

07.09 Virtual Lecture Notes

To perform an insertion sort, it helps to create an empty copy of the array you are sorting. Then, one by one, you take values from the array you want to sort and insert them into the empty array. The insertion sort gets its name from the fact that, as you are inserting elements into the empty array, you look to see where the element should go and move array elements to the right as necessary.

Let us take a look at an algorithm for the insertion sort resulting in an array of integers being sorted in ascending order:

```
int[] source = { 12, 14, 15, 11, 13 };
int[] dest = new int[ source.length ];

for( int i = 0; i < source.length; i++ )
{
    int next = source[ i ];
    int insertIndex = 0;
    int k = i;
    while( k > 0 && insertIndex == 0 )
    {
        if( next > dest[ k - 1 ] )
        {
            insertIndex = k;
        }
        else
        {
            dest[ k ] = dest[ k - 1 ];
        }
        k--;
    }

    dest[insertIndex] = next;
}
```

The `dest` array starts as an empty array. The `for` loop steps through the source array and inserts each item, one at a time, into the destination array. It accomplishes this by taking the `next` item to be inserted and finding where to insert it by going through the destination array (`dest`) to find the proper spot. Notice that `k` is set to `i` so that we do not look at entries in `dest` that have no values. Working backwards in `dest`, we keep comparing `next` to a location in `dest`. If `next` is greater than `dest[k - 1]` then we have found the spot to add it. Otherwise, we copy the element at location `k - 1` to `k` (move it right), so that we are prepared for inserting the `next` into its spot. If we did not move `k - 1`, then we would have to move it after we found the location for `next`. By moving it as we find the location, we save time in the long run.

Now, let us put the insertion sort into a method. We have choices: either we pass two arrays to the method (the one to be sorted and an empty one), or we pass one array and have the method return a sorted array.

If we choose the first method, we would have this:

```
public static void insertionSort1(HouseListing[] source,
HouseListing[] dest)
{
    for( int i = 0; i < source.length; i++ )
    {
        HouseListing next = source[ i ];
        int insertIndex = 0;
        int k = i;
        while( k > 0 && insertIndex == 0 )
        {
            if( next.getCost() > dest[k-1].getCost() )
            {
                insertIndex = k;
            }
            else
            {
                dest[ k ] = dest[ k - 1 ];
            }
            k--;
        }

        dest[ insertIndex] = next;
    } // end of for
}
```

```
}
```

Notice that this will sort our list of houses based on cost and in ascending order. Again, to get descending order, just modify the comparison of the if statement to use a <.

If we choose the second way, our method will look like this:

```
public static HouseListing[] insertionCost2(HouseListing[] source)
{
    HouseListing[] dest = new HouseListing[ source.length ];

    for( int i = 0; i < source.length; i++ )
    {
        HouseListing next = source[ i ];
        int insertIndex = 0;
        int k = i;
        while( k > 0 && insertIndex == 0 )
        {
            if( next.getCost() > dest[k-1].getCost() )
            {
                insertIndex = k;
            }
            else
            {
                dest[ k ] = dest[ k - 1 ];
            }
            k--;
        }

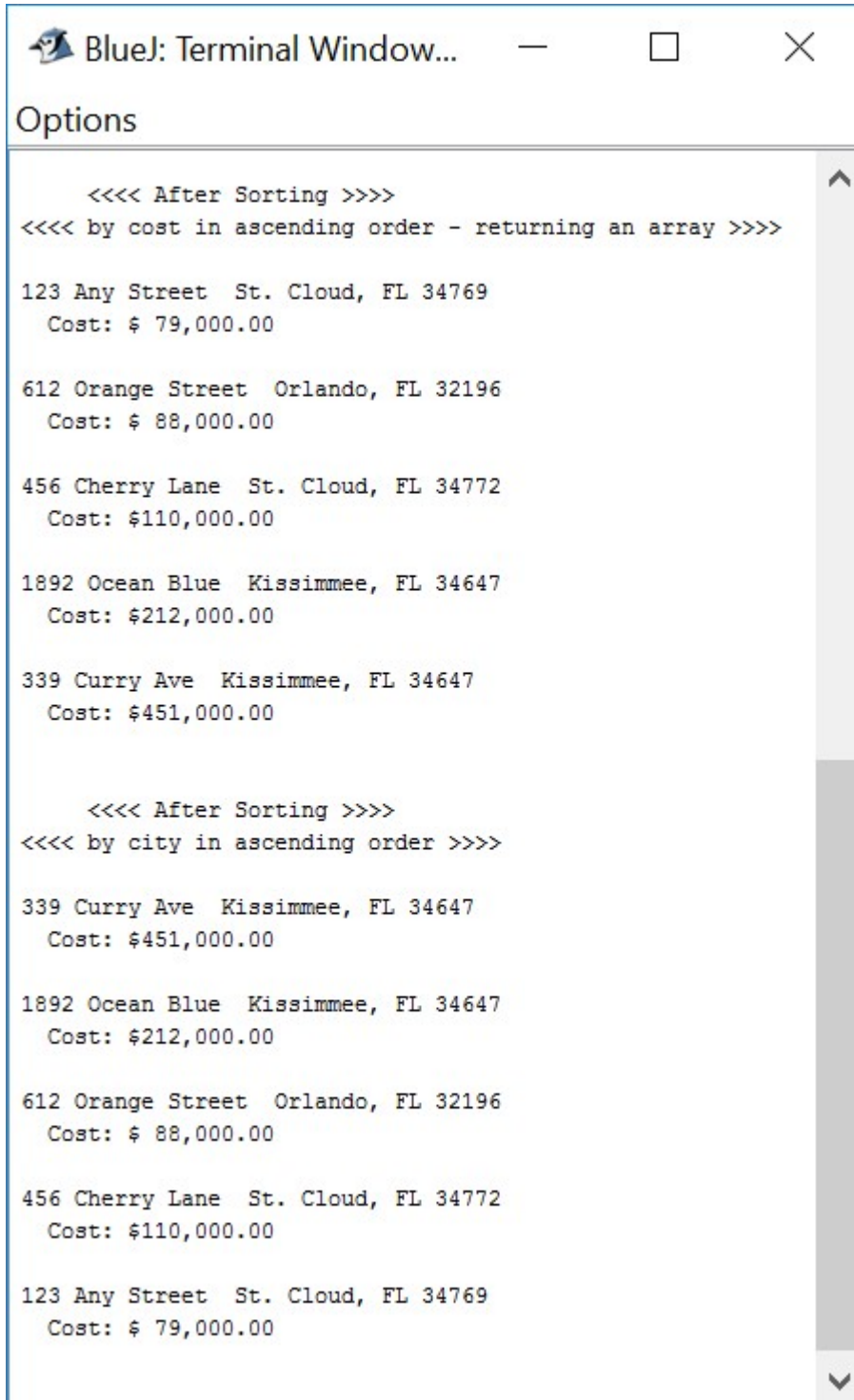
        dest[ insertIndex ] = next;
    } // end of for
    return dest;
}
```

Notice that this time we pass `dest` back via the return statement.

Both methods perform the sort we desire, but the first would require you to have two arrays in the `main` method. Therefore, many people would choose the second way. However, what if you wanted to keep

the original array in its original order for later use? The first method would allow us to do that. So depending on what you want to accomplish with the original array, you could choose either method.

Run the `TestListing2` program and observe the output. Notice, both versions of the insertion sort method are used for sorting the cost. Which you use depends on the task at hand. Also, we've been sorting a lot of numerical values. What about Strings? A method to sort on the city name is included. Do you see the `compareTo` method in action? Each sort method provides debugging print statements. Uncomment these so you can see a bit more detail related to each pass through the sort.



```
BlueJ: Terminal Window...

Options

==== After Sorting =====
==== by cost in ascending order - returning an array =====

123 Any Street  St. Cloud, FL 34769
    Cost: $ 79,000.00

612 Orange Street  Orlando, FL 32196
    Cost: $ 88,000.00

456 Cherry Lane  St. Cloud, FL 34772
    Cost: $110,000.00

1892 Ocean Blue  Kissimmee, FL 34647
    Cost: $212,000.00

339 Curry Ave  Kissimmee, FL 34647
    Cost: $451,000.00

==== After Sorting =====
==== by city in ascending order =====

339 Curry Ave  Kissimmee, FL 34647
    Cost: $451,000.00

1892 Ocean Blue  Kissimmee, FL 34647
    Cost: $212,000.00

612 Orange Street  Orlando, FL 32196
    Cost: $ 88,000.00

456 Cherry Lane  St. Cloud, FL 34772
    Cost: $110,000.00

123 Any Street  St. Cloud, FL 34769
    Cost: $ 79,000.00
```





Print