Searching: The Linear Search

Searching is the process of navigating through some data set, an array or a List, in an attempt to find a matching element. If a matching element is found, the index of the element is returned and if no match is found then a flag value (usually -1) is returned instead.

The simplest search algorithm to understand and code is the LINEAR SEARCH. This search algorithm is so simple it practically writes itself:

- Iterate over each element
- If the element matches, return it's index
- If all the elements are exhausted, then there is no match, return -1

Searching is easier to visualize using names so all the examples will use the following array:

```
String[] names={"Charlie","Becky","Adam","Fred","Ethan","Dennis"};
```

Which can be summarized by listing it’s type, indexes, and values:

String[] names

index	Value
0	"Charlie"
1	"Becky"
2	"Adam"
3	"Fred"
4	"Ethan"
5	"Dennis"

Searching an Array

Before considering the algorithm for searching an array, consider the result of such a search. Using the example names array, if the value to be searched for was “Dennis” then the result of such a search could have several different forms.

The value searched for, “Dennis,” could be returned, but that wouldn’t be terribly useful since the value was already known, after all it was the search value. A simple boolean result could be returned, the value was either in the array or it wasn’t. However, the most useful piece of information would be the index of the array element where the value was found!

If the value isn’t in the array, then a value that would not normally be returned can indicate such. This is how the String method .indexOf() works. When the value can’t be found in the string, -1 is returned because it isn’t a possible index.

Below is a summary of the information about the method to be written:

Method Attribute	Description	JAVA
Return Type	Returning an array index	`int`
Method Name	This method will search an array for a value	`linearSearch`
1st Parameter	The array to search, using a generic name	`String[] array`
2nd Parameter	The value to search for, or find	`String find`
Return value(s)	If the string to be found is in array, return the index	`0 to array.length-1`
	If the string to be found isn’t in the array, return -1	`-1`

With this information the method header can be written:

```
public static int linearSearch(String[] array, String find)
```

Recall that the proper way to compare two String values for equality is the .equals() method instead of the == operator. However, this will introduce possible complications. Recall as well that a String may have the special value null, in which case trying to use the .equals() method will generate a “NullPointerException” so it is important to first make sure that each String is not null before trying to use the .equals() method.

Such a test can be written as a simple “if” statement: `if (find==null)`

If the find variable is null, then a separate kind of equality test can be performed to see if the array value is also null!

Comparing find to each value in the array inside a “for loop” will complete the LINEAR SEARCH algorithm.

Putting it all together we can write the following method:

```
01 public static int linearSearch(String[] array, String find) {
02    for (int i=0; i<array.length; i++) {
03       if (find!=null) {
04          if (array[i]==null) {
05             return i;
06          }
07       } else {
08          if (find.equals(array[i])) {
09             return i;
10          }
11       }
12    }
13    return -1;
14 }
```

Line `01` is the method header from the prior page.

Line `02` creates the iterator variable `i` that will count from `0` to the end of the array.

Line `03` tests to see if the `String find` has a `null` value.

Line `04` executes if `find` was `null`. It checks to see if the element at index `i` also has the value `null`.

If the expression in the conditional of Line `04` is `true` Line `06` returns the current value of `i`, the index of the first matching value for `find`.

Line `07` begins the else code block that gets executed if the `String find` is not `null`.

The conditional in Line `08` uses the `.equals()` method to see if the current value at index `i` is the same as `find`.
If the expression in the conditional of Line `08` is `true` Line `09` returns the current value of `i`, the index of the first matching value for `find`.

Finally, Line `13` returns `-1` if the “for loop” has iterated over all the values in the array without finding a match!

Searching for a value that is an `int`, `double`, or `boolean` is much simpler because those types always have a value so it is not necessary to check for `null` values.

Typically a method like search array would have the return value stored in a variable for later use.

```
01 public static void main(String[] args) {
02    String[] names={"Adam","Becky","Charlie","Dennis","Ethan","Fred","Gary","Heath"};
03    System.out.println("Enter to exit program");
04    String find=ConsoleInput.inString("Search for Name:");
05    while (find.length()!=0) {
06       int index=searchArray(names,find);
07       if (index!=-1) {
08          System.out.println(names[index]+" is element#"+index);
09       } else {
10          System.out.println(find+" is not in the array!");
11       }
12       find=ConsoleInput.inString("Search for Name:");
13    }
14 }
```

The code above uses the `ConsoleInput` class to allow a user to type a name. Notice that the loop is primed in line `04` by getting console input before the main loop begins. Writing the code this way allows the user to enter a blank `String` to exit the program without performing any search.

After that, the loop searches for the `String find` and either identifies where it is in the array or indicates that the `String` entered isn't in the array!

Searching an Array for a Minimum or Maximum Value

int[] numbers

index	Value
0	12
1	15
2	11
3	9
4	10

It will be easier to consider searching for a min or max value considering numbers, but the process can be applied to String arrays as well.

Now examine a new data set for this example, an `int` array called `numbers`:

```
int[] numbers={12,15,11,9,10};
```

A quick glance at the table reveals that the smallest value, "9," is stored at index 3.

However, if this array contained a list of a million numbers, it would take much more than a glance to establish what the smallest value was in the list. In that case, an algorithm is truly needed!

Chart A: Finding a Minimum

index	Value	Low Index	Low Value	Compare	Next Low Index	Next Low Value
0	12	0	12	Assume	0	12
1	15	0	12	12<15	0	12
2	11	0	12	11<12	2	11
3	9	2	11	9<11	3	9
4	10	3	9	10<9	3	9

Use Chart A to follow this process step by step:

Begin by assuming the first value in the list, the one at index 0, is the smallest value.

Now compare the value stored at the current low index of 0 with the one stored at index 1. Since the value stored at index 0 is less than the value at index 1, index 0 remains the low index.

Next compare the value stored at the current low index of 0 with the one stored at index 2. Since the value stored at index 0 is greater than the value at index 2, index 2 becomes the new low index.

Then compare the value stored at the current low index of 2 with the one stored at index 3. Since the value stored at index 2 is greater than the value at index 3, index 3 becomes the new low index.

Finally, compare the value stored at the current low index of 3 with the one stored at index 4. Since the value stored at index 3 is less than the value at index 4, index 3 remains the new low index.

Notice that each of the comparison steps followed the same procedure, only the indexes to be compared each time changed. The first index, the current low index could easily be considered a variable and the second the second index is simply a loop iterator that starts at 1 and ends at end of the array!

This is how a LINEAR SEARCH algorithm works. Why record the index instead of the actual number?

The answer once again is a question of getting the maximum information possible. If you have the index you can get the value, but if you only have the value it is impossible to know the index too.

Now that the algorithm has been analyzed consider the method header, also known as a METHOD SIGNATURE. Just like searching for a matching value, we can construct a table to help analyze the method:

Below is a summary of the information about the method to be written:

Method Attribute	Description	JAVA
Return Type	Returning an array index	`int`
Method Name	This method will return the index of the lowest value in the array	`minIndex`
Parameter	The array to search, using a generic name	`int[] array`
Return value	Index of the element with the lowest value	`0 to array.length-1`

Notice that there are a few differences from the `linearSearch` method. The return type will still be an `int` because an array index value will be returned, but there is only one parameter, an `int` array.

Finally, notice that some index value will always be returned because **some** value in the array must be the smallest value!

Putting it all together:

```
01 public static int minIndex(int[] array) {
02    int currentMin=0;
03    for (int i=1; i<array.length; i++) {
04       if (array[i]<array[currentMin]) {
05          currentMin=i;
06       }
07    }
08    return currentMin;
09 }
```

Line `01` is the method header from the prior page.

Line `02` creates the variable `currentMin` and sets its initial value to 0 because it the algorithm assumes that the first value in the array is the minimum.

Line `03` creates the iterator variable `i` that will count from `1` to the end of the array. Notice that the first array value will be skipped by the loop because it is already assumed to be the minimum!

Line `04` tests to see if the value stored at index `i` is less than the value stored at index `currentMin`.

Line `05` executes if the expression in Line `04` was `true`. It is `true` only if a new minimum value was found. If a new minimum was found, then record its index, `i`, as the new `currentMin` by assigning `currentMin` the value of `i`.

Finally, Line `08` returns the value stored in `currentMin`, the index of the minimum value is this array!

To test this method, simply create an `int` array in the `main()` method, adapts the `outputArray()` method to work for an `int` array and then see if the correct value is returned.

```
public class FindMinInArray {
  public static void main(String[] args) {
   int[] numbers={12,15,11,9,10};
   outputArray(numbers,"numbers");
   int minI=minIndex(numbers);
   System.out.println("Minimum value: "+numbers[minI]);
   System.out.println("At index: "+minI);
  }
  public static int minIndex(int[] array) {
   int currentMin=0;
   for (int i=1; i<array.length; i++) {
      if (array[i]<array[currentMin]) {
         currentMin=i;
      }
   }
   return currentMin;
  }
  public static void outputArray(int[] arr, String arrName) {
   for (int i=0; i<arr.length; i++) {
      System.out.println(arrName+"["+i+"]="+arr[i]);
   }
  }
}
```

Console Output:

```
numbers[0]=12
numbers[1]=15
numbers[2]=11
numbers[3]=9
numbers[4]=10
Minimum value: 9
At index: 3
```

Finding a minimum or maximum value that is an `int` or `double` is much simpler than with a `String` because those types can be compared using simple relational operators greater (>) and less than (<).

To perform the same kind of comparison for two `String` values, the `String` method `.compareTo()` must be used!

But remember, `String` values may be `null`, in which case trying to use a `String` method will generate a "`NullPointerException`" so it is important to first make sure that each `String` is not `null` before trying to use the `.compareTo()` method.

Using the `.compareTo(String)` Method

The `.compareTo()` method can only be used on a String that has a value, assume the following declarations:

```
String s1="a",s2="b";
```

To compare `s1` to `s2` to see the LEXICOGRAPHIC relationship (a fancy word that essentially means alphabetical) between the two `String` values, use the `.compareTo()` method on `s1` with `s2` as the parameter like this:

```
s1.compareTo(s2)
```

The `.compareTo()` method returns an `int` value that is less than, greater than, or equal to 0 depending upon the relationship between the two `String` values.

The following table summarizes how the `.compareTo()` method works:

Relationship between `str` and `anotherStr`	`str`	`anotherStr`	Example(s)	Result
`str` lexicographically precedes `anotherStr`	`"a"`	`"b"`	`str.compareTo(anotherStr)`	`< 0`
`str` lexicographically follows `anotherStr`	`"b"`	`"a"`	`str.compareTo(anotherStr)`	`> 0`
`str` is lexicographically identical to `anotherStr`	`"a"`	`"a"`	`str.compareTo(anotherStr)`	`0`

When finding a minimum or maximum value for `int` and `double` arrays there must always be a minimum and maximum, but a `String` array might be filled with all `null` values, in which case it would make sense to return -1 instead of a valid index just like the linear search method does.

Considering the changes that would be necessary for a `minIndex(String[] array)` method the following summary can be made:

- If the current value pointed to by `currentMin` is a `null` value, then the current index position should automatically be the new `currentMin`.
- If the value at the current index position is `null`, then don't compare it to the value at `currentMin`.
- If the final array value indicated by `currentMin` is `null`, then return `-1` instead of `currentMin` because it means all the array values were `null`!