

4.14 Searching Algorithms

Self-Study Guide · About 55 minutes · With answer key

By the end you will be able to

- Explain how a linear search checks the elements of an array or ArrayList one at a time, in order, until it finds the target or runs out of elements.
- Trace a linear search and give the index it returns, including the value -1 when the target is not there.
- Explain why a search stops at the first match, and spot searches that stop too early or never stop.
- Count the comparisons a linear search makes with a counter variable, and say when it makes the fewest and the most.
- Search a list of strings or objects using equals, and search from the last element toward the first.
- Search a 2D array by taking each row in turn and running a linear search on that row.

What to remember from earlier

- Array indices run from 0 to length minus 1. ([Topic 4.3: Array Creation and Access, 4.3.6 Valid Indexes and Out-of-Bounds Errors](#))
- Arrays can be traversed with indexed for, while, and enhanced for loops. ([Topic 4.4: Array Traversals](#))
- An ArrayList uses size() and get(i) to traverse. ([Topic 4.9: ArrayList Traversals, 4.9.1 Visiting Every Element by Index](#))
- grid.length counts rows and grid[0].length counts columns. ([Topic 4.11: 2D Array Creation and Access, 4.11.5 Counting Rows and Columns](#))
- Use equals, not ==, to compare the contents of strings. ([Topic 2.6: Comparing Boolean Expressions, 2.6.6 Comparing Objects with equals](#))
- With a && b, Java skips b when a is false. ([Topic 2.5: Compound Boolean Expressions, 2.5.5 Short-Circuit Evaluation](#))

4.14.1 Checking Each Element in Order

You have a list of quiz scores and want to know whether anyone scored exactly 90, and if so, **where** in the list that score is. In Topic 4.5 you learned to decide whether at least one element has a property, which answers "is it there?" A program often needs more: the position of the value, so it can read the student's name from a matching list, change the score, or remove it.

Think about how you would find a friend's locker number on a printed list with no particular order. You start at the top and read each entry. If it matches, you stop and note where it was. If it does not, you move to the next one. If you reach the bottom without a match, you know the locker is not on the list. There is no shortcut, because an unordered list gives you no hint about where the value might be.

That procedure is called a **linear search** (also called a **sequential search**): check each element in order, starting at one end, until you find the value you want or you have checked every element.

The value you are looking for is called the **target**. Linear search is one of the standard algorithms every Java programmer knows, and it works on an array, an `ArrayList`, or (row by row) a 2D array. Here is linear search written as a method. It takes the array to search and the target, and returns the index where the target was found.

WORKED EXAMPLE Finding a score

Look at three things: the loop visits indexes 0 through `values.length - 1`, the `if` on line 7 compares one element with the target, and there are two different `return` statements.

LinearSearchBasic.java

```

1 public class LinearSearchBasic
2 {
3     public static int findIndex(int[] values, int target)
4     {
5         for (int i = 0; i < values.length; i++)
6         {
7             if (values[i] == target)
8             {
9                 return i;
10            }
11        }
12        return -1;
13    }
14
15    public static void main(String[] args)
16    {
17        int[] scores = {72, 85, 90, 64, 85};
18        System.out.println("Index of 90: " + findIndex(scores, 90));
19        System.out.println("Index of 70: " + findIndex(scores, 70));
20    }
21 }
```

```

Index of 90: 2
Index of 70: -1
```

Trace the first call, `findIndex(scores, 90)`:

i	values[i]	values[i] == 90	What happens
0	72	false	Move on
1	85	false	Move on
2	90	true	return 2; ends it

The moment line 9 runs, the method ends. Remember from Topic 3.5 that `return` leaves the method right away, even from inside a loop, so indexes 3 and 4 are never looked at. The call evaluates to 2, and `main` prints it.

Now the second call, `findIndex(scores, 70)`. The loop compares 72, 85, 90, 64, and 85 with 70. Every comparison is false, so `return i;` never runs. When `i` becomes 5, the condition `5 < 5` is false and the loop ends. Java moves to line 12 and returns -1.

Why `-1`? The method must return an `int` no matter what, so it needs some `int` that means "not found." It cannot be `0`, because `0` is a real index: a return value of `0` would mean "found at the very first position." Every valid index is between `0` and `length - 1`, so any negative number would be safe, and `-1` is the value Java programmers agree on. You have seen this convention before: the `String` method `indexOf` from Topic 1.15 also returns `-1` when the text it looks for is not in the string. A value like this, chosen because it can never be a real answer, is a **sentinel value**.

KEY IDEA

A linear search checks each element in order. When an element equals the target it returns that index right away; if the loop finishes without a match, it returns `-1`, which can never be a valid index.

WATCH OUT

Giving up after the first element. A very common first attempt puts "not found" in an `else`, right next to "found":

EarlyReturnBug.java

```
1 public class EarlyReturnBug
2 {
3     public static int findIndex(int[] values, int target)
4     {
5         for (int i = 0; i < values.length; i++)
6         {
7             if (values[i] == target)
8             {
9                 return i;
10            }
11            else
12            {
13                return -1;
14            }
15        }
16        return -1;
17    }
18
19    public static void main(String[] args)
20    {
21        int[] scores = {72, 85, 90, 64, 85};
22        System.out.println("Index of 90: " + findIndex(scores, 90));
23        System.out.println("Index of 72: " + findIndex(scores, 72));
24    }
25 }
```

```
Index of 90: -1
Index of 72: 0
```

On the very first pass, `i` is `0`. If `values[0]` matches, the method returns `0`. If it does not match, the `else` returns `-1`. Either way the method ends during the first pass, so it only ever looks at one element. It works for `72` only because `72` happens to be first. Students write this

because "if it matches, say found; otherwise, say not found" sounds right in English. But one mismatch does not prove the target is missing; only checking **every** element does. The fix is to delete the `else` and keep the single `return -1;` **after** the loop, as in `LinearSearchBasic`. (The `return -1;` after the loop is also why Java does not complain about a missing `return` statement here: it has no way to know the loop body will run at least once.)

PAUSE AND TRY

Question 1 Multiple choice

The `findIndex` method from `LinearSearchBasic` is called with this array:

```
1 int[] temps = {38, 45, 41, 50, 41};
```

What does `findIndex(temps, 41)` return?

- A. 2
- B. 41
- C. 4
- D. 3

Question 2 Short answer

A student suggests that `findIndex` should return 0 when the target is not found, "because zero means nothing." Explain why this would cause a problem.

4.14.2 Stopping at the First Match

The array `scores` contains 85 twice, at index 1 and at index 4. What should a search for 85 report? The standard linear search reports **index 1**, the first match it reaches, because it stops there. Stopping is not only about speed. It decides **which** match you get when the target appears more than once.

There are two common ways to make a search stop. The `return` inside the loop that you saw in 4.14.1 is one. The other is to build the stopping rule into a `while` loop's condition, which is useful when the search is written directly in `main` instead of in its own method.

WORKED EXAMPLE A search that stops by its loop condition

Read the loop condition as "keep going while there are elements left **and** the current element is not the target."

```

1 int[] scores = {72, 85, 90, 64, 85};
2 int target = 64;
3 int index = 0;
4 while (index < scores.length && scores[index] != target)
5 {
6     index++;
7 }
8 if (index < scores.length)
9 {
10     System.out.println("Found " + target + " at index " + index);
11 }
12 else
13 {
14     System.out.println(target + " not found");
15 }

```

Found 64 at index 3

Check	index	index < 5	scores[index] != 64	Keep going?
1	0	true	72: true	yes
2	1	true	85: true	yes
3	2	true	90: true	yes
4	3	true	64: false	no, stop

The loop body does only one thing: move `index` forward. All of the searching happens in the condition. When the loop ends, `index` is still 3, which is less than the length, so the `if` after the loop knows the target was found and reports where.

Now change one thing: search for 70, which is not in the array.

```

1 int[] scores = {72, 85, 90, 64, 85};
2 int target = 70;
3 int index = 0;
4 while (index < scores.length && scores[index] != target)
5 {
6     index++;
7 }
8 System.out.println("index is " + index);
9 if (index < scores.length)
10 {
11     System.out.println("Found " + target + " at index " + index);
12 }
13 else
14 {
15     System.out.println(target + " not found");
16 }

```

index is 5
70 not found

This time `scores[index] != 70` is true for every element, so the loop moves `index` all the way to 5. At the next check, `index < 5` is false and the loop ends. An `index` equal to the length means "every element was checked," which is why the `if` after the loop tests `index < scores.length` to tell the two outcomes apart.

There is something important hiding in that last check. When `index` is 5, `scores[5]` does not exist. Java never tries to read it, because of **short-circuit evaluation** from Topic 2.5: in a `&& b`, if `a` is false, Java does not evaluate `b`. The test `index < scores.length` comes first, so it guards the array access that follows it.

KEY IDEA

A linear search stops at the first match, so when the target appears more than once it reports the first one it reaches. In a `while` condition, put the index check before the element check so short-circuit evaluation protects the array access.

WATCH OUT

Finding a match but not stopping. This version records the index of a match, but the loop keeps going:

KeepLookingBug.java

```
1 public class KeepLookingBug
2 {
3     public static int findIndex(int[] values, int target)
4     {
5         int foundAt = -1;
6         for (int i = 0; i < values.length; i++)
7         {
8             if (values[i] == target)
9             {
10                 foundAt = i;
11             }
12         }
13         return foundAt;
14     }
15
16     public static void main(String[] args)
17     {
18         int[] scores = {72, 85, 90, 64, 85};
19         System.out.println("Index of 85: " + findIndex(scores, 85));
20     }
21 }
```

Index of 85: 4

`foundAt` becomes 1 when `i` is 1, but the loop carries on, and when `i` is 4 the second 85 overwrites it with 4. The method reports the **last** match, not the first, and it always checks every element even when the answer was found early. Students write this because "remember where you saw it" feels like enough. Fix it by returning as soon as there is a match (`return i;` inside the `if`, as in `LinearSearchBasic`), which gives Index of 85: 1.

Putting the two checks in the wrong order. Swapping the parts of the `while` condition looks harmless, since `&&` means "both must be true":

```
1 int[] scores = {72, 85, 90, 64, 85};
2 int target = 70;
3 int index = 0;
4 while (scores[index] != target && index < scores.length)
5 {
6     index++;
7 }
8 System.out.println(target + " not found");
```

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 5 out of bounds for length 5

When the target is missing, `index` reaches 5, and the condition evaluates `scores[5]` **first**, before it ever reaches `index < scores.length`. That access is out of bounds, so the program crashes before printing anything (the exact wording of the message can vary between Java versions). Put the index check first: `index < scores.length && scores[index] != target`.

PAUSE AND TRY

Question 3 Tracing

What does this code print?

```
1 int[] codes = {4, 9, 2, 9, 7};
2 int target = 9;
3 int index = 0;
4 while (index < codes.length && codes[index] != target)
5 {
6     index++;
7 }
8 System.out.println("index: " + index);
9 target = 6;
10 index = 0;
11 while (index < codes.length && codes[index] != target)
12 {
13     index++;
14 }
15 System.out.println("index: " + index);
```

Question 4 Multiple choice

Which change to the `for` loop header in `KeepLookingBug` makes `findIndex(scores, 85)` return 1, while still returning -1 for a target that is not in the array?

- A. `for (int i = 0; i < values.length && foundAt == -1; i++)`
- B. `for (int i = 0; i < values.length && foundAt != -1; i++)`
- C. `for (int i = 1; i < values.length; i++)`
- D. `for (int i = 0; i <= values.length; i++)`

4.14.3 Counting Comparisons

In Topic 2.12 you measured how much work code does by counting how many times a statement runs, using a **counter variable**. For a search, the natural thing to count is the **comparisons**: how many times the search compares an element with the target. Each comparison is one step of "is this the one?"

Unlike the loops you counted in Topic 2.12, a search does not always do the same amount of work. Finding the target at index 0 takes one comparison. Finding it at index 2 takes three: indexes 0, 1, and 2. In general, a target found at index k costs $k + 1$ comparisons, because the search checks every index from 0 up to and including k . A target that is not there costs one comparison for **every** element, because only after checking all of them can the search be sure.

WORKED EXAMPLE Counting with a class variable

`comparisons` is a class variable (Topic 3.7), so both `findIndex` and `main` can use it. Look for two things: `findIndex` resets it to 0 at the start of each search, and `comparisons++` sits right before the `if`, so it counts every comparison, including the one that finds the target.

CountComparisons.java

```
1 public class CountComparisons
2 {
3     private static int comparisons = 0;
4
5     public static int findIndex(int[] values, int target)
6     {
7         comparisons = 0;
8         for (int i = 0; i < values.length; i++)
9         {
10             comparisons++;
11             if (values[i] == target)
12             {
13                 return i;
14             }
15         }
16         return -1;
17     }
18
19     public static void main(String[] args)
20     {
21         int[] scores = {72, 85, 90, 64, 85};
22         int[] targets = {72, 90, 85, 70};
23         for (int target : targets)
24         {
25             int result = findIndex(scores, target);
26             System.out.println(target + ": index " + result + ", " + comparisons +
27 " comparisons");
28         }
29 }
```

```
72: index 0, 1 comparisons
90: index 2, 3 comparisons
85: index 1, 2 comparisons
70: index -1, 5 comparisons
```

Target	Where it is	Elements compared	Comparisons
72	index 0	72	1
90	index 2	72, 85, 90	3
85	index 1 (and index 4)	72, 85	2
70	not in the array	all 5 elements	5

Each count matches the rule: found at index k means $k + 1$ comparisons, and not found means one comparison per element.

The best case for a linear search is that the target is the first element checked: 1 comparison. The worst case is that the target is the last element checked, or not there at all: one comparison per element. So the worst-case work grows with the size of the list. This program searches a 10-element array and a 20-element array. Recall from Topic 4.3 that `new int[10]` fills every element with 0, so 99 is never found and 0 is found right away.

CountGrowth.java

```
1 public class CountGrowth
2 {
3     private static int comparisons = 0;
4
5     public static int findIndex(int[] values, int target)
6     {
7         comparisons = 0;
8         for (int i = 0; i < values.length; i++)
9         {
10             comparisons++;
11             if (values[i] == target)
12             {
13                 return i;
14             }
15         }
16         return -1;
17     }
18
19     public static void main(String[] args)
20     {
21         int[] small = new int[10];
22         int[] large = new int[20];
23         findIndex(small, 99);
24         System.out.println("10 elements, target missing: " + comparisons + "
25         comparisons");
26         findIndex(large, 99);
27         System.out.println("20 elements, target missing: " + comparisons + "
28         comparisons");
29         findIndex(large, 0);
30         System.out.println("20 elements, target first: " + comparisons + "
31         comparisons");
32     }
33 }
```

```
10 elements, target missing: 10 comparisons
20 elements, target missing: 20 comparisons
20 elements, target first: 1 comparisons
```

Doubling the length doubled the work of a failed search, the same "single loop" growth you saw in Topic 2.12. A lucky search is fast no matter how long the list is.

KEY IDEA

A linear search that finds the target at index k makes $k + 1$ comparisons. If the target is missing, it makes one comparison per element, which is also the most it can ever make.

WATCH OUT

Putting the counter where it misses a comparison. Here `comparisons++` comes after the `if`:

CounterPlacementBug.java

```
1 public class CounterPlacementBug
2 {
3     private static int comparisons = 0;
4
5     public static int findIndex(int[] values, int target)
6     {
7         comparisons = 0;
8         for (int i = 0; i < values.length; i++)
9         {
10             if (values[i] == target)
11             {
12                 return i;
13             }
14             comparisons++;
15         }
16         return -1;
17     }
18
19     public static void main(String[] args)
20     {
21         int[] scores = {72, 85, 90, 64, 85};
22         findIndex(scores, 90);
23         System.out.println("90: " + comparisons + " comparisons");
24         findIndex(scores, 70);
25         System.out.println("70: " + comparisons + " comparisons");
26     }
27 }
```

```
90: 2 comparisons
70: 5 comparisons
```

Finding 90 really takes three comparisons (72, 85, 90), but this counter says 2. On the pass where the comparison is true, `return i;` leaves the method before `comparisons++` gets a chance to run, so the successful comparison is never counted. The failed search still shows 5 because no `return` cuts it short, which makes the bug easy to miss. As Topic 2.12 advised, put the counter right next to the thing you are counting, before anything that could end the method: `comparisons++;` goes before the `if`.

PAUSE AND TRY

Question 5 Multiple choice

This array is searched with the counting `findIndex` method from `CountComparisons`.

```
1 int[] ages = {15, 17, 14, 16, 18, 13, 17, 15};
```

What are the returned index and the value of `comparisons` after `findIndex(ages, 13)`?

- A. Index 5, 6 comparisons
- B. Index 5, 5 comparisons

- C. Index 6, 6 comparisons
- D. Index 5, 8 comparisons

Question 6 Short answer

A linear search runs on an array of 200 values. What is the fewest number of comparisons it can make, and what is the most? Describe exactly when each one happens.

4.14.4 Searching an ArrayList of Strings

Searching a class roster for a student's name is the same idea as searching for a score, but two details change. The list is often an `ArrayList`, and the elements are `String` objects instead of `int` values. Both details change the code, and one of them changes it in a way the compiler will not catch.

From Topic 4.8, an `ArrayList` uses `size()` instead of `length` and `get(i)` instead of square brackets. From Topic 1.15, two strings are compared with `equals`, which checks whether they hold the same characters. With those two swaps, linear search carries over directly.

WORKED EXAMPLE Searching a roster

Compare this method line by line with `findIndex`: the loop runs while `i < names.size()`, and the comparison is `names.get(i).equals(target)`.

SearchNames.java

```
1 import java.util.ArrayList;
2
3 public class SearchNames
4 {
5     public static int findName(ArrayList<String> names, String target)
6     {
7         for (int i = 0; i < names.size(); i++)
8         {
9             if (names.get(i).equals(target))
10            {
11                return i;
12            }
13        }
14        return -1;
15    }
16
17    public static void main(String[] args)
18    {
19        ArrayList<String> roster = new ArrayList<String>();
20        roster.add("Leo");
21        roster.add("Priya");
22        roster.add("Maya");
23        roster.add("Sam");
24        System.out.println("Maya: " + findName(roster, "Maya"));
25        System.out.println("Omar: " + findName(roster, "Omar"));
26        System.out.println("maya: " + findName(roster, "maya"));
27    }
28 }
```

```
Maya: 2
Omar: -1
maya: -1
```

The search for "Maya" compares "Leo", "Priya", then "Maya", and returns 2. "Omar" matches nothing, so the loop checks all four names and the method returns -1. The third result is the variation to notice: "maya" with a lowercase m is not found. `equals` compares characters exactly, and 'm' and 'M' are different characters, so as far as `equals` is concerned "maya" is a different string.

Sometimes you only need to know **whether** the target is present, not where. Then the method can return a `boolean`, and an enhanced `for` loop (Topic 4.4) works well, because no index is needed. This is the "at least one element has a property" algorithm from Topic 4.5, with "equals the target" as the property.

HasName.java

```
1 import java.util.ArrayList;
2
3 public class HasName
4 {
5     public static boolean hasName(ArrayList<String> names, String target)
6     {
7         for (String name : names)
8         {
9             if (name.equals(target))
10            {
11                return true;
12            }
13        }
14        return false;
15    }
16
17    public static void main(String[] args)
18    {
19        ArrayList<String> roster = new ArrayList<String>();
20        roster.add("Leo");
21        roster.add("Priya");
22        roster.add("Maya");
23        roster.add("Sam");
24        System.out.println(hasName(roster, "Sam"));
25        System.out.println(hasName(roster, "Omar"));
26    }
27 }
```

```
true
false
```

The enhanced `for` loop variable `name` holds a copy of each element in turn, but there is no index variable. That is why `hasName` can say yes or no but could not return a position. Use an indexed loop whenever the search must report where the target is.

KEY IDEA

To search an `ArrayList`, loop while `i` is less than `size()` and read each element with `get(i)`. Compare strings and other objects with `equals`, never with the double-equals operator.

WATCH OUT

Comparing strings with `==`. This version is identical to `SearchNames` except for the comparison:

EqualsBug.java

```
1 import java.util.ArrayList;
2
3 public class EqualsBug
4 {
5     public static int findName(ArrayList<String> names, String target)
6     {
7         for (int i = 0; i < names.size(); i++)
8         {
9             if (names.get(i) == target)
10            {
11                return i;
12            }
13        }
14        return -1;
15    }
16
17    public static void main(String[] args)
18    {
19        ArrayList<String> roster = new ArrayList<String>();
20        roster.add("Leo");
21        roster.add("Priya");
22        roster.add("Maya");
23        roster.add("Sam");
24        String wanted = new String("Maya");
25        System.out.println("Maya: " + findName(roster, wanted));
26    }
27 }
```

Maya: -1

"Maya" is in the roster, yet the search says it is not. Recall from Topic 2.6 that for objects, `==` asks whether two references point to the **same object**, not whether the objects hold the same characters. `new String("Maya")` builds a brand-new `String` object with the same letters, so it is never the same object as the one in the list. Students make this mistake because `==` works perfectly for the `int` search, and it sometimes seems to work for strings too, which makes it worse: a search that works only some of the time is very hard to debug. Changing the comparison to `names.get(i).equals(target)` makes the same program print `Maya: 2`.

PAUSE AND TRY**Question 7** Debugging

This search is supposed to return the index of a city in an `ArrayList`, or `-1` if the city is not there. It prints `1` for `"Dallas"`, then crashes with an `IndexOutOfBoundsException` when it searches for `"Denver"`. Find the error and correct it.

PauseSizeBug.java

```

1  import java.util.ArrayList;
2
3  public class PauseSizeBug
4  {
5      public static int findCity(ArrayList<String> cities, String target)
6      {
7          for (int i = 0; i <= cities.size(); i++)
8          {
9              if (cities.get(i).equals(target))
10             {
11                 return i;
12             }
13         }
14         return -1;
15     }
16
17     public static void main(String[] args)
18     {
19         ArrayList<String> stops = new ArrayList<String>();
20         stops.add("Austin");
21         stops.add("Dallas");
22         stops.add("Tulsa");
23         System.out.println(findCity(stops, "Dallas"));
24         System.out.println(findCity(stops, "Denver"));
25     }
26 }

```

Question 8 Short answer

A student wants a method that returns the **index** of a name in an `ArrayList<String>`, and plans to write it with an enhanced `for` loop like `hasName`. Explain why this plan does not work well, and what kind of loop to use instead.

4.14.5 Searching from the Other End

A visitor log adds each new entry to the end of an `ArrayList`. If you want to know when Ana **last** visited, searching from the front finds her **first** visit instead. You could check every entry and remember the last match, but there is a cleaner way: start at the end and walk toward the front. The first match you meet is then the last one in the list.

A linear search can begin at either end. Only three things change when it runs backward: the loop starts at the last valid index, `length - 1` (or `size() - 1`); it continues while the index is `>= 0`; and it counts down with `i--`.

WORKED EXAMPLE Searching backward through scores

Compare the loop header with `findIndex`. The `if`, the `return i`, the `return -1`, and the counter are unchanged.

SearchBackward.java

```

1 public class SearchBackward
2 {
3     private static int comparisons = 0;
4
5     public static int findLastIndex(int[] values, int target)
6     {
7         comparisons = 0;
8         for (int i = values.length - 1; i >= 0; i--)
9         {
10             comparisons++;
11             if (values[i] == target)
12             {
13                 return i;
14             }
15         }
16         return -1;
17     }
18
19     public static void main(String[] args)
20     {
21         int[] scores = {72, 85, 90, 64, 85};
22         int[] targets = {85, 72, 70};
23         for (int target : targets)
24         {
25             int result = findLastIndex(scores, target);
26             System.out.println(target + ": index " + result + ", " + comparisons +
27 " comparisons");
28         }
29     }

```

```

85: index 4, 1 comparisons
72: index 0, 5 comparisons
70: index -1, 5 comparisons

```

Target	Elements compared (from the end)	Result	Comparisons
85	85	index 4	1
72	85, 64, 90, 85, 72	index 0	5
70	all 5 elements	-1	5

The same array gives very different counts from the forward search in 4.14.3. Searching forward found 72 in 1 comparison; backward it takes 5. Searching forward for 85 returned index 1; backward it returns index 4, the other copy. A missing target still costs 5 comparisons either way, because every element must be checked.

The same pattern works on an `ArrayList`, starting at `size() - 1`:

LatestVisit.java

```
1 import java.util.ArrayList;
2
3 public class LatestVisit
4 {
5     public static int findLatest(ArrayList<String> visitors, String target)
6     {
7         for (int i = visitors.size() - 1; i >= 0; i--)
8         {
9             if (visitors.get(i).equals(target))
10            {
11                return i;
12            }
13        }
14        return -1;
15    }
16
17    public static void main(String[] args)
18    {
19        ArrayList<String> log = new ArrayList<String>();
20        log.add("Ana");
21        log.add("Ben");
22        log.add("Ana");
23        log.add("Cole");
24        log.add("Ben");
25        log.add("Dee");
26        System.out.println("Ana last visited at entry " + findLatest(log, "Ana"));
27        System.out.println("Ben last visited at entry " + findLatest(log, "Ben"));
28    }
29 }
```

```
Ana last visited at entry 2
Ben last visited at entry 4
```

The log is Ana, Ben, Ana, Cole, Ben, Dee (indexes 0 to 5). Searching for "Ana" from the end checks Dee, Ben, Cole, and then Ana at index 2, and stops there, so the earlier Ana at index 0 is never reached. Searching for "Ben" stops right away at index 4.

KEY IDEA

A linear search can start at either end. Searching from the last index down to 0 finds the last occurrence of the target, and its comparison counts are measured from the end.

WATCH OUT

Starting at length instead of length - 1. The last valid index is one less than the length, but it is tempting to start the loop at the length itself:

BackwardStartBug.java

```
1 public class BackwardStartBug
2 {
3     public static int findLastIndex(int[] values, int target)
4     {
5         for (int i = values.length; i >= 0; i--)
6         {
7             if (values[i] == target)
8             {
9                 return i;
10            }
11        }
12        return -1;
13    }
14
15    public static void main(String[] args)
16    {
17        int[] scores = {72, 85, 90, 64, 85};
18        System.out.println("Searching...");
19        System.out.println(findLastIndex(scores, 85));
20    }
21 }
```

Searching...

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 5 out of bounds for length 5

On the very first pass `i` is 5, and `values[5]` does not exist, so the program crashes before checking anything, whether or not the target is in the array. (The exact message may vary with your Java version.) This is the same off-by-one idea from Topic 4.3, at the other end of the array. Start at `values.length - 1`.

PAUSE AND TRY

Question 9 Tracing

What does this program print?

PauseBackward.java

```

1 public class PauseBackward
2 {
3     private static int comparisons = 0;
4
5     public static int findLastIndex(int[] values, int target)
6     {
7         comparisons = 0;
8         for (int i = values.length - 1; i >= 0; i--)
9         {
10             comparisons++;
11             if (values[i] == target)
12             {
13                 return i;
14             }
15         }
16         return -1;
17     }
18
19     public static void main(String[] args)
20     {
21         int[] nums = {3, 7, 3, 9, 7, 1};
22         int a = findLastIndex(nums, 7);
23         System.out.println("7: " + a + " after " + comparisons);
24         int b = findLastIndex(nums, 3);
25         System.out.println("3: " + b + " after " + comparisons);
26     }
27 }

```

4.14.6 Searching a 2D Array

A school stores locker numbers in a 2D array, one row per hallway. Given a locker number, the office wants its hallway (row) and position (column). Every search so far has walked down a single line of elements. A 2D array is not a single line, so where do you start?

Recall from Topic 4.11 that a 2D array is an array of rows, and `grid[r]` is row `r` as an ordinary 1D array. That makes the plan simple: take the rows one at a time, and run a linear search on each row. If a row contains the target, stop and report where. If no row does, the target is not in the 2D array. You do not need a new algorithm, just the 1D search applied to each row in turn.

WORKED EXAMPLE Finding a locker

`locate` loops over the row indexes and hands each row, `grid[row]`, to the same `findIndex` method you have used all chapter. Look at what `locate` does with the column `findIndex` returns.

SearchGrid.java

```

1 public class SearchGrid
2 {
3     private static int comparisons = 0;
4

```

```

5     public static int findIndex(int[] values, int target)
6     {
7         for (int i = 0; i < values.length; i++)
8         {
9             comparisons++;
10            if (values[i] == target)
11            {
12                return i;
13            }
14        }
15        return -1;
16    }
17
18    public static String locate(int[][] grid, int target)
19    {
20        comparisons = 0;
21        for (int row = 0; row < grid.length; row++)
22        {
23            int col = findIndex(grid[row], target);
24            if (col != -1)
25            {
26                return "row " + row + ", column " + col;
27            }
28        }
29        return "not found";
30    }
31
32    public static void main(String[] args)
33    {
34        int[][] lockers = { {12, 40, 7, 33},
35                           {25, 18, 61, 9},
36                           {50, 3, 44, 27} };
37        System.out.println("61: " + locate(lockers, 61) + " (" + comparisons + "
comparisons)");
38        System.out.println("12: " + locate(lockers, 12) + " (" + comparisons + "
comparisons)");
39        System.out.println("80: " + locate(lockers, 80) + " (" + comparisons + "
comparisons)");
40    }
41 }

```

```

61: row 1, column 2 (7 comparisons)
12: row 0, column 0 (1 comparisons)
80: not found (12 comparisons)

```

Trace the search for 61:

row	Row searched	findIndex compares	Returns	comparisons after
0	{12, 40, 7, 33}	12, 40, 7, 33	-1	4
1	{25, 18, 61, 9}	25, 18, 61	2	7

Row 0 has no 61, so `findIndex` checks all four elements and returns `-1`, and `locate` moves to row 1. In row 1, `findIndex` finds 61 at index 2, so `col` is 2, the test `col != -1` is true, and `locate` returns "row 1, column 2". Row 2 is never searched.

Notice where the counter is reset: in `locate`, once per search of the whole grid, not in `findIndex`. That lets the count keep adding up across rows: 4 for row 0 plus 3 for row 1 is 7. A missing target such as 80 costs one comparison for every element in the grid, 3 rows $\times$ 4 columns = 12.

The same row-by-row idea can be written with nested loops instead of a helper method. When you only need to know **whether** the target is present, nested enhanced `for` loops (Topic 4.12) work: the outer variable is a row (a 1D array), and the inner loop searches that row.

SeatChart.java

```

1 public class SeatChart
2 {
3     public static boolean isSeated(String[][] chart, String name)
4     {
5         for (String[] row : chart)
6         {
7             for (String student : row)
8             {
9                 if (student.equals(name))
10                {
11                    return true;
12                }
13            }
14        }
15        return false;
16    }
17
18    public static void main(String[] args)
19    {
20        String[][] chart = { {"Ava", "Raj", "Lin"},
21                             {"Kai", "Zoe", "Eli"} };
22        System.out.println("Zoe seated? " + isSeated(chart, "Zoe"));
23        System.out.println("Max seated? " + isSeated(chart, "Max"));
24    }
25 }

```

```

Zoe seated? true
Max seated? false

```

The outer loop's `row` holds `{"Ava", "Raj", "Lin"}` first; the inner loop compares each name with the target using `equals`. No match, so the outer loop moves on to `{"Kai", "Zoe", "Eli"}`, where "Zoe" matches and `return true`; leaves both loops at once. For "Max", both rows are searched completely, and the method returns `false` after the loops.

Because the search finishes row 0 before starting row 1, it visits elements in row-major order (Topic 4.12). If the target appears in two places, it reports the one in the earlier row, even if the other one has a smaller column.

KEY IDEA

To search a 2D array, access each row and run a linear search on that row. Stop as soon as a row contains the target; report "not found" only after every row has been searched.

WATCH OUT

Giving up after the first row. This is the 2D version of the mistake from 4.14.1. The only difference from `SearchGrid` is the extra `return "not found";` inside the outer loop:

`GridReturnBug.java`

```
1 public class GridReturnBug
2 {
3     public static int findIndex(int[] values, int target)
4     {
5         for (int i = 0; i < values.length; i++)
6         {
7             if (values[i] == target)
8             {
9                 return i;
10            }
11        }
12        return -1;
13    }
14
15    public static String locate(int[][] grid, int target)
16    {
17        for (int row = 0; row < grid.length; row++)
18        {
19            int col = findIndex(grid[row], target);
20            if (col != -1)
21            {
22                return "row " + row + ", column " + col;
23            }
24            return "not found";
25        }
26        return "not found";
27    }
28
29    public static void main(String[] args)
30    {
31        int[][] lockers = { {12, 40, 7, 33},
32                           {25, 18, 61, 9},
33                           {50, 3, 44, 27} };
34        System.out.println("61: " + locate(lockers, 61));
35    }
36 }
```

61: not found

Row 0 does not contain 61, so `col` is `-1`, the `if` is skipped, and the next statement is `return "not found";`, still inside the outer loop. The method ends after searching only row 0.

Students put it there because "if this row did not have it, say not found" sounds reasonable, but one row without the target says nothing about the other rows. Remove the `return` inside the loop and keep only the one after the loop, as in `SearchGrid`, which correctly reports row 1, column 2.

PAUSE AND TRY

Question 10 Tracing

What does this program print?

PauseGrid.java

```
1 public class PauseGrid
2 {
3     private static int comparisons = 0;
4
5     public static int findIndex(int[] values, int target)
6     {
7         for (int i = 0; i < values.length; i++)
8         {
9             comparisons++;
10            if (values[i] == target)
11            {
12                return i;
13            }
14        }
15        return -1;
16    }
17
18    public static String locate(int[][] grid, int target)
19    {
20        comparisons = 0;
21        for (int row = 0; row < grid.length; row++)
22        {
23            int col = findIndex(grid[row], target);
24            if (col != -1)
25            {
26                return "row " + row + ", column " + col;
27            }
28        }
29        return "not found";
30    }
31
32    public static void main(String[] args)
33    {
34        int[][] board = { {6, 1, 8},
35                           {7, 5, 3},
36                           {2, 9, 4},
37                           {5, 0, 5} };
38        System.out.println(locate(board, 9) + " after " + comparisons);
39        System.out.println(locate(board, 5) + " after " + comparisons);
40    }
```

```
41 }
```

Question 11 Multiple choice

A 2D array has 4 rows and 5 columns. Its only copy of the target is at row 2, column 3. How many comparisons does a row-by-row linear search like `locate` make before it finds the target?

- A. 14
- B. 6
- C. 13
- D. 20

Putting it together

A music app keeps a listening history as an array of `Song` objects, oldest first. It needs two answers about an artist: the first song by that artist in the history, and the most recent one. This program combines a search on objects, `equals`, searching from both ends, and a comparison counter. Java allows only one `public` class per file, so the helper class here leaves off `public`; in a real project each class gets its own file.

WORKED EXAMPLE Searching a listening history

The searches compare `songs[i].getArtist()` with the target, so they search by one attribute of each object and return the index of the whole `Song`. Predict all three lines of output before reading on.

PlaylistSearch.java

```
1 class Song
2 {
3     private String title;
4     private String artist;
5
6     public Song(String title, String artist)
7     {
8         this.title = title;
9         this.artist = artist;
10    }
11
12    public String getTitle()
13    {
14        return title;
15    }
16
17    public String getArtist()
18    {
19        return artist;
20    }
21 }
```

```
22
23 public class PlaylistSearch
24 {
25     private static int comparisons = 0;
26
27     public static int firstBy(Song[] songs, String artist)
28     {
29         comparisons = 0;
30         for (int i = 0; i < songs.length; i++)
31         {
32             comparisons++;
33             if (songs[i].getArtist().equals(artist))
34             {
35                 return i;
36             }
37         }
38         return -1;
39     }
40
41     public static int latestBy(Song[] songs, String artist)
42     {
43         comparisons = 0;
44         for (int i = songs.length - 1; i >= 0; i--)
45         {
46             comparisons++;
47             if (songs[i].getArtist().equals(artist))
48             {
49                 return i;
50             }
51         }
52         return -1;
53     }
54
55     public static void main(String[] args)
56     {
57         Song[] history = { new Song("Sunrise", "Nova"),
58                             new Song("Static", "Echo Park"),
59                             new Song("Glow", "Nova"),
60                             new Song("Drift", "Lumen"),
61                             new Song("Tides", "Echo Park"),
62                             new Song("Ember", "Lumen") };
63
64         int first = firstBy(history, "Echo Park");
65         System.out.println("First Echo Park song: " + history[first].getTitle()
66                             + " (" + comparisons + " comparisons)");
67
68         int latest = latestBy(history, "Echo Park");
69         System.out.println("Latest Echo Park song: " + history[latest].getTitle()
70                             + " (" + comparisons + " comparisons)");
71
72         int missing = latestBy(history, "Orbit");
73         System.out.println("Orbit: index " + missing + " (" + comparisons + "
74                             comparisons)");
75     }
76 }
```

75 }

First Echo Park song: Static (2 comparisons)
 Latest Echo Park song: Tides (2 comparisons)
 Orbit: index -1 (6 comparisons)

The history, by index:

Index	Title	Artist
0	Sunrise	Nova
1	Static	Echo Park
2	Glow	Nova
3	Drift	Lumen
4	Tides	Echo Park
5	Ember	Lumen

- `firstBy(history, "Echo Park")` compares the artist at index 0 (Nova, no) and index 1 (Echo Park, yes). It returns 1 after 2 comparisons, and `main` uses `history[1].getTitle()` to print Static.
- `latestBy(history, "Echo Park")` starts at index 5: Lumen, no; then index 4: Echo Park, yes. It returns 4 after 2 comparisons, so `main` prints Tides. The two searches found different songs because they started from different ends.
- `latestBy(history, "Orbit")` finds no match, compares all 6 artists, and returns -1. `main` prints the index instead of calling `history[missing].getTitle()`, because `history[-1]` would throw an `ArrayIndexOutOfBoundsException`. Whenever a search can return -1, check the result before using it as an index.

Every comparison uses `equals`, because the artist names are `String` objects. With `==` the search could fail to find an artist that is really there, exactly as in 4.14.4.

Vocabulary

Term	Definition	Example
linear search	An algorithm that checks each element in order until it finds the target or has checked every element	<code>findIndex</code> in 4.14.1
sequential search	Another name for linear search	Searching a roster from the first name to the last
target	The value a search is looking for	In <code>findIndex(scores, 90)</code> , the target is 90
sentinel value	A special value that can never be a real answer, used to signal a situation such as "not found"	A search returns -1 when the target is missing

comparisons	The number of times a search compares an element with the target; a measure of how much work the search did	Finding the target at index 2 takes 3 comparisons
counter variable	A variable that starts at 0 and goes up by 1 each time something happens, used to count it	<code>comparisons++</code> ; right before the <code>if</code>
short-circuit evaluation	In <code>a && b</code> , Java skips <code>b</code> when <code>a</code> is false	<code>index < scores.length && scores[index] != target</code>

Chapter summary

- A linear (sequential) search checks the elements of an array or `ArrayList` one at a time, in order, until it finds the target or has checked them all. It returns the index of the match, or `-1` when the target is missing, because `-1` can never be a valid index.
- "Not found" can only be decided after the loop has checked every element, so `return -1;` goes after the loop. The search stops at the first match, which decides which copy it reports when the target appears more than once.
- A target found at index `k` costs `k + 1` comparisons; a missing target costs one comparison per element, the most a linear search can make. A counter variable placed right before the comparison verifies these counts.
- Searching an `ArrayList` uses `size()` and `get(i)`, and strings and other objects are compared with `equals`, never `==`. A search can also start at the last index and count down, which finds the last occurrence.
- A 2D array is searched by taking each row in turn and running a linear search on that row, stopping as soon as a row contains the target.

Review questions

Question 12 Multiple choice

Both methods below are linear searches; `searchForward` starts at index 0 and `searchBackward` starts at the last index. Each counts its comparisons.

```
1 int[] data = {6, 2, 9, 2, 4, 9, 1};
```

Which statement about searching `data` for `9` is true?

- Forward returns index 2 after 3 comparisons; backward returns index 5 after 2 comparisons.
- Both return index 2, because a linear search always reports the first copy.
- Forward returns index 2 after 2 comparisons; backward returns index 5 after 1 comparison.
- Forward returns index 5 after 6 comparisons; backward returns index 2 after 5 comparisons.

Question 13 Tracing

This search visits a 2D array in **column-major** order (Topic 4.12) instead of row by row. What does it print?

ReviewColumnMajor.java

```
1 public class ReviewColumnMajor
2 {
3     private static int comparisons = 0;
4
5     public static String locateByColumn(int[][] grid, int target)
6     {
7         comparisons = 0;
8         for (int col = 0; col < grid[0].length; col++)
9         {
10             for (int row = 0; row < grid.length; row++)
11             {
12                 comparisons++;
13                 if (grid[row][col] == target)
14                 {
15                     return "row " + row + ", column " + col;
16                 }
17             }
18         }
19         return "not found";
20     }
21
22     public static void main(String[] args)
23     {
24         int[][] grid = { {4, 8, 1},
25                          {7, 2, 9},
26                          {3, 6, 5} };
27         System.out.println(locateByColumn(grid, 6));
28         System.out.println("comparisons: " + comparisons);
29     }
30 }
```

Question 14 Debugging

`contains` should report whether a value appears anywhere in a 2D array. For this 2-row, 4-column array it prints `true` for 12 but `false` for 11, even though 11 is in the array. Find the error and correct it.

ReviewColumnsBug.java

```
1 public class ReviewColumnsBug
2 {
3     public static boolean contains(int[][] grid, int target)
4     {
5         for (int row = 0; row < grid.length; row++)
6         {
7             for (int col = 0; col < grid.length; col++)
8             {
9                 if (grid[row][col] == target)
10                {
11                    return true;
12                }
13            }
14        }
15        return false;
16    }
17
18    public static void main(String[] args)
19    {
20        int[][] prices = { {5, 12, 8, 20},
21                           {15, 3, 11, 7} };
22        System.out.println(contains(prices, 12));
23        System.out.println(contains(prices, 11));
24    }
25 }
```

Question 15 Multiple choice

This method is a linear search with one change in its comparison. What do the two calls print?

ReviewFirstAbove.java

```
1 public class ReviewFirstAbove
2 {
3     public static int mystery(int[] values, int target)
4     {
5         for (int i = 0; i < values.length; i++)
6         {
7             if (values[i] > target)
8             {
9                 return i;
10            }
11        }
12        return -1;
13    }
14
15    public static void main(String[] args)
16    {
17        int[] scores = {62, 78, 91, 55, 88};
18        System.out.println(mystery(scores, 80));
19        System.out.println(mystery(scores, 95));
20    }
21 }
```

A. 2 then -1

- B. 91 then -1
- C. 4 then -1
- D. 2 then 4

Question 16 Multiple choice

`findItem` is meant to be a linear search of an `ArrayList<String>`. The list is `["milk", "eggs", "bread", "rice"]`. For which target does it return the wrong answer?

`ReviewSkipLast.java`

```
1 import java.util.ArrayList;
2
3 public class ReviewSkipLast
4 {
5     public static int findItem(ArrayList<String> items, String target)
6     {
7         for (int i = 0; i < items.size() - 1; i++)
8         {
9             if (items.get(i).equals(target))
10            {
11                return i;
12            }
13        }
14        return -1;
15    }
16
17    public static void main(String[] args)
18    {
19        ArrayList<String> cart = new ArrayList<String>();
20        cart.add("milk");
21        cart.add("eggs");
22        cart.add("bread");
23        cart.add("rice");
24        System.out.println(findItem(cart, "milk"));
25        System.out.println(findItem(cart, "bread"));
26        System.out.println(findItem(cart, "rice"));
27        System.out.println(findItem(cart, "jam"));
28    }
29 }
```

- A. "rice"
- B. "milk"
- C. "bread"
- D. "jam"

Question 17 Short answer

An `ArrayList` holds 50 different names. The name being searched for is at index 10. How many comparisons does a linear search make if it starts at the front, and how many if it starts at the back? Then explain why neither direction is better in general.

Reflection

Check each statement. If the answer is "not yet," reread the linked subsection.

- ☐ I can trace a linear search and give the index it returns, or -1 when the target is missing. Reread: 4.14.1 Checking Each Element in Order
- ☐ I can explain why a search stops at the first match and spot a search that gives up too early or never stops. Reread: 4.14.2 Stopping at the First Match
- ☐ I can count the comparisons a linear search makes and say when it makes the fewest and the most. Reread: 4.14.3 Counting Comparisons
- ☐ I can search an ArrayList of strings with size, get, and equals. Reread: 4.14.4 Searching an ArrayList of Strings
- ☐ I can trace a search that starts at the last element and explain which copy of the target it finds. Reread: 4.14.5 Searching from the Other End
- ☐ I can search a 2D array by running a linear search on each row in turn. Reread: 4.14.6 Searching a 2D Array

Check Your Reasoning

Answers and explanations for every question, numbered to match.

Question 1

A. The search compares 38 (index 0), then 45 (index 1), then 41 (index 2). That comparison is true, so the method returns 2 immediately.

- **B** is wrong: the method returns `i`, the **index** where the match was found, not `values[i]`, the value itself. You already know the value; you asked for its position.
- **C** is wrong: 41 also appears at index 4, but the search never reaches it. It returns the first time the comparison is true.
- **D** is wrong: it counts positions starting from 1. Array indexes start at 0, so the third element is at index 2.

Reread 4.14.1 *Checking Each Element in Order* if you chose differently.

Question 2

Model answer: 0 is a real index: it is the position of the first element. If `findIndex` returned 0 for "not found," a program could not tell the difference between "the target is the first element" and "the target is not in the array." -1 works because no array has an element at index -1, so it can only mean "not found."

A complete answer must:

- say that 0 is a valid index (the first element),
- explain that the two meanings would be impossible to tell apart, and
- say why -1 does not have this problem.

Reread 4.14.1 *Checking Each Element in Order* if your answer differed.

Question 3

```
index: 1
index: 5
```

First search (target 9): at index 0, `codes[0]` is 4, which is not 9, so the loop continues. At index 1, `codes[1]` is 9, so `codes[index] !== target` is false and the loop stops. `index` is 1. The second 9 at index 3 is never reached.

Second search (target 6): no element is 6, so the loop moves `index` through 0, 1, 2, 3, 4 and on to 5. At the check with `index` equal to 5, `5 < 5` is false, so the loop stops without ever evaluating `codes[5]`. `index` is 5, the length of the array, which means "not found."

Reread 4.14.2 *Stopping at the First Match* if your output differed.

Question 4

A. The loop now keeps going only while nothing has been found. As soon as `foundAt` becomes 1, the next check of `foundAt == -1` is false and the loop ends, so the second 85 is never reached. If the target is missing, `foundAt` stays -1 and the loop stops normally at the end of the array. With this header the program prints `Index of 85: 1`, and a search for 70 returns -1.

- **B** is wrong: `foundAt` starts at `-1`, so `foundAt != -1` is false on the very first check. The loop body never runs and the method returns `-1` even though `85` is in the array.
- **C** is wrong: starting at index `1` skips the first element, and the loop still does not stop, so it still returns `4` for `85`.
- **D** is wrong: it still does not stop at the first match, and when `i` reaches `values.length` it reads past the end and throws an `ArrayIndexOutOfBoundsException`.

Reread 4.14.2 *Stopping at the First Match* if you chose differently.

Question 5

A. `13` is the sixth element, at index `5`. The search compares `15`, `17`, `14`, `16`, `18`, and `13`, which is `6` comparisons, matching the rule `k + 1` with `k` equal to `5`.

- **B** is wrong: it forgets that the comparison that finds the target also counts. Indexes `0` through `5` are six elements.
- **C** is wrong: it counts positions starting from `1` for the index. The index is `5`; the count of `6` is correct.
- **D** is wrong: `8` would be the count only if the search kept going to the end. It stops as soon as it finds `13`.

Reread 4.14.3 *Counting Comparisons* if you chose differently.

Question 6

Model answer: The fewest is `1` comparison, when the target is the first element checked (index `0` for a search that starts at the front). The most is `200` comparisons. That happens in two situations: when the target is the last element checked (index `199`), and when the target is not in the array at all, because the search must compare every element before it can return `-1`.

A complete answer must:

- give `1` as the fewest, when the target is the first element checked,
- give `200` as the most, and
- name both worst-case situations: target at the last position, and target missing.

Reread 4.14.3 *Counting Comparisons* if your answer differed.

Question 7

The loop runs one index too far. The condition `i <= cities.size()` lets `i` reach `3` in a list of size `3`, but the valid indexes are only `0`, `1`, and `2` (Topic 4.8). The search for `"Dallas"` returns at index `1` before reaching the end, so it works. The search for `"Denver"` finds no match, so `i` keeps going until `cities.get(3)` is called, and `ArrayList` throws an `IndexOutOfBoundsException` (Topic 4.9).

Corrected line: change `<=` to `<`.

PauseSizeFixed.java

```
1 import java.util.ArrayList;
2
3 public class PauseSizeFixed
4 {
5     public static int findCity(ArrayList<String> cities, String target)
6     {
7         for (int i = 0; i < cities.size(); i++)
8         {
9             if (cities.get(i).equals(target))
10            {
11                return i;
12            }
13        }
14        return -1;
15    }
16
17    public static void main(String[] args)
18    {
19        ArrayList<String> stops = new ArrayList<String>();
20        stops.add("Austin");
21        stops.add("Dallas");
22        stops.add("Tulsa");
23        System.out.println(findCity(stops, "Dallas"));
24        System.out.println(findCity(stops, "Denver"));
25    }
26 }
```

Now the loop stops after index 2, and the program prints 1 and then -1. Notice that the bug only showed up when the target was missing, because that is the only time a linear search reaches the end of the list.

Reread 4.14.4 *Searching an ArrayList of Strings* if you missed it.

Question 8

Model answer: An enhanced `for` loop gives you each element but not its index; the loop variable holds a copy of the element and nothing tells you its position. A search that must return an index needs to know `i` at the moment of the match, so it should use an indexed `for` loop (or a `while` loop with an index variable) that runs while `i < names.size()` and uses `names.get(i)`.

A complete answer must:

- say the enhanced `for` loop has no index variable,
- explain that the method needs the index of the match to return it, and
- name an indexed `for` loop or `while` loop as the fix.

Reread 4.14.4 *Searching an ArrayList of Strings* if your answer differed.

Question 9

```
7: 4 after 2
3: 2 after 4
```

The array is 3, 7, 3, 9, 7, 1 (indexes 0 to 5), and the search starts at index 5.

Search for	Elements compared, from index 5 down	Stops at	Comparisons
7	1 (index 5), 7 (index 4)	index 4	2
3	1, 7, 9, 3 (index 2)	index 2	4

Both targets appear twice, and both times the backward search reports the copy with the **larger** index, because it reaches that copy first. `comparisons` is reset to 0 at the start of each call, so the second count is 4, not 6.

Reread 4.14.5 *Searching from the Other End* if your output differed.

Question 10

```
row 2, column 1 after 8
row 1, column 1 after 5
```

Search for 9: row 0 {6, 1, 8} has no 9, so `findIndex` compares all 3 and returns -1. Row 1 {7, 5, 3} has no 9: 3 more comparisons, 6 in total. Row 2 {2, 9, 4}: 2 is not 9, then 9 matches at index 1. That is 2 more comparisons, 8 in total. `locate` returns "row 2, column 1".

Search for 5: 5 appears in row 1 and twice in row 3. Row 0 costs 3 comparisons with no match. In row 1, 7 is not 5, then 5 matches at index 1: 2 more, 5 in total. The search stops there, so the 5s in row 3 are never reached. Rows are searched in order, so the earliest row wins.

Reread 4.14.6 *Searching a 2D Array* if your output differed.

Question 11

A. Rows 0 and 1 are searched completely and do not contain the target: $5 + 5 = 10$ comparisons. In row 2, the search compares columns 0, 1, 2, and 3, and column 3 matches: 4 more. The total is $10 + 4 = 14$.

- **B** is wrong: $2 + 3 + 1 = 6$ adds the row and column numbers instead of counting the elements in the rows searched before it.
- **C** is wrong: it forgets that the comparison that finds the target also counts. Columns 0 through 3 are four comparisons, not three.
- **D** is wrong: $4 \times 5 = 20$ is the count for a target that is missing or in the very last position. This search stops in row 2.

Reread 4.14.6 *Searching a 2D Array* and the counting rule in 4.14.3 if you chose differently.

Question 12

A. Forward compares 6, 2, 9: the match is at index 2, after 3 comparisons. Backward starts at index 6 and compares 1, then 9 at index 5: 2 comparisons. The `ReviewBothEnds` program prints forward: index 2, 3 comparisons and backward: index 5, 2 comparisons.

`ReviewBothEnds.java`

```
1 public class ReviewBothEnds
2 {
3     private static int comparisons = 0;
4
5     public static int searchForward(int[] values, int target)
6     {
7         comparisons = 0;
```

```

8      for (int i = 0; i < values.length; i++)
9      {
10         comparisons++;
11         if (values[i] == target)
12         {
13             return i;
14         }
15     }
16     return -1;
17 }
18
19 public static int searchBackward(int[] values, int target)
20 {
21     comparisons = 0;
22     for (int i = values.length - 1; i >= 0; i--)
23     {
24         comparisons++;
25         if (values[i] == target)
26         {
27             return i;
28         }
29     }
30     return -1;
31 }
32
33 public static void main(String[] args)
34 {
35     int[] data = {6, 2, 9, 2, 4, 9, 1};
36     int f = searchForward(data, 9);
37     System.out.println("forward: index " + f + ", " + comparisons + "
comparisons");
38     int b = searchBackward(data, 9);
39     System.out.println("backward: index " + b + ", " + comparisons + "
comparisons");
40 }
41 }

```

- **B** is wrong: a search reports the first copy **it reaches**. Starting from the end, that is the copy at index 5.
- **C** is wrong: each count leaves out the comparison that finds the target. Index 2 from the front is the third element; index 5 from the back is the second element checked.
- **D** is wrong: it swaps the directions. A forward search stops at the first 9 it meets, at index 2; it never gets to index 5.

Reread 4.14.5 *Searching from the Other End* and 4.14.3 *Counting Comparisons* if you chose differently.

Question 13

```

row 2, column 1
comparisons: 6

```

The outer loop picks a column and the inner loop goes down that column, so the elements are compared in this order:

Comparison	col	row	grid[row][col]	Equals 6?
1	0	0	4	no
2	0	1	7	no
3	0	2	3	no
4	1	0	8	no
5	1	1	2	no
6	1	2	6	yes

Column 0 is searched completely (3 comparisons), then column 1 down to row 2. A row-by-row search would have taken 8 comparisons for the same target (3 for row 0, 3 for row 1, 2 into row 2). The order of the search changes the count, but either order finds the target if it is there.

Reread 4.14.6 *Searching a 2D Array* if your output differed.

Question 14

The inner loop uses the number of rows as its limit. `grid.length` is the number of rows, 2 (Topic 4.11). So the inner loop only checks columns 0 and 1 of each row. 12 is at row 0, column 1, so it is found. 11 is at row 1, column 2, which is never checked, so the method wrongly returns `false`. No exception happens, because columns 0 and 1 always exist; the bug shows only as a wrong answer.

Corrected line: the inner loop must run over the length of the row being searched.

ReviewColumnsFixed.java

```

1 public class ReviewColumnsFixed
2 {
3     public static boolean contains(int[][] grid, int target)
4     {
5         for (int row = 0; row < grid.length; row++)
6         {
7             for (int col = 0; col < grid[row].length; col++)
8             {
9                 if (grid[row][col] == target)
10                {
11                    return true;
12                }
13            }
14        }
15        return false;
16    }
17
18    public static void main(String[] args)
19    {
20        int[][] prices = { {5, 12, 8, 20},
21                           {15, 3, 11, 7} };
22        System.out.println(contains(prices, 12));
23        System.out.println(contains(prices, 11));
24    }
25 }

```

`grid[row].length` is 4, so every column of each row is searched, and the program prints `true` twice. (`grid[0].length` would also work for a rectangular array.)

Reread 4.14.6 *Searching a 2D Array* if you missed it.

Question 15

A. The method still checks elements in order and returns at the first `true` comparison, but the comparison is now `values[i] > target`. So it returns the index of the **first element greater than the target**. For 80: 62 no, 78 no, 91 yes, so it returns 2. For 95: no score is greater than 95, so the loop finishes and returns -1.

- **B** is wrong: the method returns the index `i`, not the element `values[i]`.
- **C** is wrong: 88 at index 4 is also greater than 80, but the search stops at the first match, index 2.
- **D** is wrong: for 95, no comparison is ever true, so `return i;` never runs and the method returns -1 after the loop.

Reread 4.14.2 *Stopping at the First Match* if you chose differently.

Question 16

A. The condition `i < items.size() - 1` stops the loop at `i` equal to 2, so the last element, at index 3, is never compared. "rice" is at index 3, so the method returns -1 even though it is in the list. The program prints 0, 2, -1, -1; only the third line is wrong.

- **B** is wrong: "milk" is at index 0, which the loop does check, so it correctly returns 0.
- **C** is wrong: "bread" is at index 2, the last index the loop checks, so it correctly returns 2.
- **D** is wrong: "jam" is not in the list, so -1 is the right answer, even though the method never looked at index 3.

The loop limit for a full search is `i < items.size()`. The - 1 belongs only in loops that compare an element with the **next** one, as in Topic 4.5's consecutive-pairs algorithm.

Reread 4.14.4 *Searching an ArrayList of Strings* if you chose differently.

Question 17

Model answer: From the front, the search compares indexes 0 through 10, which is 11 comparisons. From the back, it compares indexes 49 down to 10, which is $49 - 10 + 1 = 40$ comparisons. For this target the front is much faster, but a name near the end would favor the back. When you do not know in advance where the target is, neither direction wins: both make between 1 and 50 comparisons, and both make all 50 when the name is missing.

A complete answer must:

- give 11 from the front and 40 from the back,
- explain that the faster direction depends on where the target happens to be, and
- note that both directions make every comparison when the target is missing.

Reread 4.14.3 *Counting Comparisons* and 4.14.5 *Searching from the Other End* if your answer differed.