Module 4

Sorting Algorithms

Objectives:

1. Understand how running time and space requirements affect the choice of sorting algorithm.
2. Understand what a stable sort is, and how it can be important when choosing a sorting algorithm.
3. Understand the basics of the following sorting algorithms, as well as the pros/cons and Big-Oh running time for each algorithm:
    - Selection Sort
    - Insertion Sort
    - Shellsort
    - Mergesort
    - Quicksort
    - Radix Sort

Textbook:

Chapter 15, sections 15.1 - 15.25.

Chapter 16, sections 16.1 - 16.23.

Labs:

Do the Insertion Sort and Quicksort labs to trace the source code for these algorithms as it is executing.

Assignment:

Complete the assignment for Module 4.

Quiz:

Complete the quiz for Module 4.

Note: In this module you will notice that in several cases I provide code that deviates from the code presented in the book. This reinforces the notion that there can be multiple solutions for a given problem. There are many ways in which the problem of sorting a collection of elements can be solved. Each way represents a different algorithm. The differences between two algorithms can be relatively large, as in bubble sort vs. merge sort, or slight, as in the different variants of quicksort that exist. In all cases, keep in mind that the algorithm is the solution, while the source code is just a means for implementing the solution in some programming language. I hope to encourage you to understand the algorithm—how and why it works—rather than get hung up on memorizing source code. I will not expect you to regurgitate the source code for any of algorithms on an exam, although I do expect you to understand how the algorithms work. After this module you should be able to implement at least one of the basic search algorithms, such as insertion sort, given only an English description of the procedure the algorithm follows.

Some Factors to Consider when Choosing a Sorting Algorithm

Sorting is one of the most fundamental problems in computer science and a large number of sorting algorithms have been written over the years, some better than others. When choosing a sorting algorithm there are several important factors to consider.

Running Time

We are almost always interested primarily in minimizing the time required to sort a collection of N items. As with the searching algorithms we covered in Module 4, the number of element comparisons a sorting algorithm must make is a key factor in determining a sorting algorithm's efficiency. But unlike search algorithms, some sorting algorithms may involve multiple traversals of a collection of elements. So, when looking at the efficiency of a sorting algorithm, we are primarily interested in looking at the number of element comparisons and how many times the collection must be traversed in order to sort the collection.

Memory Requirements

Some collections can be sorted entirely in main memory, but sorting a large collection may require storing portions of the collection externally, to a hard drive or network resource. The space requirement is important, but remember that performing disk reads/writes are very expensive compared to executing instructions in memory, so the running time is also affected when external storage is used.

Stable vs. Unstable Sorts

A stable sort ensures that duplicate items are in the same positions relative to each other after sorting as they were before sorting. An unstable sort does not guarantee this. In many cases we don't care about this, but there are certain cases where this becomes important. Usually in such cases we are sorting items based not on the values of the items themselves, but rather on the values of keys associated with the items. For example, suppose you have a list of student names. Each name is associated with a key that represents the student's class standing (1 = Freshman, 2 = Sophomore, 3 = Junior, 4 = Senior). Now suppose the students are sorted according to their class standing. Figure 4.1 illustrates the difference between a stable sort and an unstable sort. Column 1 in the figure shows the initial state of the list. Column 2 shows the list following an unstable sort. Column 3 shows the list following a stable sort.

Unsorted	**Unstable Sort**	**Stable Sort**
Jones, 2	Smith, 1	Davis, 1
Davis, 1	Davis, 1	Smith, 1
Williams, 3	Thompson, 2	Jones, 2
Smith, 1	Jones, 2	Thompson, 2
Thompson, 2	Williams, 3	Williams, 3

Figure 4.1. Example of a stable sort vs. an unstable sort.

Selection Sort

The Algorithm

The algorithm for selection sort is very straightforward. Proceeding from left to right, find the smallest item and place it in the first position. Then starting at the second position, proceed from left to right to find the next smallest item, and place it in the second position, etc. The code for an iterative version of this algorithm is shown in the text in section 8.5.

Running Time

Although it may not be obvious, the algorithm uses two nested loops. The main loop iterates from position 0 to the last position. For every iteration of this loop the indexOfSmallest method is called. The `indexOfS-mallest` method also uses a loop, which iterates from position `first + 1` to the last position. Both loops are O(*N*). Since the loops are nested, the overall running time for the algorithm is O(N^2).

Stable or Unstable?

This is an unstable sort. An example demonstrating this is shown in the table below:

Initial List:	**7_1, 9, 7_2, 3_1, 3_2**
1st Iteration of Main Loop:	**3_1, 9, 7_2, 7_1, 3_2**
2nd Iteration of Main Loop:	**3_1, 3_2, 7_2, 7_1, 9**

In the table, subscripts are used to indicate the relative order of duplicate items. The list is sorted after the second iteration of the main loop. Although the algorithm will continue until the main loop reaches its final iteration, no more items will be repositioned. As you can see, before the sort the first occurrence of 7 comes before the second occurrence of 7, but after the sort the first occurrence of 7 comes after the second occurrence 7.

Insertion Sort

The Algorithm

The basic algorithm is this:

1. Iterate through the items from left to right.

2. Remove the current item and store it in temporary storage.

3. Loop from right to left, beginning at the position immediately before the current item, shifting items to the right as you go, until you find the proper position for the current item.

4. When you reach the position where the current item should go, insert it in that position.

In Figure 4.2, I show a pretty much standard, iterative version of insertion sort, along with the complexity analysis in terms of total statement executions and total element comparisons. In this module, I am deviating from showing exploded versions of the source code, and displaying the complexities for all statements in a compound statement on the same line. I'm also showing the complexities of the implicit loop jumps at the ends of the loops. The text shows some alternative implementations for insertion sort.

A couple of things should be noted about the statement complexities in Figure 4.2. Since the outer loop iterates from 1 to N, rather than from 0 to N like we have seen before, the loop increment statement will only execute N - 1 times instead of N times. Likewise, the loop termination condition is not executed N + 1 times, but only N times. This affects the complexities of the statements within the loop as well. This is why care must be taken to pay attention to the initial and final values of the loop variable when determining a precise Big-Oh value.

	Worst Case (items in reverse sorted order)	Best Case (items already sorted)
`public Comparable[] insertionSort(Comparable[] objArray)`		
`{`		
`    for (int i = 1; i < objArray.length; i++)`	1 + (N) + (N - 1)	1 + (N) + (N - 1)
`    {`		
`        Comparable firstUnsorted = objArray[i];`	N - 1	N - 1
`        int index = i - 1;`	N - 1	N - 1
`        while (index >= 0 && firstUnsorted.compareTo(objArray[index]) < 0)`	[N(N + 1)/2 - 1] + [N(N + 1)/2 - N]	(N - 1) + (N - 1)
`        {`		
`            objArray[index + 1] = objArray[index];`	N(N + 1)/2 - N	0
`            index--;`	N(N + 1)/2 - N	0
`        }`	(jump to beginning of while loop): N(N + 1)/2 - N	(jump to beginning of while loop): 0
`        objArray[index + 1] = firstUnsorted;`	N - 1	N - 1
`    }`	(jump to beginning of for loop): N - 1	(jump to beginning of for loop): N - 1
`    return objArray;`	1	1
`}`		
Complexity:	**(5N2 + 9N - 8)/2**	**8N - 5**
Complexity (element comparisons only):	**N(N + 1)/2 - N**	**N - 1**

Figure 4.2. An iterative implementation of insertion sort, showing the complexities of the worst and best case scenarios.

Running Time

Insertion sort uses 2 nested O(*N*) loops, as does selection sort, so its worst-case running time is O(N^2). Its advantage over selection sort is that if the list is already sorted, the innermost loop exits immediately because the second loop condition, which is responsible for performing the actual element comparisons, evaluates to false every time. This effectively renders the loop a O(1) statement. Therefore, for a pre-sorted list, insertion sort will run in O(*N*) time.

Stable or Unstable?

Insertion sort is a stable sort, as written here and in the text.

Shellsort

The Algorithm

Shellsort is an improvement upon the insertion sort algorithm. The biggest problem with insertion sort occurs when many items need to be shifted in order to place an item into its sorted position. The root cause of this problem is the fact that insertion sort only swaps items that are adjacent to each other. Donald Shell devised a way to allow non-adjacent items to be swapped, allowing items to “leap” from one position to another. The key is to sort subarrays (or sublists) in which the items in each subarray are spaced evenly apart (for example, every 5th item). The gap between the items in the subarray is gradually decreased, until eventually the gap becomes 1 (i.e., adjacent items are sorted). The gradual decrease in the size of the gap is defined by an increment sequence, which must have the following properties:

- The sequence must be a sequence of integers sorted in ascending order.
- The first integer in the sequence must be 1.
- All values in the sequence are unique (i.e., no duplicate values).

An example of a valid increment sequence would be: {1, 3, 5, 7}. First all items 7 items apart will be sorted, then all items that are 5 items apart, and so on. The text shows a good diagram of how this is done conceptually in the section on Shell Sort.

Section 8.24 shows the source code for shellsort. In Figure 4.3 I show you a more compact version of shellsort (this version sorts the entire array or list by default, whereas the text's algorithm allows portions of the array or list to be sorted).

```
public static void shellsort(Comparable[] objArray)
{
    Comparable temp;
    int j;

    for (int gap = objArray.length / 2; gap > 0; gap = gap == 2 ? 1 : (int)(gap / 2.2))
    {
        for (int i = gap; i < objArray.length; i++)
        {
            temp = objArray[i];

            for (int j = i; j >= gap && temp.compareTo(objArray[j - gap]) < 0; j -= gap)
            {
                objArray[j] = objArray[j - gap];
            }

            objArray[j] = temp;
        }
    }
}
```

Figure 4.3. A standard variant of the Shellsort algorithm.

Running Time

The choice of increment sequence is crucial to obtaining good performance. Starting with `gap` = *N/2* and halving the gap at each pass until it reaches 1 results in a worst-case running time of O(N^2), which occurs only if:

- *N* is an exact power of 2.
- All the large items are in the even-numbered positions.
- All the small items are in the odd-numbered positions.

On average, the running time is O($N^{3/2}$), making it better than either insertion sort or selection sort. Many variations of shellsort have been tried, but the 2 modifications that seem to be most significant are:

1. If `gap/2` is even, add 1 to make it odd; this alone brings the worst-case running time down to O($N^{3/2}$).

2. Divide `gap` by 2.2 instead of 2; to date no one has been able to prove why this helps.

Together, these modifications bring the average running time down to below O($N^{5/4}$). The shellsort algorithm I show above reflects both these modifications.

Stable or Unstable?

Shellsort is an unstable sort. Even though insertion sort is stable, shellsort doesn't guarantee the relative ordering of duplicate items will be preserved from increment to increment.

Mergesort

The Algorithm

Sorting large arrays can be time-consuming. Smaller arrays can be sorted much more quickly, and small arrays can be sorted using the same process used to sort a large array. This sounds like an ideal problem for recursion. Mergesort uses the divide and conquer approach to recursively divides an array into halves and sorts each half. Then all the small, sorted arrays are merged back into one large sorted array.

The text only gives you pseudocode for the mergesort algorithm, not the actual source code. Figure 4.4 shows the complete source code for mergesort. Note that there are 3 methods involved:

1. a public method used to call mergesort (this is the one that gets called externally)
2. a private method for recursively dividing the array
3. a private method for merging the divided arrays back together

The code is faithful to the pseudocode in the text, with the exception that I eliminated the increment statements for the variables `beginHalf1`, `beginHalf2`, and `index`. Instead I chose to increment these variables inside the array brackets. Because the increment operator appears after the variable name, the value of the variable will be evaluated before it is incremented. The author also uses more local variables in the merge method than are necessary, but I have left those intact.

Here are a few key points to remember:

- The recursive mergesort method does not actually do any sorting; it simply divides the array. All the sorting is performed by the merge method.

- The recursion always operates on equal-sized subarrays.

- The merged subarray is stored in a temporary array.

Running Time

The running time of mergesort is O(*N log N*). Recall that a divide and conquer algorithm that performs repeated halving will give you O(*log N*) performance. However, the merging process is linear, and is performed every time the recursive mergeSort method is called. Therefore the overall running time is O(*N log N*).

Stable or Unstable?

As coded in Figure 4.4, mergesort is a stable sort.

```java
/** Public method for calling mergesort */
public static void mergeSort(Comparable[] objArray)
{
    Comparable[] tempArray = new Comparable[objArray.length];
    mergeSort(objArray, tempArray, 0, objArray.length - 1);
}

/** Recursive method */
private void mergeSort(Comparable[] objArray, Comparable[] tempArray, int first, int last)
{
    if (first < last)
    {
        int mid = (first + last) / 2;
        mergeSort(objArray, tempArray, first, mid);
        mergeSort(objArray, tempArray, mid + 1, last);
        merge(objArray, tempArray, first, mid + 1, last);
    }
}

/** Merge two sorted halves of a subarray. */
private void merge(Comparable[] objArray, Comparable[] tempArray, int first, int mid, int last)
{
    int beginHalf1 = first;
    int endHalf1 = mid - 1;
    int beginHalf2 = mid;
    int endHalf2 = last;
    int index = first;
    int numElements = last - first + 1;

    // main loop
    while (beginHalf1 <= endHalf1 && beginHalf2 <= endHalf2)
    {
        if (objArray[beginHalf1].compareTo(objArray[beginHalf2]) < 0)
        {
            tempArray[index++] = objArray[beginHalf1++];
        }
        else
        {
            tempArray[index++] = objArray[beginHalf2++];
        }
    }

    // copy rest of left half
    while (beginHalf1 <= endHalf1)
    {
        tempArray[index++] = objArray[beginHalf1++];
    }

    // copy rest of right half
    while (beginHalf2 <= endHalf2)
    {
        tempArray[index++] = objArray[beginHalf2++];
    }

    // copy tempArray back into objArray
    for (int i = 0; i < numElements; i++, last--)
    {
        objArray[last] = tempArray[last];
    }
}
```

Figure 4.4. An implementation of the mergesort algorithm.

Quicksort

The Algorithm

In practice, quicksort is currently the fastest known sorting algorithm for data that reside entirely in main memory. This algorithm has been studied extensively and many variants have been developed over the years. The basic procedure for performing a quicksort is:

1. If the array is small enough use insertion sort instead.
2. Choose a pivot element that will be used to partition the elements.
3. Move the pivot element out of the way by placing it in the next to last position in the array.
4. Partition the array such that all elements < pivot are to the left of the pivot, and all elements >= pivot are to the right of the pivot.
5. Recursively quicksort the “left” half of the array (the half to the left of the pivot).
6. Recursively quicksort the “right” half of the array (the half to the right of the pivot, and including the pivot).

The book gives you the complete source code for one of the common variants of quicksort. It is relatively easy to follow. If you are unsure how quicksort works, take a look at the quicksort demo I’ve provided for this module. It will allow you to trace the execution of quicksort step by step. The source code I use for the demo is slightly different than the code in the text, but you should still be able to follow what is going on. The code is not broken down into as many pieces to make it easier to trace what’s going on, and the loops are structured a little differently, but you should be able to convince yourselves that the algorithm is the same.

Choosing the Pivot

The pivot step is crucial to achieving optimum performance from quicksort. Ideally, every time the pivot element is chosen it should divide the array into 2 equal-sized subarrays. However, depending on the elements in the array that may not be the case. Randomly choosing a single element as the pivot is a bad idea - suppose the smallest or largest element in the array is chosen? It would be too costly to examine every element to find the one that would yield 2 equal-sized subsets. As a compromise, a statistical method is used. The median-of-three method takes 3 elements that are easy to locate (the first, middle and last elements). Taking the median of a subpopulation of 3 elements doesn’t guarantee the ideal pivot, but it is still better than choosing any single element.

The problem with having subarrays that are very different in size is that more recursive calls are required. In the extreme case the pivot would divide the array into an empty subarray and a subarray containing all the elements, thus defeating the purpose of the divide and conquer approach, and severely impairing the algorithm’s performance.

Using Insertion Sort when the Array is Small

During the recursion, the subarray that gets sorted becomes increasingly smaller due to the repeated halving. Once the array reaches a certain threshold size, quicksort’s efficiency is compromised due to the overhead incurred by the recursive calls. For example, a subarray of 3 elements is completely sorted after the low, middle and high elements have been sorted, so it is redundant to choose a pivot element and partition such a small subarray. To avoid this problem, when the subarray reaches the threshold size it is more efficient to use an algorithm such as insertion sort. In practice, a minimum size in the range of 10 to 20 is supposed to work well.

Running Time

For quicksort, we deviate a little from the convention of using the worst-case as determining the characteristic running time. The best case running time is the same as the average case running time: O(*N log N*). The O(*N*) portion of the running time comes from the partitioning step, which must traverse all the items in the list. Repeated halving is always O(*log N*) provided the halving is "true" halving; i.e., the list is divided into two equal sized halves. If the division is unequal, the halving process will take longer because it must be performed more times. In the worst case the list will always be divided into an empty sublist plus a sublist containing all the elements. As a result, there will be *N* halvings rather than *log N* halvings, and the running time for the halving step will be O(*N*). Since partitioning occurs for every halving, the running times for partitioning and halving must be multiplied. In the best and average cases this results in O(*N log N*) running time, but in the worst case the running time will be O(N^2).

It turns out that O(*N log N*) is the best we can hope to achieve for an algorithm that sorts by comparing items. It has been shown that such algorithms must always do at least *N log N* comparisons (i.e., Ω(*N log N*)). The only way to get a faster running time is to sort without comparing items, which can be done. An example of such an algorithm is radix sort, described next.

Stable or Unstable?

Quicksort is an unstable sort.

Radix Sort

The Algorithm

Radix sort breaks the O(*N log N*) sorting barrier by sorting items without comparing them. The basic idea behind the radix sort algorithm is simple. It assumes, however, that the data can be processed as integers or strings, where each digit or character can be extracted from the rest of the item. You need a number of "buckets" equal to the number of different digits or characters that are possible (10 buckets if you're dealing with integers, for example). The pseudocode in the text does a good job of explaining the process. If you've ever tried to sort dates, you may have used a type of radix sort. First, you look at the year, then the month, and finally the day. The way in which radix sort works thus imposes a restriction on the kinds of data it can sort. However, even if your data can't be sorted directly using radix sort, you can always associate an alpha/numeric key with the items and sort the items by their keys instead.

The authors decline to show the actual source code for their algorithm. Although writing a radix sort is not difficult, it can be very tedious. Using only simple arrays to represent the buckets requires that the arrays be resized when their size limits have been reached. In Figure 4.5 I show you a very naive implementation of the authors' algorithm, designed to sort integers. It uses ArrayLists to represent the individual buckets, so no explicit resizing is needed. If the use of ArrayLists and iterators is confusing, don't worry—we'll be covering both of these topics soon. For now, it is enough to know that an ArrayList behaves like an array that can resize itself when it becomes full. An iterator is just a general mechanism for traversing a data structure (it works essentially the same as a for loop where the array indexes are specified).

Running Time

Radix sort can be O(*N*) in the worst case, provided the sizes of the data items are fixed, and the size of any given item is much smaller than the number of items to be sorted. The fixed size is important, since it renders the outer loop to be a O(1) statement rather than a O(*N*) statement. As long as the sizes of the items are very small compared to the total number of items, the number of times the outer loop executes becomes negligible. However, if the sizes of the items are very large (e.g., million character strings), the outer loop will dominate the running time.

Stable or Unstable?

As implemented in Figure 4.5, the radix sort is stable. All arrays (and ArrayLists) are processed from left to right, and no rearrangement occurs.

```
/**
* Naive implementation of a simple radix sort routine that works for integers
* number of digits in all integers must be <= maxDigits; integers do not have
* to be prefaced with leading zeroes
*
* An ArrayList is used to implement each bucket to avoid having to explicitly
* resize arrays
*/
public void radixsort(Comparable[] objArray, int first, int last, int maxDigits)
{
    int k; // index into objArray when placing items into objArray from buckets

    for (int i = 1; i <= maxDigits; i++)
    {
        clearBuckets(); // empty all buckets

        // place items into appropriate buckets
        for (int index = first; index <= last; index++)
        {
            int digit = ((Integer)objArray[index]).intValue();

            for (int j = 1; j < i; j++) // remove rightmost ith - 1 digits
            {
                digit /= 10;
            }

            digit = Math.abs(digit % 10); // extract ith digit; need absolute value for array index

            buckets[digit].add(objArray[index]);
        }

        // place items from buckets back into objArray
        k = 0;

        for (int j = 0; j <= 9; j++)
        {
            ArrayList a = (ArrayList)buckets[j];

            if (!a.isEmpty())
            {
                for (Iterator itr = a.iterator(); itr.hasNext(); )
                {
                    objArray[k++] = (Comparable)itr.next();
                }
            }
        }
    }
}

/**
* method to clear buckets
* rather than reset each array cell, the arrays are simply recreated
*/
private void clearBuckets()
{
    buckets = new ArrayList[10];

    for (int i = 0; i <= 9; i++)
    {
        buckets[i] = new ArrayList();
    }
}
```

Figure 4.5. Naive implementation of radix sort.

Which Sorting Algorithm Should You Use?

Of the sorting algorithms we've discussed, quicksort is the best choice for sorts where all the items can fit in main memory. If external storage must be used, mergesort is a good bet. Although radix sort has the best worst case performance, the restrictions imposed upon the kinds of data it can sort often makes it impractical. But if you know you have data that meet the restrictions, radix sort can be better than either quicksort or mergesort, provided the implementation is efficient. If you only need to sort a small number of items, quadratic algorithms such as selection sort and insertion sort will work fine, and if the number of items is small enough these algorithms can even be more efficient than quicksort.

Often the argument is made that you should choose one sorting algorithm over another based on how easy it is to implement the algorithm from scratch. Nowadays most of the commonly used search algorithms can probably be found already implemented in component libraries in many programming languages, so it's simply a matter of calling the appropriate method with the appropriate input. However, it is still important to understand how these algorithms work, not just to have some mastery over the concepts, but also in the event you need to implement a sorting algorithm in a language that does not already have an implementation of a suitable sorting algorithm within its libraries.

MSC07-J. Prevent multiple instantiations of singleton objects

The singleton pattern is an object-oriented design pattern. As defined by Gamma[1], the purpose of the singleton pattern is to:

Ensure a class only has one instance, and provide a global point of access to it.

In other words, there should only be one instance of a singleton class in an executing application, period. This concept also works with attributes and methods. For a singleton attribute or method, there is only one instance of that attribute or method, and all instances of the enclosing class reference this single instance of the attribute or method.

In Java, singletons are declared using the keyword, `static`. Attributes that are declared as `static` must be initialized at the point of declaration. For `static` reference types, a single instance of that type is created at the time of declaration. New instances of that attribute cannot be created, although Java does allow for the values of `static` (but not `final`) attributes to be changed, but only via `static` methods. It is important to understand that while a `static` object itself cannot be re-instantiated, the contents of that `static` object can be manipulated without these restrictions. For example, if you have a `static ArrayList`, you cannot change the reference of that `ArrayList` by re-instantiating it or by re-assigning it, but you can freely add items to the list, remove items from it, or modify items it contains. For a method declared as `static`, there will be a single, shared instance of that method for all instances of the enclosing class. A method declared as `static` must provide an implementation; i.e., Java does not allow `static` methods to be declared as `abstract`.

To prevent multiple instantiations of a singleton class, one or more of the following techniques should be used, as appropriate for the circumstances under which the class will be used:

1. *Make its constructor private.* Reducing the accessibility to `private` ensures that untrusted code will not be able to create additional instances of the class. An example of this is shown below.

```
class MySingleton
{
      private static final MySingleton Instance = new MySingleton();

      private MySingleton()
      {
            // private constructor prevents instantiation by untrusted
            // callers
      }

      public static synchronized MySingleton getInstance()
      {
            return Instance;
      }
}
```

2. *Employ lock mechanisms to prevent an initialization routine from running simultaneously by multiple threads.* In a multi-threaded application it is possible for different threads to create multiple instances of a singleton class. One way to circumvent this is to declare a method that returns the reference to the singleton class, and declare that method using the synchronized keyword, as shown below.

```
class MySingleton
{
      private static MySingleton Instance;

      private MySingleton()
      {
            // private constructor prevents instantiation by untrusted
            // callers
      }

      // Lazy initialization
      public static synchronized MySingleton getInstance()
      {
            if (Instance == null)
            {
                  Instance = new MySingleton();
            }

            return Instance;
      }
}
```

Other approaches can also be used, such as using a double-checked locking pattern, or the initialize-on-demand pattern. An example of each approach is shown below.

```
// Using the double-checked locking pattern
class MySingleton
{
      private static volatile MySingleton Instance;

      private MySingleton()
      {
            // private constructor prevents instantiation by untrusted
            // callers
      }

      // Double-checked locking
      public static MySingleton getInstance()
      {
            if (Instance = = null)
            {
                  synchronized (MySingleton.class)
                  {
                        if (Instance == null)
                        {
                              Instance = new MySingleton();
                        }
                  }
            }

            return Instance;
      }
}
```

```
// Using the initialize-on-demand pattern - uses a static inner class
// to create the singleton instance
class MySingleton
{
      static class SingletonHolder
      {
            static MySingleton Instance = new MySingleton();
      }

      public static MySingleton getInstance()
      {
            return SingletonHolder.Instance;
      }
}
```

3. *Ensure the class is not serializable. Serialization allows an object to be written to a file and retrieved later.* The problem with this with respect to singletons is that it is possible to read in a serialized singleton, deserialize it, and then create additional instances of it. To avoid this problem it is recommended that an `enum` be used as the singleton, as shown in the following example. This works because an `enum` provides serializability if it is required, but the way an `enum` is serialized and deserialized is different from the way ordinary objects are serialized and deserialized, as stated in the Java SE documentation:

   "The rules for serializing an `enum` instance differ from those for serializing an "ordinary" serializable object: the serialized form of an `enum` instance consists only of its `enum` constant name, along with information identifying its base `enum` type. Deserialization behavior differs as well—the class information is used to find the appropriate `enum` class, and the `Enum.valueOf` method is called with that class and the received constant name in order to obtain the `enum` constant to return."

```
public enum MySingleton
{
      private static MySingleton Instance;

      // non-transient field
      private String[] str = {" one", "two", "three"};

      public void displayStr()
      {
            System.out.println(Arrays.toString(str));
      }
}
```

4. *Ensure the class cannot be cloned.* Cloning is one way to obtain additional instances of a class without directly invoking the class' constructor. To prevent allowing multiple instances of a singleton class from being created, the class should not implement the `Cloneable` interface. If the class must indirectly implement the `Cloneable` interface via inheritance, the class' `clone()` method must be overridden with one that throws a `CloneNotSupportedException`. An example of this is shown in the following example:

```
class MySingleton implements Cloneable
{
      private static MySingleton Instance; private MySingleton()
      {
            // private constructor prevents instantiation by untrusted
            // callers
      }

      // Lazy initialization
      public static synchronized MySingleton getInstance()
      {
            if (Instance == null)
            {
                  Instance = new MySingleton();
            }

            return Instance;
      }

      public Object clone() throws CloneNotSupportedException
      {
            throw new CloneNotSupportedException();
      }
}
```

5. Prevent the class from being garbage-collected if it was loaded by a custom class loader. Recall that an object can be garbage-collected when there are no longer any other objects holding a reference to the object. This includes the class loader that was used to load the class in the first place. Once an object's class loader becomes eligible for garbage collection, so does the object. This most often happens when custom class loaders are used. Garbage collection of a singleton object presents a problem if the singleton must persist throughout the lifetime of the application.

   The solution is to ensure there is always a live thread that maintains a reference to the singleton object. The singleton's class loader will then never become eligible for garbage collection, and thus neither will the singleton object. This can be accomplished in several ways, but one way of doing this that allows for multiple singletons to be protected is to create a special class that stores references to the singletons in a `ConcurrentHashMap`, and keeps itself alive for the duration of the Java Virtual Machine by putting it under the control of a daemon thread.

References:

1. Gamma, E., Helm, R., Johnson, R., and Vlissides, J. M., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional, Boston, 1995.