JAVA NOTES

DATA STRUCTURES AND ALGORITHMS

Terry Marris July 2001

4 SELECTION SORT

4.1 LEARNING OUTCOMES

By the end of this lesson the student should be able to

- explain the concept of sorting
- implement the Comparable interface
- explain the selection sort algorithm
- use the selection sort algorithm to arrange the contents of an array in some natural order

4.2 PRE-REQUISITES

The student should be familiar with the *String* and *Integer* classes (Part A §10 and §11) and Interfaces (Part A §18).

4.3 INTRODUCTION

In the last lesson we looked at a searching algorithm, looking for an object in an array of objects. In this lesson we look at sorting an array of objects.

Sorting is placing a collection of objects into some kind of order such as ascending numerical order (1, 2, 3, ...) or alphabetical order (A, B, C, ...).

The relational operator, <, allows us to determine which of two *int* variables has the least value.

```
int a = 3;
int b = 2;
...
if (a < b)
  ...
```

To determine which of two *String* variables had the least value (in alphabetical or dictionary order) we use the *compareTo(Object)* method.

You may recall that the *String* class method *int compareTo(Object obj)* returns an *int* less than one if this string comes before the given object, zero if this string is the same as the given object and an *int* greater than zero if this string comes after the given object in alphabetical order. For example, each of the following expressions is true:

```
"bat".compareTo("cat") < 1
"cat".compareTo("bat") > 1
"cat".compareTo("cat) == 0
```

But what about two employee objects? You might decide that one employee comes before another because the surname of one comes before the other in alphabetical order. Or you might decide that employees are ordered on their length of service or on their salaries. If we are going to sort employee objects into some kind of natural order we need to implement an interface named *Comparable*; we do this in the *Employee* class.

4.4 THE COMPARABLE INTERFACE

The *Comparable* interface has just one method.

```
public interface Comparable {
  public int compareTo(Object obj);
}
```

We are expected to implement *int compareTo(Object obj)* so that it returns an *int* less than zero if this object comes before the given object in some natural order, zero if this object equals the given object, and an *int* more than zero if this object comes after the given object. Class *Employee*, shown below, implements *Comparable* so that two employees may be ordered on their salaries.

```
class Employee implements Comparable {
  private String name;
  private int salary;

  public Employee(String name, int salary)
  {
    this.name = name;
    this.salary = salary;
  }


  public int compareTo(Object obj)
  {
    Employee emp = (Employee)obj;
    if (this.salary < emp.salary)
      return -1;
    if (this.salary > emp.salary)
      return 1;
    return 0;
  }
```

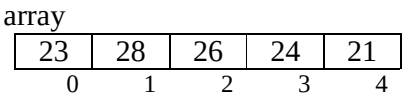
General-purpose sorting algorithms require the objects they are ordering to have implemented the *Comparable* interface.

4.5 SELECTION SORT

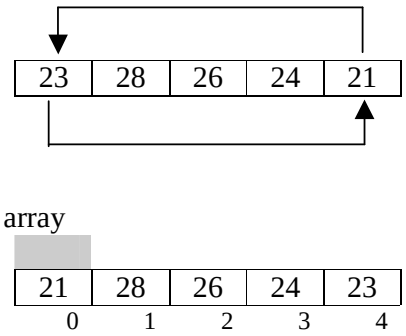
There are many sorting algorithms. Perhaps the simplest is the selection sort.

The essence of the selection sort is that we look for the next smallest (or least) item and place it in its correct position.

We start with the array shown below.

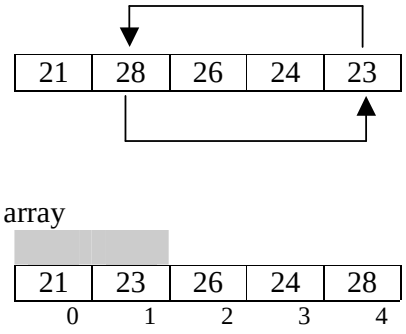


Looking along the array we see that 21 is the smallest. We swap positions with 23.



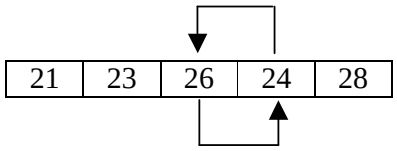
21 is now in its right place. The array is now *partitioned* - split into two parts. The sorted part is shown shaded. The unsorted part has no shading.

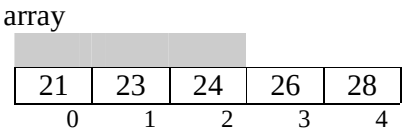
We look through the unsorted part of the array for the next smallest item. It is 23. We swop positions with 28.



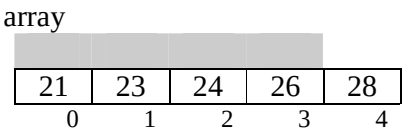
23 is now in its correct position. The sorted portion of the array has grown by one element; the unsorted portion has shrunk.

We look through the unsorted part of the array for the next smallest item. It is 24. We swop positions with 26.

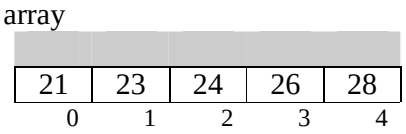




24 is now in its correct position. We look through the unsorted part of the array for the next smallest number. It is 26. We see that 26 is already in its correct position. So no swaps are required here.



We have reached the last element in the array and so sorting is finished.



We use variables *min* and *pMin* to store the least item and its position in the array. Index *i* refers to the beginning of the unsorted part of the array.

```
for (int i = 0; i + 1 < array.length; i++) {
  min = array[i];
  pMin = i;
```

Now we scan the rest of the array, beyond location *i*, for the smallest item. We store the smallest item, and its location.

```
 for (int j = i + 1; j < array.length; j++) {
   if (((Comparable)array[j]).
        compareTo((Comparable)min) < 0) {
     min = array[j];
     pMin = j;
   }
 }
```

The *(Comparable)* cast operator is necessary because the *Object* class itself does not implement *Comparable* and its an array of non-specific objects that we are sorting.

Having found the least item and its location, we swap positions with the item in location *i*.

```
array[pMin] = array[i];
array[i] = min;
```

And then move *i* on by one.

The entire method is shown below.

```
/* SelectionSort.java
   Terry Marris  21 July 2001
*/

public class SelectionSort {
  public static void sort(Object[] array)
  {
    int pMin; // position of minimum in unsorted part of array
    Object min;  // current minimum in unsorted part of array

    for (int i = 0; i + 1 < array.length; i++) {
      min = array[i];
      pMin = i;

      for (int j = i + 1; j < array.length; j++) {
        if (((Comparable)array[j]).
                        compareTo((Comparable)min) < 0) {
          min = array[j];
          pMin = j;
        }
      }

      array[pMin] = array[i];
      array[i] = min;
    }
  }
}
```

4.6 USING THE SELECTION SORT

In the first test we populate an array by filling it with a collection of random *Integer* objects.

```
/* TestSelectionSort.java   Terry Marris  21 July 2001
*/

import java.util.*;

public class TestSelectionSort {
  public static Random generator = new Random();

  public static void populate(Object array[])
  {
    for (int i = 0; i < array.length; i++) {
      array[i] = new Integer(generator.nextInt(100));
    }
  }

  public static void print(Object array[])
  {
    String string = "[ ";
    for (int i = 0; i < array.length; i++) {
      string += array[i];
      if (i + 1 < array.length)
        string += ", ";
    }
    string += " ]";
    System.out.println(string);
  }

  public static void main(String[] s)
  {
    final int capacity = 5;  // indexed 0..4
    Integer[] array = new Integer[capacity];
    populate(array);
    System.out.println("Before sorting");
    print(array);
    SelectionSort.sort(array);
    System.out.println("After sorting");
    print(array);
  }
}
```

Output

```
Before sorting
[ 38, 53, 4, 22, 5 ]
After sorting
[ 4, 5, 22, 38, 53 ]
```

In the second test we sort a collection of employee objects in increasing order of salary.

First, we create the *Employee* class. You may remember in §4.4 above we stated that the objects in the collection to be sorted must implement the *Comparable* interface.

```
/* TestSelectionSort2.java on Employee objects
   Terry Marris  21 July 2001
*/

class Employee implements Comparable {
  private String name;
  private int salary;

  public Employee(String name, int salary)
  {
    this.name = name;
    this.salary = salary;
  }


  public int compareTo(Object obj)
  {
    Employee emp = (Employee)obj;
    if (this.salary < emp.salary)
      return -1;
    if (this.salary > emp.salary)
      return 1;
    return 0;
  }


  public String toString()
  {
    return name + " " + salary;
  }
}
```

```
public class TestSelectionSort2 {
  public static void print(Object array[])
  {
    String string = "[ ";
    for (int i = 0; i < array.length; i++) {
      string += array[i];
      if (i + 1 < array.length)
        string += ", ";
    }
    string += " ]";
    System.out.println(string);
  }


  public static void main(String[] s)
  {
    Employee[] empArray = new Employee[5];  // indexed 0..4
    empArray[0] = new Employee("tom", 56);
    empArray[1] = new Employee("dee", 75);
    empArray[2] = new Employee("hari", 36);
    empArray[3] = new Employee("ann", 43);
    empArray[4] = new Employee("may", 28);

    System.out.println("Before sorting");
    print(empArray);

    SelectionSort.sort(empArray);

    System.out.println("After sorting");
    print(empArray);
  }
}


Before sorting
[ tom 56, dee 75, hari 36, ann 43, may 28 ]
After sorting
[ may 28, hari 36, ann 43, tom 56, dee 75 ]
```

4.7 REVIEW

4.8 FURTHER READING

HORSTMANN & CORNELL *Core Java 2 Volume 1* pp 226
DROMEY RG *How to Solve it by Computer* pp 192
LEWIS J & LOFTUS W *Java Software Solutions* pp 287

In the next lesson we look at a searching method that requires its elements to be sorted.

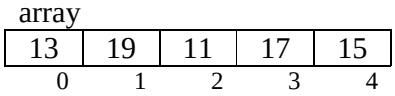
4.9 EXERCISES

1 Explain what is meant by the term *sorting.* Give two different examples.

2 Explain what the *Comparable* interface method *int compareTo(Object obj)* should do.

3 Re-write the *Employee* class so that it implements *Comparable* in terms of employee names.

4 Dry run the selection sort algorithm on the array shown below, drawing out the contents of the array at each stage.



5 Devise and test an algorithm *boolean isSorted(Object[] obj)* that returns *true* if all the elements in the given array of objects are arranged in their natural order, and *false* otherwise.