other properties, such as a *compareByName()* method that could sort by *lastName* concatenated with *firstName.* The code for such a method is shown here:

```
/* this method will compare two instances of employee by name: lastName first, then by
firstName if the last names are equal. It makes use of the String class compareTo()
method.
It returns: 0 if the names are equal, a negative value if this name comes first, a
positive value if other's name comes first. */

int compareByName(Employee other){

   // create full names by concatenating last and first
   String thisFullName = this.getLast() + this.getFirst();
   String otherFullName = other.getLast() + other.getFirst();

   // compare and return values using the String class compareTo() method
   return thisFullName.compareTo(otherFullName);
}
```

This method could be included in the employee class, or you could extend the employee class and add this as a method in the new subclass.

So, to summarize, *compareTo()* should be used for a class's natural order, if one is apparent, and additional compare methods can be created for sorting by other criteria if needed.

19.6 File I/O for Objects

In the last chapter we saw how to implement text file I/O and binary file I/O. here we will examine object file I/O in Java. We could implement object file output by saving each property of an object one

[1] see the Oracle documentation for the Comparable interface, online at: http://docs.oracle.com/javase/7/docs/api/java/lang/Comparable.html

at a time a text file, with one object per line, or we could use Java's built in features for object file I/O. Examples of both are shown and discussed below.

File I/O for Objects – CSV Files

The methods shown here can be used to save all the properties of the Employee class from above, in a CSV file – a text file with one instance of Employee on each line and the data delimited by comas.

```
/* This method saves all of the properties of an instance of the employee class as a
single line comma delimited String.  It is different from the toString() method.

void toCSVstring(Employee e){

   String record = e.employeeID + ","
                 + e.lastName  + ","
                 + firstName + ","
                 + rate + ","
                 + hours + "\n";

   return record;
}
```

The *toCSVstring()* method can then be used to save an Employee array to a CSV file, as in the following:

```
    /* This method writes an array of Employee objects to a CSV text file,
     * with data for one instance of the object on each line.
     * It uses the toCSVstring() method to create each line for the file.
     *
     * The count parameter identifies how many records are written,
     * starting from 0 in the array.
     */

    public static void writeCSV(Employee[] empArray, int count) throws Exception {

        // create a File class object and give the file the name employees.csv
        java.io.File empCSV = new java.io.File("employees.csv");

        // Create a PrintWriter text output stream and link it to the File empCSV
        java.io.PrintWriter outfile = new java.io.PrintWriter(empCSV);

        // iterate the elements actually used
        for (int i = 0; i < count; i++) {
            outfile.write(toCSVstring(empArray[i], lines[i]);
        } // end for

        outfile.close();

    } // end writeCSV()
     //***********************************************
```

Many applications, such as Microsoft Excel, can read CSV files. The methods above – *toCSVstring()* and *writeCSV()* – can be used to output an array of objects in a CSV file that can be directly read into Excel. The code in these two methods could also be combined into a single method. Such methods can be quite useful for moving data between Java software and applications that can read CSV files.

A corresponding method can be created to read data from a CSV file into an array of objects.

File I/O for Objects – Object Serialization and Binary Data

If we do not need to work with text files such as CSV files, then writing objects directly to binary I/O files is fast, convenient, and results in smaller files. This is especially useful for large amounts of data.

Java provides a mechanism for writing objects to binary files know as serialization. **Object Serialization** is the process of turning an instance of an object into a binary data stream. **Object Deserialization** is the process of restoring an instance of an object from a binary data stream. Java provides the *ObjectOutputStream* class that has a *writeObject()* method and the *ObjectInputStream* class that has a *readObject()* method to handle object serialization and deserialization.

There is one catch, however – in order for an instance of a class to be serialized, it must implement the *Serializable* interface, which is part of the Java.io package. This is really an easy thing to do because the *Serializable* interface is a marker interface. A **marker interface** is an interface used to marks a class for special treatment by a Java compiler and a JVM. Marker interfaces have no properties and no methods. They are used to activate features in Java that would normally be turned off. *Serializabe*, *Cloneable*, and *SingleThreadModel* are examples of marker interfaces.

annotations box

In order for an object to be written to or read from a data file using the *writeObject() and readObject()* methods, it needs to be in a serializable class. A class is serializable if it implements the serializable class, and any data members of the class that are objects must also be serializable. You may recall from discussions of inheritance that any class which extends a class implementing an interface also implements that interface, so a **serializabe class** is a class that either implements the serializable interface or is a subclass of a class that implements the serializable interface, and all of its data members that are objects must also be serializable.

When an instance of an object is serialized – in other words, when we write it to an *ObjectOutputStream* using the *writeObject()* method – the state of an instance the object is turned into binary data stream and written to the target destination. Recall that the state of an instance of an object is the collection of all of the values of the properties of an object. Of course, this does not include any static properties, which are class level and do not pertain to an instance of an object. So, when we write an object to a data file using the *writeObject()* method, all of its properties are written to the file as binary data, along with some identifying information.

Data written to a file using the *writeObject()* method can be retrieved using the readObject() method. The following code demonstrates this with the Employee class used earlier in the chapter. It is included with the files for this chapter as the zipped NetBeans project ***ObjectSerializationDemo.*** The project includes the code for the Employee class need to run the software. The software is discussed after the listing of the code.

NOTE: Since Java 1.5, Java annotations can be used to indicate a serializable class. However, using annotations for runtime options results in significantly slower code than by implementing a marker interface. The use of the Serializable marker interface is recommended. For more about Java annotations see: http://docs.oracle.com/javase/tutorial/java/annotations

```
/*
 * ObjectSerializationDemo.java
 * CSCI 112 - Spring 2014
 * This code demonstrates the use of the ObjectOutputStream class and its
 * writeObject() method, along with the ObjectInputStream class and its
 * readObject() method for binary object I/O in Java.
 *
 * It creates a hardcoded instance of the Employee class and
 * writes it to a data file. It then retrieves it from the file into a new object
 * and displays the new object's properties on the console.
 *
 * The Employee class is included in this code.
 */

package objectserializationdemo;
import java.io.*;

public class ObjectSerializationDemo {

    public static void main(String[] args) throws Exception {

        // instantiate an Employees for this demo
        Employee empA = new Employee("1111", "Smith", "John");

        // instantiate an employee object to receive the data from a data file
        Employee empB = new Employee();

        // set the hours
        empA.setHours(40.0);

        // set the pay rates
        empA.setRate(12.50);

        // alert the user and write the object to a data file
        System.out.println( "Employee A is: " + empA.toString() );
        System.out.println("writing employee A data to a file.\n");
        writeToFile(empA);

        // read from data file into a new object and tell the user
        empB = readFromFile();
        System.out.println("Reading employee B data from a file:");
        System.out.println( "Employee B is: " + empB.toString() );
        System.out.println();

    } // end main()
      // ************************************************************************


    /* This method writes an instance of the Employee class to a binary file.
     * It demonstrates how to serialize an object using ObjectOutputStream
     */
    public static void writeToFile(Employee emp) throws Exception {

        // declare file and stream variables before thr try block
        FileOutputStream empFile= null;
        ObjectOutputStream outfile = null;

        try {
            // set up the File and stream objects
            empFile = new FileOutputStream("employees.ser");
            outfile = new ObjectOutputStream(empFile);

        // write an Employee object to a file
            outfile.writeObject(emp);

        // use a standard catch block
        } catch (IOException e) {
            e.printStackTrace();

        // no matter what happens, close the file
        } finally {
            outfile.close();
        }
    }  // end writeToFile(Employee e)
    //*****************************************************************************

    /* This method reads an instance of the Employee class from a binary file.
     * It demonstrates how to deserialize an object using ObjectInputStream
     */
    public static Employee readFromFile() throws Exception {

        Employee emp = null;   // an Employee object to hold data from the file

        // declare file and stream variables before thr try block
        FileInputStream empFile= null;
        ObjectInputStream infile = null;

        try {
            // set up the File and stream objects
            empFile = new FileInputStream("employees.ser");
            infile = new ObjectInputStream(empFile);

            // read the object from a file
            emp = (Employee) infile.readObject();

        // use a standard catch block
        } catch (IOException exc) {
            exc.printStackTrace();

        // no matter what happens, close the file
        } finally {
            infile.close();
        }  // end finally

        // return the object
        return emp;

    } // readFromFile()

} // end class ObjectSerializationDemo
//*****************************************************************************
```

The main method in the software shown above creates an instance of the Employee class named *empA* and initializes the properties of the object. It also creates an empty object of the Employee class named empB. The method then calls *writeToFile* to write empA to a data file. It ends by calling the method *readFromFile* to read the data from the file into empB.

The method *writeToFile* creates a File class object named *empFile* associated with the file *employees.ser.* It is common practice among Java professionals to end binary object files with the extension *ser. (Look up "file extension ser" using Google.)* It then associates the File with an ObjectOutputStream named *outfile.* The program uses the *outfile.writeObject(emp).*

The method *readFromFile* creates a File class object named *empFile* associated with the file *employees.ser* and associates the File with an ObjectInputStream named *infile.* The instruction

```
emp = (Employee) infile.readObject();
```

reads the serialized data from the file into the method's variable *emp.* Notice the typecasting to an *Employee* object. This is necessary because *readObject()* will read the data as an object of the superclass *Object*, so we always need to cast the incoming data to the appropriate class. Be careful – for this to work, the class must exist and must be accessible to the instruction, either as an inner class of the class containing the instruction, as another class in the same package, or by some other mechanism, such as using a proper import statement. Otherwise, a *ClassNotFoundException* will be generated.

For the *writeToFile* and *readFromFile* methods to work, the Employee class must be serializable. Here is part of the code for the Employee class, showing the header for the class and the data members of the class:

```
class Employee implements Serializable {

    private String employeeID;  // key field -- uniquely identifies an employee
    private String lastName;    // employee's last name
    private String firstName;   // employee's last name
    private double rate;        // hourly pay rate
    private double hours;       // hours worked in the pay period

// the remainder of the class is not shown . . .
```

The employee class implements serializable and all of its data members are serializable. This means we can use the *writeObject()* method from the *ObjectOutputStream* class and *readObject()* method from the *ObjectInputStream* class to read and write objects, as the methods *writeToFile* and *readFromFile* do above. (Primitive variables and the String class are serializable.)

So, to summarize, an instance of an object can be written to a data file or read from a data file with a single instruction using the *java.io.ObjectOutputStream* class that has a *writeObject()* method and the *java.io.ObjectInputStream* class that has a *readObject()* method. In order for this to work, the object must be serializable and the instruction with *readObject()* must be able to cast the incoming data into an accessible class of object.

File I/O for Objects – Array Serialization and Data Persistence

An array is an object. In fact, an array is an object that implements the Serializable interface. This has enormous implications in terms of simplifying Java programming. It means that if the underlying data type for the array, such as the *Employee* class above, is serializable, then an entire array can be written to a data file to a data file with a single instruction with the *java.io.ObjectOutputStream* class *writeObject()* method and or read from a data file using the the *java.io.ObjectInputStream* class *readObject()* method.

Primitive data types and the String class are serializable. File I/O for arrays of primitive data types, the String class, or any class that is serializable can handled using object streams. This makes the code simpler, quicker and more reliable than code that involves File I/O in which elements of an array and the properties of objects in arrays of object are individually written to and read from a file.

There are still many cases in which it is desirable to write object properties and arrays as text data files, but for code that simply needs to save and retrieve large amounts of data object serialization is good solution. Professional Java programmers often use object serialization for data communication and for data persistence. **Data Persistence** is the long term storage of data so that it exists even when the code that created it is no longer running, such as storing data in files on external storage devices.

People using traditional HTML browsers on personal computers and handheld devices are often connected with large corporate databases by professionally written Java software serving as transactional middleware in a three-tiered system. The user interface on the browser is the front end, the large database is the back end, and the Java software is the tier in the middle tying everything together. The middleware needs to communicate data and often needs to make persistent copies of important data. This is commonly done professionally using serialization.

Serialization and proper data persistence are such important topics among Java professionals that a number of websites exist discussing questions about serialization often asked by major employers who hire Java programmers – banks, insurance companies, retailers like Amazon, and so on. An example of this can be found on the *JavaRevisited* Website at:

http://javarevisited.blogspot.com/2011/04/top-10-java-serialization-interview.html

Serialization is also used by middleware for data communications. Serialized data can be transmitted from one application to another *using ObjectOutputStreams and ObjectInputStream*s. Data communication software is beyond the scope of this chapter, but it uses serialization of objects in the same way that data storage and retrieval software does.

The following example demonstrates simplified array serialization for data persistence. It creates a short array of Employee objects and writes it to a file using the *ObjectOutputStream* class *writeObject()* method, then reads it back into another array using the *readObject()* method from the class.

```
/*
 * ArraySerializationDemo.java
 * CSCI 112 - Spring 2014
 * last edited March 8, 2014 by C. Herbert
 *
 * This code demonstrates the use of the ObjectOutputStream class and its
 * writeObject() method, along with the ObjectInputStream class and its
 * readObject() method for simplified array I/O in Java. It is an example of the
 * way in which array serialzation can be used for data persistence.
 *
 * It creates a hardcoded instance of an Employee array and writes it to a data
 * file. It then retrieves it from the file into a new array
 * and displays the new array on the console.
 *
 * The Employee class is included in this code.
 *
 */
package arrayserializationdemo;
import java.io.*;

public class ArraySerializationDemo {

    public static void main(String[] args) throws Exception {

        // create an Employee array for this demo
        Employee[] empA = new Employee[5];

        // instantiate five Employees in the array
        empA[0] = new Employee("1111", "Smith", "John");
        empA[1] = new Employee("2222", "Jones", "Mary");
        empA[2] = new Employee("3333", "Ali", "Ben");
        empA[3] = new Employee("1111", "Rodgers", "Fred");
        empA[4] = new Employee("1111", "Rambo", "John");

        // set the hours
        empA[0].setHours(40.0);
        empA[1].setHours(45.0);
        empA[2].setHours(37.5);
        empA[3].setHours(40.0);
        empA[4].setHours(40.0);

        // set the pay rates
        empA[0].setRate(12.5);
        empA[1].setRate(12.5);
        empA[2].setRate(11.0);
        empA[3].setRate(15.0);
        empA[4].setRate(15.0);

    // instantiate an employee array to receive the data from a data file
        Employee[] empB = new Employee[5];

        // show the user the original array using an enhanced for loop (Chapter 18)
        System.out.println( "The original arrays is: ");
        for ( Employee emp : empA)
            System.out.println( "\t" + emp.toString() );

        // alert the user and write the array to a file
        System.out.println("\nWriting employee array a data to a file.");
        writeToFile(empA);

        // alert the user and read data from a file into the duplicate array
        System.out.println("\nReading employee B data from a file.");
        empB = readFromFile();

        // show the user the duplicate array
        System.out.println( "\nThe duplicate array is: ");
        for ( Employee emp : empB)
            System.out.println( "\t" + emp.toString() );

    } // end main()
    //*****************************************************************************


    /* This method writes an instance of the Employee class to a binary file.
     * It demonstrates how to serialize an object using ObjectOutputStream
     */
    public static void writeToFile(Employee[] emp) throws Exception {

        // declare file and stream variables before thr try block
        FileOutputStream empFile = null;
        ObjectOutputStream outfile = null;

        try {
            // set up the File and stream objects
            empFile = new FileOutputStream("employees.ser");
            outfile = new ObjectOutputStream(empFile);

        // write an Employee object to a file
        outfile.writeObject(emp);

        // use a standard catch block
        } catch (IOException e) {
            e.printStackTrace();

        // no matter what happens, close the file
        } finally {
            outfile.close();
        }
    }  // end writeToFile(Employee e)
    //*****************************************************************************


    /* This method reads an instance of the Employee class from a binary file.
     * It demonstrates how to deserialize an object using ObjectInputStream
     */
    public static Employee[] readFromFile() throws Exception {

        Employee[] emp = null;   // an Employee object to hold data from the file

        // declare file and stream variables before thr try block
        FileInputStream empFile= null;
        ObjectInputStream infile = null;

        try {
            // set up the File and stream objects
            empFile = new FileInputStream("employees.ser");
            infile = new ObjectInputStream(empFile);

            // read the object from a file
            emp = (Employee[]) infile.readObject();

        // use a standard catch block
        } catch (IOException exc) {
            exc.printStackTrace();

        // no matter what happens, close the file
        } finally {
            infile.close();
        }  // end finally

        // return the object
        return emp;

    } // readFromFile()

} // end class ArraySerializationDemo
//*****************************************************************************
```

You should read through the code above and make sure you know how it works. It uses the same employee class as the example above.

You can see that the code to write an array using *writeObject()* and the code to read and array using *readObject()* isn't very different from the code in the previous example that writes and reads a single object. That's because an array is a single object.

Transient Data

Object serialization sends the data for the current state of an instance of an object to an object output stream. It does not include static properties of an object because these are class-level properties that are not associated with an instance of an object.

But what if there is data in properties of an instance that we do not wish to send to a data file? For example, what if, for convenience, a student record has a property for GPA, but it can be calculated, so its current value should not be saved as part of the persistent data when the object is saved? What if we wish to make a copy of an object to send to another program, but we do not want to include certain sensitive information, such as account numbers or passwords?

Transient data is data that should not be included with the persistent data for an object. Java provides the *transient* access modifier to indicate that a property should not be included in the serialization of an object. (See the *Java Language Specification (JLS)*, chapter 8, section 3.) Here is an example:

```
class Student implements Serializable {

    private String studentID;        // key field -- uniquely identifies each student
    private lastName;                // employee's last name
    private String firstName;        // employee's last name
    private transient String gpa;    // hours worked in the pay period
    private String major;            // hourly pay rate

// the remainder of the class is not shown . . .
```

The *writeObject()* method will not include transient data in an ObjectOutputStream. The *readObject()* method, as it deserializes incoming data from an ObjectInputStream, will initialize transient properties using the default value as defined in the class declaration – most commonly a null value.

In the above example, a student's gpa would not be included in an ObjectOutputStream during serialization. During deserialization, the *gpa* property of the new instance of Student would be initialized to a default value – a null value in this case.

So, to summarize, if the underlying data type for an array is serializable, then the array is serializable. This means we can use object stream I/O for simplified code to read and write arrays in Java software. Object and array serialization is used by professional Java programmers for data persistence and data communications software. Data that should not be included with the persistent data can be indicated by the *transient* access modifier in the class definition.

Chapter Review

Section 1 of this chapter discussed transitive data sets and the bubble sort. Key terms included *transitive property of equality, transitive property of inequality, comparable set, comparable data types,* and *bubble sort*.

Section 2 introduced the selection sort.

Section 3 briefly discussed the concept of algorithmic complexity and how the temporal and spatial complexity of the selection sort compares to that of the bubble sort. Key terms included *temporal complexity and spatial complexity.*

Section 3 introduced the insertion sort and discussed how it compares to the bubble sort and selection sort. The key term *data entropy* was introduced.

Section 4 described how to sort objects. Key terms included *key property*, *natural ordering* and *natural comparison method.*

Section 5 discussed file I/O for objects, including writing objects to a CSV file, serialization of objects for writing to a binary file, array serialization, and data persistence. Key terms included *Object serialization*, *Object deserialization*, *marker interface*, *serializabe class* and *data persistence.*

Chapter Questions

1. What do the transitive property of equality and the transitive property of inequality have to do with sorting data? What is a comparable set and how is it related to the concept of a comparable data type?
2. How does a bubble sort work? What happens during each pass in a bubble sort? When does the algorithm stop making passes in a bubble sort?
3. How is the time it takes to complete a bubble sort related to the size of the data set being sorted? How is it related to the way in which the data set being sorted is organized?
4. How does a selection sort work? What happens during each pass in a selection sort? Where does the "work" of sorting occur in a selection sort?
5. How is the time it takes to complete a bubble sort related to the size of the data set being sorted? How is it related to the way in which the data set being sorted is organized?
6. What does the temporal complexity of an algorithm describe? What does the spatial complexity of an algorithm describe?
7. How do the temporal and spatial complexity of bubble sort, section sort, and insertion sort compare to one another? Which of these three can be implemented as an in-place sort?
8. How does an insertion sort work? What happens during each pass in an insertion sort? Where does the "work" of sorting occur in an insertion sort?

9. What is meant by the term *data entropy*? How does it affect the time it takes to conduct a sort? Which of the three sorting methods in this chapter is least affected by data entropy, and why?
10. How can we compare instances of a class of objects? Why is this important for sorting object-oriented data?
11. What interface should be implemented if a class has a *compareTo()* method? How is the *natural ordering* of a class of data related to its *compareTo()* method? How is this related to the concept of a key property for a class of objects?
12. What does it mean to save a set of instances of an object as a CSV file? Why is it advantageous to save text data as a CSV file?
13. Why would we want to sue binary file I/O for saving and retrieving objects instead of working with text files, such as CSV files?
14. What mechanism does java provide for writing objects to binary files? What classes does Java use for binary I/O with objects? What methods in these classes are most often used to read and write objects as binary data?
15. What are serialization and deserialization in Java programming? What interface must be implemented in Java to do this?
16. What is a marker interface and what are marker interfaces used for? What are some examples of marker interfaces in java?
17. What characteristic must the data in an object have in order for the object to be serializable? Under what conditions is a class serializable? What data types are serializable?
18. Why is type casting needed when reading an object from a data file using the readObject() method in the ObjectInputStream class?
19. Under what conditions can an entire array be serialized and written to a file with a single write instruction? What tasks do professional Java programmers use object serialization for?
20. What is transient data? What mechanism does java provide to make a property of an object transient?

Chapter Exercises

1. In this exercise you will practice the insertion sort and the bubble sort using a standard deck of playing cards.

   First, deal a hand of 8 cards. Then sort the cards by following the steps in the selection sort algorithm described in this chapter. Follow the algorithm – be careful not to take any shortcuts.

   Next, deal a hand of 8 cards and sort the cards by following the steps in the insertion sort algorithm described in this chapter. Follow the algorithm – be careful not to take any shortcuts.

   if necessary, repeat this a few times until you are familiar with each algorithm.

   Did the exercise help you to understand how the two sorting methods work, and how they are different from one another? Which one was easier to follow? Did one method seem faster than another?

2. In this exercise you will play the role of a principal investigator doing research on the time it takes a person to sort a set of playing cards using selection sort and insertion sort. Your have three questions to research:

   - What differences are there in the time it takes to perform a selections sort vs. the time it takes to perform an insertion sort?
   - How much does the size of the set of cards affect the time it takes to sort the data? If the set doubles in size, does the time double? Does it increase more or less than the increase in the size of the set of cards?
   - How hard or easy is it for a person to carry out each algorithm? Does the person follow the algorithm closely, or does the person take shortcuts, or leave out steps?

   Obtain a standard deck of 52 playing cards. Find a willing subject to work with you, such as a spouse, a child, or a sibling. The willing subject will need to be patient, because you are going to ask them to sort 18 sets of playing cards.

   First, describe the selection sort process to the willing subject. Then, do the following, keeping track of how long each sort takes:

   repeat three times -- deal your willing subject a set of 5 playing cards and have the subject sort the cards using a selection sort.

   repeat three times -- deal your willing subject a hand of 10 playing cards and have the subject sort the cards using a selection sort.

   repeat three times -- deal your willing subject a set of 20 playing cards and have the subject sort the cards using a selection sort.

Next, describe the insertion sort process to your willing subject and do the following, again keeping track of how long each sort takes:

repeat three times -- deal your willing subject a set of 5 playing cards and have the subject sort the cards using an insertion sort.

repeat three times -- deal your willing subject a hand of 10 playing cards and have the subject sort the cards using an insertion sort. .

repeat three times -- deal your willing subject a set of 20 playing cards and have the subject sort the cards using an insertion sort.

Fill in the ***Sorting Experiment spreadsheet***, included with the files for this chapter. What conclusions can you draw from your research? What additional research could you perform to help better answer the research questions for this experiment?

3. The bubble sort described in this chapter is a standard bubble sort. There are a number of ways the bubble sort can be modified to make it work a little faster. Using the Internet as a research tool, see if you can find at least three ways to modify a bubble sort to make it work faster. Describe your findings, including a description of the modification, how it actually saves time when sorting, and how the entropy of the data affects how the modification saves time.

4. This is an exercise in correctly implementing insertion sort and selection sort.

   The NetBeans project ***tutorial*** was used earlier in this course and in CSCI 111. It reads a list of programming language tutorials in from a data file, sorts the list, then writes the list to another file. The method that does the sorting uses the bubble sort method to do its sorting.

   Create two additional methods for the project – one that performs a selection sort on a String array and one that performs an insertion sort on a String array.

   After the two additional sorting methods are in place, modify the main method in the project so that it repeats the process it carried out for the bubble sort (*read-sort-write*) for the selection sort and for the insertion sort. In each case the file should read the original file, sort the data, then write the data to a new file.

5. The file “fall2014.ser” included with the files for this chapter has the real Fall 2014 CIS and Computer Science course offerings (other than CIS 103) in a binary file. The file “Course.java” is the declaration for the class used to create this file.

   Your task is to complete the following

   1. Create a Java application to read the data from the “fall2014.bin” file into a *Course* array. There are 24 sections in the data file; each will be an element in the array. If you do this properly, then you should be able to use a single *readObject()* instruction, similar to what’s done in the *ArraySerializationDemo* application in this chapter.

   2. Sort the data according to *course* and *section* using either a selection sort or an insertion sort.

   3. Write the data to a CSV text file, with one course on each line in the file, delimited with commas.

   Your software should have separate methods to read the data into the array, sort the data, then write the data to a CSV file. These methods can be called from the main method.

   Don’t forget to include the Course class declaration from the *Course.java* file in your project. There are several ways to do this – as an inner class, as a separate class declaration in your project, or as a separately compiled java file’s *.class* in the same folder as your project’s *.class* file.

   If your application works, you should be able to open the CSV file in Microsoft Excel, then use the features of Excel to work with the data.

   Submit a zipped NetBeans project folder with a copy of the CSV file and your lab report in the folder.