

4.5 Implementing Array Algorithms

Self-Study Guide · About 70 minutes · With answer key

By the end you will be able to

- Trace array traversals that find a sum, an average, a minimum, or a maximum, including the index where the minimum or maximum is.
- Count the elements of an array that have a property, and decide whether at least one element or every element has it.
- Visit every pair of neighboring elements without going past the end of the array.
- Use a loop inside a loop to decide whether an array contains a repeated value.
- Predict the result of shifting, rotating, or reversing the elements of an array, and explain why the order of the copies matters.
- Find and fix the common bugs in these algorithms, such as mixing up an index with an element.

What to remember from earlier

- An accumulator starts at 0 before a loop and collects a sum or a count inside it. ([Topic 2.9: Implementing Selection and Iteration Algorithms](#))
- A minimum or maximum keeps a best-so-far value that starts at the first value. ([Topic 2.9: Implementing Selection and Iteration Algorithms, 2.9.6 Finding the Minimum or Maximum](#))
- A boolean flag is switched inside a loop when something you are watching for happens. ([Topic 2.10: Implementing String Algorithms, 2.10.2 Checking a Property with a Flag](#))
- An indexed for loop visits every index of an array. ([Topic 4.4: Array Traversals, 4.4.1 Visiting Every Element with a for Loop](#))
- An enhanced for loop hands you a copy of each element without its index. ([Topic 4.4: Array Traversals, 4.4.4 The Enhanced for Loop](#))

4.5.1 Sum and Average of an Array

In Topic 2.9 you added up values as a loop produced them: numbers from a loop control variable, or numbers read one at a time from a `Scanner`. Once a value was added, it was gone. An array changes that. All the values sit in the array at once, so a program can add them up, then go back and look at them again, compare neighbors, or move them around. This chapter collects the **standard array algorithms**: well-known traversals that answer the questions programmers ask about a list of values again and again. Each one is a pattern you already know from Topic 2.9 or 2.10, now fed by a traversal from Topic 4.4.

The first two questions are "What is the total?" and "What is the average?" The accumulator from Topic 2.9 still does the work: start a total at 0 before the loop, add one element on each pass, and use the total after the loop. The only new part is where the values come from. Instead of reading them, the loop visits each element of the array.

There is a bonus. In Topic 2.9 you needed a separate counter to know how many values you had added. An array already knows: `values.length` is the number of elements. So the average is the sum divided by the length.

KEY IDEA

To total an array, start an accumulator at 0, add every element to it during a traversal, and use it after the loop. The average is that total divided by the array's `length`, with a `(double)` cast when the total is an `int`.

WORKED EXAMPLE Total and average daily sales

A snack stand records how many items it sold each day for five days. An enhanced for loop (Topic 4.4) is a good fit, because the sum needs every element's value but never its index.

```
1 int[] sales = {12, 30, 8, 25, 16};
2 int total = 0;
3 for (int amount : sales)
4 {
5     total += amount;
6 }
7 System.out.println(total);
8 double average = (double) total / sales.length;
9 System.out.println(average);
```

On each pass, `amount` holds a copy of the next element, and line 5 adds it to `total`.

Pass	amount	total after line 5
1	12	12
2	30	42
3	8	50
4	25	75
5	16	91

Line 7 prints the total. Line 8 casts `total` to `double` first, so Java divides `91.0` by `5`, which is `18.2`. Without the cast, `91 / 5` would be `int` division and give `18`.

```
91
18.2
```

Now change two things: the element type and the kind of loop. When the array holds `double` values, the sum is already a `double`, so no cast is needed. And an indexed for loop (Topic 4.4) does the same job as the enhanced for loop; it reaches each element through its index instead.

WORKED EXAMPLE Average lap time

```
1 double[] lapTimes = {9.5, 10.25, 8.75, 11.0};
2 double sum = 0.0;
3 for (int i = 0; i < lapTimes.length; i++)
4 {
5     sum += lapTimes[i];
6 }
7 System.out.println(sum);
8 System.out.println(sum / lapTimes.length);
```

`i` runs 0, 1, 2, 3, so line 5 adds `lapTimes[0]` through `lapTimes[3]`: $9.5 + 10.25 + 8.75 + 11.0$ is 39.5. The condition `i < lapTimes.length` stops the loop after index 3, the last valid index. Line 8 divides a `double` by an `int`, which keeps the decimal part.

```
39.5
9.875
```

The length of an array can be zero (for example, `int[] none = {};`). For an empty array the total is 0, but there is no average: dividing by a length of 0 makes no sense. As with the counter in Topic 2.9, check that the length is greater than 0 before you divide when an empty array is possible.

WATCH OUT

Starting the loop at index 1. Students who are used to counting from 1 sometimes start the index there:

```
1 int[] sales = {12, 30, 8, 25, 16};
2 int total = 0;
3 for (int i = 1; i < sales.length; i++)
4 {
5     total += sales[i];
6 }
7 System.out.println(total);
```

This prints 79, not 91. The loop never visits index 0, so the first day's 12 is left out. Java gives no error, because every index the loop does use is valid. Array indices start at 0, so a traversal of every element starts with `int i = 0`. (An enhanced for loop avoids this mistake, since it always visits every element.)

PAUSE AND TRY**Question 1** Tracing

What does this code print? Make a trace table with columns for `i`, `points[i]`, and `total`.

```
1 int[] points = {4, 7, 1, 9, 6};
2 int total = 0;
3 for (int i = 0; i < points.length; i += 2)
4 {
5     total += points[i];
6 }
7 System.out.println(total);
```

Question 2 Multiple choice

scores is the array {7, 9, 8, 6} and the int variable total holds its sum, 30. Which expression evaluates to 7.5?

- A. total / scores.length
- B. (double) (total / scores.length)
- C. (double) total / scores.length
- D. (double) total / scores.length()

4.5.2 Minimum and Maximum

Which day was warmest? Which store has the lowest price? These ask for the **maximum** or **minimum** element of an array. The plan is the best-so-far idea from Topic 2.9: remember the best value seen so far, and replace it whenever the current value beats it.

An array makes the starting value easy. The safest start is a real piece of data, and the first element, `values[0]`, is always there (as long as the array is not empty). Because element 0 is already the best so far, the loop can begin at index 1.

KEY IDEA

To find a maximum, start the best so far at `values[0]`, then loop from index 1 and replace it whenever `values[i]` is larger. For a minimum, replace it whenever `values[i]` is smaller.

WORKED EXAMPLE The warmest day

```
1 int[] temps = {58, 64, 61, 70, 66};
2 int warmest = temps[0];
3 for (int i = 1; i < temps.length; i++)
4 {
5     if (temps[i] > warmest)
6     {
7         warmest = temps[i];
8     }
9 }
10 System.out.println(warmest);
```

Line 2 makes `temps[0]`, which is 58, the warmest so far. The loop then compares each later element with it.

i	temps[i]	temps[i] > warmest?	warmest after the if
—	—	(line 2)	58
1	64	true	64
2	61	false	64
3	70	true	70
4	66	false	70

70

Starting the loop at index 0 would also print 70. The first pass would compare `58 > 58`, which is false, so nothing would change; it is one wasted comparison, not a bug.

Often you need to know **where** the minimum or maximum is, not only what it is: which day, which store. An array makes that natural, because every element has an index. Instead of remembering the best value, remember the best **index**. The value is then always available as `values[bestIndex]`.

WORKED EXAMPLE Where is the cheapest price?

Five stores sell the same game. Find the index of the cheapest one.

```

1 int[] prices = {42, 35, 50, 35, 47};
2 int minIndex = 0;
3 for (int i = 1; i < prices.length; i++)
4 {
5     if (prices[i] < prices[minIndex])
6     {
7         minIndex = i;
8     }
9 }
10 System.out.println(minIndex);
11 System.out.println(prices[minIndex]);

```

`minIndex` starts at 0, meaning "store 0 is the cheapest so far." Line 5 compares the current price with the price **at** `minIndex`.

i	prices[i]	prices[i] < prices[minIndex]?	minIndex after
1	35	35 < 42 true	1
2	50	50 < 35 false	1
3	35	35 < 35 false (a tie)	1
4	47	47 < 35 false	1

1
35

Store 3 ties store 1, but `35 < 35` is false, so the first cheapest store is kept. With `<=` instead, a tie would replace the best so far, and the answer would be the last cheapest store, index 3. One index variable gives you both answers: line 10 prints where, and line 11 prints what.

An enhanced for loop can find the minimum or maximum **value**, but it cannot find the index, because its variable holds a copy of the element and no position. When the question asks "where," use an indexed loop.

WATCH OUT

Comparing an element with an index. When a program keeps a best index, it is easy to compare the element with the index itself instead of with the element **at** that index:

```
1 int[] temps = {58, 64, 61, 70, 66};
2 int maxIndex = 0;
3 for (int i = 1; i < temps.length; i++)
4 {
5     if (temps[i] > maxIndex)
6     {
7         maxIndex = i;
8     }
9 }
10 System.out.println(maxIndex);
11 System.out.println(temps[maxIndex]);
```

```
4
66
```

The warmest day is index 3 (70 degrees), but this prints index 4. Line 5 asks whether `temps[i] > maxIndex`, comparing a temperature such as 64 with an index such as 0 or 1. Every temperature is bigger than every index here, so the condition is true on every pass and `maxIndex` ends at the last index. Java gives no error, since both are `int` values. The fix is to compare with the element at the best index:

```
1 if (temps[i] > temps[maxIndex])
```

With that change the program prints 3 and 70. When a loop uses both indices and elements, say each variable's role out loud: `i` and `maxIndex` are positions; `temps[i]` and `temps[maxIndex]` are temperatures.

PAUSE AND TRY

Question 3 Multiple choice

This code is meant to find the smallest element of an `int` array `values`:

```
1 int smallest = 0;
2 for (int v : values)
3 {
4     if (v < smallest)
5     {
6         smallest = v;
7     }
8 }
```

For which array does `smallest` end up holding a value that is **not** the smallest element?

- A. `{-4, -9, -2}`
- B. `{3, 8, 5}`
- C. `{0, 6, 2}`
- D. `{-1, 4, -3}`

4.5.3 Counting Elements with a Property

How many students passed? How many days were below freezing? These ask for the number of elements that have a **property**: a condition that is either true or false for each element, such as "at least 65" or "even." The counter from Topic 2.9 answers them. Start a count at 0 before the traversal, and add one inside an if statement only when the current element has the property.

KEY IDEA

To count the elements with a property, start a counter at 0 before the traversal, and inside the loop add one only when the current element makes the condition true.

WORKED EXAMPLE How many passed?

A passing score is 65 or more.

```
1 int[] scores = {72, 58, 90, 65, 81, 49};
2 int passing = 0;
3 for (int score : scores)
4 {
5     if (score >= 65)
6     {
7         passing++;
8     }
9 }
10 System.out.println(passing);
```

score	score >= 65?	passing
72	true	1
58	false	1
90	true	2
65	true	3
81	true	4
49	false	4

4

The 65 counts because the condition is `>=`. As in Topic 2.9, check the boundary value every time you read a counting condition.

Because an array keeps its values, the property can depend on something that is known only after a first traversal. "How many days were warmer than average?" needs the average before any counting can start. With an array, that is two traversals of the same data, one after the other.

WORKED EXAMPLE Days above the average

```
1 int[] temps = {58, 64, 61, 70, 66};
2 int total = 0;
3 for (int t : temps)
4 {
5     total += t;
6 }
7 double average = (double) total / temps.length;
8 System.out.println(average);
9 int aboveCount = 0;
10 for (int t : temps)
11 {
12     if (t > average)
13     {
14         aboveCount++;
15     }
16 }
17 System.out.println(aboveCount);
```

The first loop (lines 3–6) totals the five temperatures: $58 + 64 + 61 + 70 + 66$ is 319. Line 7 gives $319.0 / 5$, which is 63.8. The second loop (lines 10–16) visits the same elements again and counts those above 63.8:

t	t > 63.8?	aboveCount
58	false	0
64	true	1
61	false	1
70	true	2
66	true	3

63.8
3

In Topic 2.9, values read from a `Scanner` were used once and lost, so this question could not be answered without reading the input twice. An array lets the program look at its data as many times as it needs.

WATCH OUT

Testing the index instead of the element. With an indexed loop, both `i` and `nums[i]` are `int` values, and it is easy to test the wrong one. This code is meant to count the even numbers in the array:

```
1 int[] nums = {3, 8, 5, 6, 7};
2 int evens = 0;
3 for (int i = 0; i < nums.length; i++)
4 {
5     if (i % 2 == 0)
6     {
7         evens++;
8     }
9 }
10 System.out.println(evens);
```

It prints 3, but the array holds only two even numbers, 8 and 6. Line 5 tests `i % 2 == 0`, which is true for the even **indices** 0, 2, and 4, no matter what is stored there. The fix tests the element:

```
1 int[] nums = {3, 8, 5, 6, 7};
2 int evens = 0;
3 for (int i = 0; i < nums.length; i++)
4 {
5     if (nums[i] % 2 == 0)
6     {
7         evens++;
8     }
9 }
10 System.out.println(evens);
```

This prints 2. Ask of every condition: is this a question about a position or about the value stored there?

PAUSE AND TRY

Question 4 Tracing

What does this code print? Show which elements are counted.

```
1 int[] ages = {12, 25, 10, 19, 31, 20};
2 int teens = 0;
3 for (int age : ages)
4 {
5     if (age > 10 && age <= 20)
6     {
7         teens++;
8     }
9 }
10 System.out.println(teens);
```

Question 5 Debugging

This program is supposed to print how many games a team held its opponent to zero goals, which is 3. It does not compile. Find the error and correct it.

CountInsideLoop.java

```

1 public class CountInsideLoop
2 {
3     public static void main(String[] args)
4     {
5         int[] goals = {0, 2, 0, 1, 0};
6         for (int g : goals)
7         {
8             int shutouts = 0;
9             if (g == 0)
10            {
11                shutouts++;
12            }
13        }
14        System.out.println(shutouts);
15    }
16 }

```

4.5.4 At Least One or All

Some questions want a yes or no instead of a number. Is **any** account overdrawn? Did **every** student pass? In Topic 2.10 you answered these about the characters of a string with a flag: a `boolean` that starts with one value and switches when the loop finds something. The same two patterns work on an array.

- **At least one** element has the property: start the flag at `false` ("none found yet") and set it to `true` when an element has the property.
- **All** elements have the property: start the flag at `true` ("no problem found yet") and set it to `false` when an element does **not** have the property.

In both cases, once the flag switches, nothing switches it back, and the answer is read only after the loop.

WORKED EXAMPLE Is any account overdrawn?

```

1 int[] balances = {120, 45, -10, 300};
2 boolean anyOverdrawn = false;
3 for (int balance : balances)
4 {
5     if (balance < 0)
6     {
7         anyOverdrawn = true;
8     }
9 }
10 System.out.println(anyOverdrawn);

```

balance	balance < 0?	anyOverdrawn
120	false	false

45	false	false
-10	true	true
300	false	true

true

The 300 after the -10 does not undo anything, because there is no `else`. One overdrawn account is enough.

Inside a method there is a shorter way. As soon as the answer is known, the method can `return` it, and a `return` statement ends the method at once (Topic 3.5). For "all," the answer is known to be `false` the moment one element fails. The answer can be `true` only after **every** element has been checked, so `return true;` goes **after** the loop.

WORKED EXAMPLE Did every student pass?

AllPassing.java

```

1  public class AllPassing
2  {
3      public static boolean allPassing(int[] scores)
4      {
5          for (int score : scores)
6          {
7              if (score < 65)
8              {
9                  return false;
10             }
11         }
12         return true;
13     }
14
15     public static void main(String[] args)
16     {
17         int[] teamA = {80, 72, 91, 65};
18         int[] teamB = {88, 50, 94, 70};
19         int[] noScores = {};
20         System.out.println(allPassing(teamA));
21         System.out.println(allPassing(teamB));
22         System.out.println(allPassing(noScores));
23     }
24 }
```

For `teamA`, every score is at least 65, so the `if` on line 7 is never true, the loop finishes, and line 12 returns `true`. For `teamB`, the second score, 50, makes line 9 run: the method returns `false` immediately and never looks at 94 or 70.

The third call passes an empty array. The loop body never runs, so the method reaches line 12 and returns `true`. That sounds odd, but it follows the rule: "all" fails only when some element fails, and an empty array has no element that fails. In the same way, an "at least one" check on an empty array returns `false`, since there is no element that has the property.

```
true
false
true
```

Question	Flag starts at	Switch (or return) when the element...	Answer known early?
Does any element have it?	false	has the property → true	Yes, at a match
Do all elements have it?	true	lacks the property → false	Yes, at a failure

WATCH OUT

Returning from both branches inside the loop. A student wants `allPassing` to say true for passing scores and false for failing ones, and writes a `return` in each branch:

`AllPassingBug.java`

```

1 public class AllPassingBug
2 {
3     public static boolean allPassing(int[] scores)
4     {
5         for (int score : scores)
6         {
7             if (score < 65)
8             {
9                 return false;
10            }
11            else
12            {
13                return true;
14            }
15        }
16        return true;
17    }
18
19    public static void main(String[] args)
20    {
21        int[] teamB = {88, 50, 94, 70};
22        System.out.println(allPassing(teamB));
23    }
24 }
```

This prints `true` for `teamB`, even though 50 fails. On the first pass, `88 < 65` is false, so the `else` on line 11 runs and line 13 returns `true`. The method has ended after looking at one element. (The `return true;` on line 16 is still required: the compiler knows the loop might not run at all, so without a `return` after the loop, Java reports `missing return statement`.) The fix is to delete the `else` branch: a passing score proves nothing on its own, so only a failure may end the loop early, as in `AllPassing.java`.

PAUSE AND TRY**Question 6** Multiple choice

Each method below is supposed to return `true` exactly when every element of `nums` is even. Which one is correct for every array?

AllEvenChoices.java

```
1 public class AllEvenChoices
2 {
3     public static boolean choiceA(int[] nums)
4     {
5         boolean allEven = false;
6         for (int n : nums)
7         {
8             if (n % 2 == 0)
9             {
10                allEven = true;
11            }
12        }
13        return allEven;
14    }
15
16    public static boolean choiceB(int[] nums)
17    {
18        boolean allEven = true;
19        for (int n : nums)
20        {
21            if (n % 2 != 0)
22            {
23                allEven = false;
24            }
25        }
26        return allEven;
27    }
28
29    public static boolean choiceC(int[] nums)
30    {
31        boolean allEven = true;
32        for (int n : nums)
33        {
34            if (n % 2 == 0)
35            {
36                allEven = true;
37            }
38            else
39            {
40                allEven = false;
41            }
42        }
43        return allEven;
44    }
45
46    public static boolean choiceD(int[] nums)
```

```
47     {
48         boolean allEven = false;
49         for (int n : nums)
50         {
51             if (n % 2 != 0)
52             {
53                 allEven = false;
54             }
55         }
56         return allEven;
57     }
58
59     public static void main(String[] args)
60     {
61         int[] mixed = {4, 2, 7};
62         int[] evens = {4, 6, 2};
63         int[] oddFirst = {3, 6, 2};
64         System.out.println(choiceA(mixed) + " " + choiceA(evens) + " " +
choiceA(oddFirst));
65         System.out.println(choiceB(mixed) + " " + choiceB(evens) + " " +
choiceB(oddFirst));
66         System.out.println(choiceC(mixed) + " " + choiceC(evens) + " " +
choiceC(oddFirst));
67         System.out.println(choiceD(mixed) + " " + choiceD(evens) + " " +
choiceD(oddFirst));
68     }
69 }
```

- A. choiceA
- B. choiceB
- C. choiceC
- D. choiceD

Question 7 Tracing

What does this program print? Pay attention to how many words each call checks.

AnyLongWord.java

```

1 public class AnyLongWord
2 {
3     public static boolean anyLonger(String[] words, int limit)
4     {
5         for (String w : words)
6         {
7             System.out.println("checking " + w);
8             if (w.length() > limit)
9             {
10                 return true;
11             }
12         }
13         return false;
14     }
15
16     public static void main(String[] args)
17     {
18         String[] words = {"cat", "giraffe", "ox", "elephant"};
19         System.out.println(anyLonger(words, 5));
20         System.out.println(anyLonger(words, 8));
21     }
22 }

```

4.5.5 Consecutive Pairs

Did the step count go up from one day to the next? Did the same number come up twice in a row? These questions are about **consecutive pairs**: two elements that sit next to each other, at indices i and $i + 1$. An array of five elements has four such pairs:

index:	0	1	2	3	4
	4000	5200	5100	6000	7500
	_____/ _____/ _____/ _____/				
pair:	0	1	2	3	

Pair i is `values[i]` and `values[i + 1]`. The last pair starts at index 3, one before the last index, because its partner, index 4, is the last element. So the loop must stop **one index early**: $i < \text{values.length} - 1$. Going any further would ask for `values[i + 1]` past the end of the array.

KEY IDEA

To visit every consecutive pair, loop i from 0 while $i < \text{values.length} - 1$, and compare `values[i]` with `values[i + 1]`. An array of length n has $n - 1$ consecutive pairs.

WORKED EXAMPLE Counting day-to-day increases

```

1 int[] steps = {4000, 5200, 5100, 6000, 7500};
2 int increases = 0;
3 for (int i = 0; i < steps.length - 1; i++)
4 {
5     if (steps[i + 1] > steps[i])
6     {
7         increases++;
8     }
9 }
10 System.out.println(increases);

```

i	Pair steps[i], steps[i + 1]	Increase?	increases
0	4000, 5200	yes	1
1	5200, 5100	no	1
2	5100, 6000	yes	2
3	6000, 7500	yes	3

After `i` is 3, it becomes 4, and `4 < 4` is false. The loop ran four times for four pairs.

3

There is a second common way to write the same traversal: start at index 1 and pair each element with the one **before** it, `values[i - 1]` and `values[i]`. Now the loop can run all the way to the last index, `i < values.length`, because `i - 1` never drops below 0. This example also combines pairs with the maximum algorithm from 4.5.2: it finds the biggest rise from one day to the next.

WORKED EXAMPLE The biggest one-day temperature rise

```

1 int[] temps = {61, 58, 66, 70, 63};
2 int biggestRise = temps[1] - temps[0];
3 int riseDay = 1;
4 for (int i = 2; i < temps.length; i++)
5 {
6     int rise = temps[i] - temps[i - 1];
7     if (rise > biggestRise)
8     {
9         biggestRise = rise;
10        riseDay = i;
11    }
12 }
13 System.out.println(biggestRise);
14 System.out.println(riseDay);

```

Lines 2 and 3 make the first pair (indices 0 and 1) the best so far, so the loop on line 4 starts at index 2.

i	Pair temps[i - 1], temps[i]	rise	biggestRise, riseDay
(lines 2–3)	61, 58	-3	-3, 1

2	58, 66	8	8, 2
3	66, 70	4	8, 2
4	70, 63	-7	8, 2

8
2

The biggest rise was 8 degrees, arriving on day 2. The first rise is negative (a drop), which is why the best so far starts at a real difference rather than at 0.

Pair written as	Loop starts at	Loop condition
values[i], values[i + 1]	0	i < values.length - 1
values[i - 1], values[i]	1	i < values.length

WATCH OUT

Running one pair too far. If the pair loop uses the ordinary traversal condition, the last pass reaches past the end:

PairsOutOfBounds.java

```

1 public class PairsOutOfBounds
2 {
3     public static void main(String[] args)
4     {
5         int[] steps = {4000, 5200, 5100, 6000, 7500};
6         int increases = 0;
7         for (int i = 0; i < steps.length; i++)
8         {
9             if (steps[i + 1] > steps[i])
10            {
11                increases++;
12            }
13        }
14        System.out.println(increases);
15    }
16 }
```

The program crashes before it prints anything:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 5 out of
bounds for length 5
    at PairsOutOfBounds.main(PairsOutOfBounds.java:9)
```

(The exact wording can vary between Java versions.) On the last pass, `i` is 4, the last valid index, so line 9 asks for `steps[5]`, which does not exist (Topic 4.3). The fix is `i < steps.length - 1`. Whenever the body uses `i + 1`, the loop must stop one index earlier than a plain traversal.

PAUSE AND TRY**Question 8** Tracing

A die is rolled six times. What does this code print?

```
1 int[] rolls = {3, 3, 5, 5, 5, 2};
2 int repeats = 0;
3 for (int i = 0; i < rolls.length - 1; i++)
4 {
5     if (rolls[i] == rolls[i + 1])
6     {
7         repeats++;
8     }
9 }
10 System.out.println(repeats);
```

Question 9 Multiple choice

What does this program print?

InOrderCheck.java

```
1 public class InOrderCheck
2 {
3     public static boolean inOrder(int[] values)
4     {
5         for (int i = 1; i < values.length; i++)
6         {
7             if (values[i] < values[i - 1])
8             {
9                 return false;
10            }
11        }
12        return true;
13    }
14
15    public static void main(String[] args)
16    {
17        int[] a = {2, 5, 5, 9};
18        int[] b = {2, 5, 4, 9};
19        int[] c = {7};
20        System.out.println(inOrder(a));
21        System.out.println(inOrder(b));
22        System.out.println(inOrder(c));
23    }
24 }
```

- A. true, false, true
- B. false, false, true
- C. true, false, and then an `ArrayIndexOutOfBoundsException`
- D. true, true, true

4.5.6 Detecting Duplicates

A raffle must not sell the same ticket number twice. A seating chart must not list a student in two seats. Each question asks whether an array contains a **duplicate**: a value that appears at two or more different indices.

Consecutive pairs are not enough, because the two copies may be far apart. To be sure, every element must be compared with every **other** element. Think of a group of people shaking hands: the first person shakes hands with everyone after them, then the second person shakes hands with everyone after them, and so on. Nobody shakes their own hand, and no two people shake hands twice.

That plan is a nested loop (Topic 2.11). The outer loop picks one element, index `i`. The inner loop compares it with every element **after** it, so `j` starts at `i + 1`.

```
1 for (int i = 0; i < values.length; i++)
2 {
3     for (int j = i + 1; j < values.length; j++)
4     {
5         // compare values[i] with values[j]
6     }
7 }
```

Starting `j` at `i + 1` does two jobs. It skips `j == i`, so an element is never compared with itself. And it skips the indices before `i`, whose pairs were already compared on an earlier pass of the outer loop.

KEY IDEA

To check for duplicates, use a nested loop: the outer loop picks index `i`, the inner loop runs `j` from `i + 1` to the end, and a match `values[i] == values[j]` means the array has a duplicate.

WORKED EXAMPLE Was a ticket number sold twice?

This version uses the "at least one" flag from 4.5.4, and prints each match it finds so you can see where it is.

```
1 int[] tickets = {17, 42, 8, 42, 3};
2 boolean hasDuplicate = false;
3 for (int i = 0; i < tickets.length; i++)
4 {
5     for (int j = i + 1; j < tickets.length; j++)
6     {
7         if (tickets[i] == tickets[j])
8         {
9             hasDuplicate = true;
10            System.out.println("Match at " + i + " and " + j);
11        }
12    }
13 }
14 System.out.println(hasDuplicate);
```

i	tickets[i]	j values compared	Match?
0	17	1, 2, 3, 4	none

1	42	2, 3, 4	at j = 3 (42)
2	8	3, 4	none
3	42	4	none
4	3	(no j left)	—

Match at 1 and 3
true

Notice row 3: the second 42 is compared only with index 4. Its match with index 1 was already found when `i` was 1. When `i` is 4, `j` starts at 5, and `5 < 5` is false, so the inner loop does not run.

Answering "no duplicates" is harder than answering "yes," for the same reason "all" was harder than "at least one" in 4.5.4: one match proves a duplicate, but proving there is none takes every comparison. This version counts the comparisons.

WORKED EXAMPLE Proving there is no duplicate

```

1 int[] codes = {5, 9, 2, 7};
2 int comparisons = 0;
3 boolean hasDuplicate = false;
4 for (int i = 0; i < codes.length; i++)
5 {
6     for (int j = i + 1; j < codes.length; j++)
7     {
8         comparisons++;
9         if (codes[i] == codes[j])
10        {
11            hasDuplicate = true;
12        }
13    }
14 }
15 System.out.println(hasDuplicate);
16 System.out.println(comparisons);

```

false
6

The pairs compared are (0, 1), (0, 2), (0, 3), (1, 2), (1, 3), and (2, 3): $3 + 2 + 1 = 6$ comparisons, and none match, so the flag stays `false`. Every pair of indices was checked exactly once. As Topic 2.12 showed for nested loops, the number of comparisons grows quickly: an array of 100 elements needs $99 + 98 + \dots + 1 = 4950$ of them.

WATCH OUT

Starting the inner loop at 0. A student starts `j` at 0, thinking the inner loop should cover the whole array:

```
1 int[] codes = {5, 9, 2, 7};
2 boolean hasDuplicate = false;
3 for (int i = 0; i < codes.length; i++)
4 {
5     for (int j = 0; j < codes.length; j++)
6     {
7         if (codes[i] == codes[j])
8         {
9             hasDuplicate = true;
10        }
11    }
12 }
13 System.out.println(hasDuplicate);
```

This prints `true`, but `{5, 9, 2, 7}` has no duplicate. When `j` equals `i`, line 7 compares an element with itself, `codes[0] == codes[0]`, which is always true. Every array would be reported as having a duplicate. The fix is `int j = i + 1`, which also stops each pair from being compared twice.

PAUSE AND TRY

Question 10 Tracing

What does this code print? List the pairs of indices `(i, j)` that match.

```
1 int[] nums = {2, 7, 2, 2};
2 int pairs = 0;
3 for (int i = 0; i < nums.length; i++)
4 {
5     for (int j = i + 1; j < nums.length; j++)
6     {
7         if (nums[i] == nums[j])
8         {
9             pairs++;
10        }
11    }
12 }
13 System.out.println(pairs);
```

Question 11 Short answer

A student checks `{4, 1, 4}` for duplicates using consecutive pairs from 4.5.5:

```

1 int[] colors = {4, 1, 4};
2 boolean twoInARow = false;
3 for (int i = 0; i < colors.length - 1; i++)
4 {
5     if (colors[i] == colors[i + 1])
6     {
7         twoInARow = true;
8     }
9 }
10 System.out.println(twoInARow);

```

It prints `false`. Is that the right answer to "does the array contain a duplicate?" Explain why or why not, and say which algorithm answers the question correctly.

4.5.7 Shifting and Rotating

A line of people moves up when the person at the front is served. A scrolling sign moves every letter one place to the left, and the letter that falls off the front reappears at the back. Both move every element of an array one position. The difference is what happens to the element that falls off the end.

- **Shifting** moves every element one position left (toward index 0) or right. The element pushed off the end is lost, and the empty spot at the other end gets a fill value, such as 0.
- **Rotating** moves every element the same way, but the element pushed off one end wraps around to the empty spot at the other end. Nothing is lost.

start:	10	20	30	40	50	
shift left:	20	30	40	50	0	(10 is lost; 0 fills the end)
rotate left:	20	30	40	50	10	(10 wraps to the end)
shift right:	0	10	20	30	40	(50 is lost; 0 fills the front)
rotate right:	50	10	20	30	40	(50 wraps to the front)

The tool is the assignment `line[i] = line[i + 1];`, which copies the element at index `i + 1` into index `i`. Each copy **overwrites** what was there, so two questions decide whether the algorithm works: in which order are the copies made, and is the overwritten value still needed?

WORKED EXAMPLE Shifting left

```

1  int[] line = {10, 20, 30, 40, 50};
2  for (int i = 0; i < line.length - 1; i++)
3  {
4      line[i] = line[i + 1];
5  }
6  line[line.length - 1] = 0;
7  for (int value : line)
8  {
9      System.out.print(value + " ");
10 }
11 System.out.println();

```

The loop is a consecutive-pair loop from 4.5.5: it stops at `i < line.length - 1` because it uses `line[i + 1]`. Watch the array after each pass of line 4:

After	Line
start	10, 20, 30, 40, 50
i = 0	20, 20, 30, 40, 50
i = 1	20, 30, 30, 40, 50
i = 2	20, 30, 40, 40, 50
i = 3	20, 30, 40, 50, 50
line 6	20, 30, 40, 50, 0

```
20 30 40 50 0
```

Each pass copies a value one place left. Moving left to right is safe: from the second pass on, each pass overwrites a value that the pass before it already copied one place to the left, so nothing still needed is destroyed. The first value, 10, was overwritten on the very first pass; that is what "lost" means in a shift. After the loop, the last two spots hold the same value, so line 6 puts the fill value 0 at the end. Lines 7–10 print each element followed by a space.

To **rotate**, keep the element that falls off before the loop overwrites it, then put it in the empty spot afterward. Rotating to the right also changes the direction of the loop, for a reason the Watch Out below makes clear.

WORKED EXAMPLE Rotating right

```

1 int[] line = {10, 20, 30, 40, 50};
2 int last = line[line.length - 1];
3 for (int i = line.length - 1; i > 0; i--)
4 {
5     line[i] = line[i - 1];
6 }
7 line[0] = last;
8 for (int value : line)
9 {
10     System.out.print(value + " ");
11 }
12 System.out.println();

```

Rotating right pushes the last element off, so line 2 saves `line[4]`, which is 50, in `last`. The loop starts at the **last** index and moves toward the front, copying the element on the left into each spot.

After	Line
line 2	10, 20, 30, 40, 50 (last is 50)
i = 4	10, 20, 30, 40, 40
i = 3	10, 20, 30, 30, 40
i = 2	10, 20, 20, 30, 40
i = 1	10, 10, 20, 30, 40
line 7	50, 10, 20, 30, 40

```
50 10 20 30 40
```

The loop condition `i > 0` stops before `i` reaches 0, because `line[i - 1]` would be index -1. Line 7 puts the saved 50 into the empty front spot. Without line 2, the 50 would have been overwritten on the first pass and could not wrap around.

KEY IDEA

To shift or rotate in place, copy each element into its neighbor's spot, moving through the array **opposite** to the direction the elements move: left to right for a left move, right to left for a right move. To rotate, save the element that falls off before the loop and put it in the empty spot after it.

WATCH OUT

Shifting right while looping forward. A student shifts right with the forward loop from the shift-left example, changing only the copy:

```

1 int[] line = {10, 20, 30, 40, 50};
2 for (int i = 0; i < line.length - 1; i++)
3 {
4     line[i + 1] = line[i];
5 }
6 line[0] = 0;
7 for (int value : line)
8 {
9     System.out.print(value + " ");
10 }
11 System.out.println();

```

```
0 10 10 10 10
```

Instead of 0 10 20 30 40, almost every spot holds 10. On the first pass, line 4 copies 10 into index 1, overwriting the 20 before it has been moved. On the next pass, the "element at index 1" is now 10, so 10 is copied again, and so on down the line. Each copy destroys the value the next pass needs. The fix is to loop from the end toward the front, as the rotate-right example does: `for (int i = line.length - 1; i > 0; i--)` with `line[i] = line[i - 1];`. Then each value is moved out of the way before anything overwrites it, and the program prints 0 10 20 30 40.

PAUSE AND TRY

Question 12 Tracing

What does this code print? Show the array after each pass of the loop.

```

1 int[] queue = {3, 6, 9, 12};
2 int first = queue[0];
3 for (int i = 0; i < queue.length - 1; i++)
4 {
5     queue[i] = queue[i + 1];
6 }
7 queue[queue.length - 1] = first;
8 for (int value : queue)
9 {
10     System.out.print(value + " ");
11 }
12 System.out.println();

```

Question 13 Multiple choice

A student tries to rotate {10, 20, 30, 40, 50} to the left without saving the first element:

```
1 int[] line = {10, 20, 30, 40, 50};
2 for (int i = 0; i < line.length - 1; i++)
3 {
4     line[i] = line[i + 1];
5 }
6 line[line.length - 1] = line[0];
7 for (int value : line)
8 {
9     System.out.print(value + " ");
10 }
11 System.out.println();
```

What does it print?

- A. 20 30 40 50 10
- B. 20 30 40 50 20
- C. 20 30 40 50 50
- D. 10 20 30 40 50

4.5.8 Reversing an Array

A race app lists runners from first place to last, but the award ceremony calls them from last to first. To **reverse** an array is to put its elements in the opposite order: the first becomes the last, the second becomes the second-to-last, and so on.

Each element trades places with its **mirror**: the element the same distance from the other end. For an array of length n , the mirror of index i is index $n - 1 - i$. Index 0 pairs with index $n - 1$, index 1 with index $n - 2$, and so on toward the middle.

Trading two elements needs care. If you write `a[i] = a[mirror]`; first, the old `a[i]` is overwritten and gone. So a **swap** uses a third variable: copy one value into `temp`, overwrite that spot with the other value, then put `temp` into the second spot. It is like swapping the contents of two full cups: you need an empty third cup.

```
1 int temp = values[i];
2 values[i] = values[mirror];
3 values[mirror] = temp;
```

KEY IDEA

To reverse an array in place, loop `i` from 0 while `i < values.length / 2`, and swap `values[i]` with its mirror `values[values.length - 1 - i]` using a temporary variable.

WORKED EXAMPLE Reversing the finishing order

```

1 String[] runners = {"Ava", "Ben", "Cal", "Dee", "Eli"};
2 for (int i = 0; i < runners.length / 2; i++)
3 {
4     int mirror = runners.length - 1 - i;
5     String temp = runners[i];
6     runners[i] = runners[mirror];
7     runners[mirror] = temp;
8 }
9 for (String name : runners)
10 {
11     System.out.print(name + " ");
12 }
13 System.out.println();

```

The array has 5 elements, so `runners.length / 2` is 2 (integer division), and the loop runs for `i = 0` and 1.

i	mirror (line 4)	Swap	runners after line 7
—	—	—	Ava, Ben, Cal, Dee, Eli
0	4	Ava ↔ Eli	Eli, Ben, Cal, Dee, Ava
1	3	Ben ↔ Dee	Eli, Dee, Cal, Ben, Ava

```
Eli Dee Cal Ben Ava
```

Why stop at the halfway point? Each swap fixes two positions at once, one in each half. After the first half has been swapped with the second half, every element is in place. The middle element, `Cal` at index 2, is its own mirror ($5 - 1 - 2 = 2$), so it never needs to move. The same loop works on an `int` array or an array of any other type; only the type of `temp` changes.

Sometimes the original order must be kept. Then build a **new** array of the same length and fill it in reverse. No swap is needed, because nothing in the original is overwritten.

WORKED EXAMPLE A reversed copy

```
1 int[] nums = {1, 4, 9, 16};
2 int[] backward = new int[nums.length];
3 for (int i = 0; i < nums.length; i++)
4 {
5     backward[nums.length - 1 - i] = nums[i];
6 }
7 for (int value : backward)
8 {
9     System.out.print(value + " ");
10 }
11 System.out.println();
12 for (int value : nums)
13 {
14     System.out.print(value + " ");
15 }
16 System.out.println();
```

Line 2 creates an array of four 0s (Topic 4.3). Line 5 copies `nums[i]` into the mirror position of the new array: `nums[0]` goes to index 3, `nums[1]` to index 2, and so on. This loop visits **every** index, because each pass fills one spot of the new array.

```
16 9 4 1
1 4 9 16
```

The second line shows that `nums` is unchanged.

WATCH OUT

Swapping all the way to the end. It seems natural to loop over every index:

```
1 int[] nums = {1, 4, 9, 16};
2 for (int i = 0; i < nums.length; i++)
3 {
4     int mirror = nums.length - 1 - i;
5     int temp = nums[i];
6     nums[i] = nums[mirror];
7     nums[mirror] = temp;
8 }
9 for (int value : nums)
10 {
11     System.out.print(value + " ");
12 }
13 System.out.println();
```

```
1 4 9 16
```

The array comes out in its original order. For `i = 0` and 1, the swaps reverse the array. Then `i = 2` swaps index 2 with its mirror, index 1, which trades back the pair that `i = 1` just swapped;

`i = 3` undoes the swap made at `i = 0`. Every pair is swapped twice, and two swaps cancel out. Stop at `i < nums.length / 2`.

PAUSE AND TRY

Question 14 Debugging

This code is supposed to reverse `{1, 2, 3, 4}`, but it prints `4 3 3 4`. Find the error and correct it.

```
1 int[] nums = {1, 2, 3, 4};
2 for (int i = 0; i < nums.length / 2; i++)
3 {
4     int mirror = nums.length - 1 - i;
5     nums[i] = nums[mirror];
6     nums[mirror] = nums[i];
7 }
8 for (int value : nums)
9 {
10     System.out.print(value + " ");
11 }
12 System.out.println();
```

Question 15 Multiple choice

The array `nums` holds `{1, 2, 3, 4, 5, 6, 7}`. The loop below swaps each `nums[i]` with `nums[nums.length - 1 - i]` using `temp`, as in the worked example. Which loop header reverses the array?

```
1 for (/* header */)
2 {
3     int mirror = nums.length - 1 - i;
4     int temp = nums[i];
5     nums[i] = nums[mirror];
6     nums[mirror] = temp;
7 }
```

- A. `int i = 0; i < nums.length / 2; i++`
- B. `int i = 0; i < nums.length; i++`
- C. `int i = 0; i < 2; i++`
- D. `int i = 0; i < nums.length - 1; i++`

Putting it together

Real programs give each algorithm its own method, then call the methods on the same array. This program tracks one student's six quiz scores, oldest first. Each method is one algorithm from this chapter, labeled with a comment. The methods are `public static` class methods (Topic 3.7), so `main` calls them by name.

WORKED EXAMPLE A quiz score report

QuizTracker.java

```
1 public class QuizTracker
2 {
3     // sum, then divide by the length
4     public static double average(int[] scores)
5     {
6         int total = 0;
7         for (int score : scores)
8         {
9             total += score;
10        }
11        return (double) total / scores.length;
12    }
13
14    // maximum: remember the index of the best so far
15    public static int bestIndex(int[] scores)
16    {
17        int best = 0;
18        for (int i = 1; i < scores.length; i++)
19        {
20            if (scores[i] > scores[best])
21            {
22                best = i;
23            }
24        }
25        return best;
26    }
27
28    // count: how many scores are below the cutoff
29    public static int countBelow(int[] scores, int cutoff)
30    {
31        int count = 0;
32        for (int score : scores)
33        {
34            if (score < cutoff)
35            {
36                count++;
37            }
38        }
39        return count;
40    }
41
42    // consecutive pairs: how many times a score beat the one before it
43    public static int improvements(int[] scores)
44    {
45        int count = 0;
46        for (int i = 0; i < scores.length - 1; i++)
47        {
48            if (scores[i + 1] > scores[i])
49            {
50                count++;
51            }
52        }
53    }
```

```

53         return count;
54     }
55
56     // duplicates: does any score appear more than once?
57     public static boolean hasRepeat(int[] scores)
58     {
59         for (int i = 0; i < scores.length; i++)
60         {
61             for (int j = i + 1; j < scores.length; j++)
62             {
63                 if (scores[i] == scores[j])
64                 {
65                     return true;
66                 }
67             }
68         }
69         return false;
70     }
71
72     // reverse: swap each element in the first half with its mirror
73     public static void reverse(int[] scores)
74     {
75         for (int i = 0; i < scores.length / 2; i++)
76         {
77             int mirror = scores.length - 1 - i;
78             int temp = scores[i];
79             scores[i] = scores[mirror];
80             scores[mirror] = temp;
81         }
82     }
83
84     public static void main(String[] args)
85     {
86         int[] quizzes = {78, 85, 85, 62, 90, 86};
87         System.out.println("Average: " + average(quizzes));
88         int best = bestIndex(quizzes);
89         System.out.println("Best: quiz " + best + " with " + quizzes[best]);
90         System.out.println("Below 80: " + countBelow(quizzes, 80));
91         System.out.println("Improvements: " + improvements(quizzes));
92         System.out.println("Repeat score: " + hasRepeat(quizzes));
93         reverse(quizzes);
94         System.out.print("Newest first: ");
95         for (int score : quizzes)
96         {
97             System.out.print(score + " ");
98         }
99         System.out.println();
100     }
101 }

```

The array is {78, 85, 85, 62, 90, 86}. Here is what each call works out:

Method	Algorithm	Work	Result
average	sum and average	78 + 85 + 85 + 62 + 90 + 86 = 486; 486.0 / 6	81.0

bestIndex	maximum (index)	best index moves 0 → 1 → 4; ties at 85 do not replace	4
countBelow	counting	78 and 62 are below 80	2
improvements	consecutive pairs	78→85 up, 85→85, 85→62, 62→90 up, 90→86	2
hasRepeat	duplicates	i = 1, j = 2: 85 equals 85, so it returns true at once	true
reverse	reversing	swaps (0, 5), (1, 4), (2, 3)	changes the array

```

Average: 81.0
Best: quiz 4 with 90
Below 80: 2
Improvements: 2
Repeat score: true
Newest first: 86 90 62 85 85 78

```

Two details are worth a closer look. First, `85 → 85` is not an improvement, because `85 > 85` is false. Second, `reverse` is a `void` method, yet the last line in `main` prints the scores in reverse. As in Topic 3.6, the parameter `scores` receives a copy of the **reference**, so it refers to the very same array as `quizzes`. The swaps change that one array, and `main` sees the change. The other methods only read the array, so they leave it as they found it, which is what a method should do unless changing its parameter is its job.

Vocabulary

Term	Definition	Example
standard array algorithms	Well-known array traversals that answer common questions about a list of values	Sum, count, maximum, reverse
maximum / minimum	The largest / smallest element of an array	70 in {58, 64, 61, 70, 66}
property	A condition that is true or false for each element	"at least 65"
consecutive pairs	Two elements next to each other, at indices <code>i</code> and <code>i + 1</code>	5200 and 5100 in the step array
duplicate	A value that appears at two or more different indices	42 in {17, 42, 8, 42, 3}
shifting	Moving every element one position; the element pushed off the end is lost	{10, 20, 30} → {20, 30, 0}
rotating	Moving every element one position; the element pushed off one end wraps to the other	{10, 20, 30} → {20, 30, 10}

reverse	Put the elements in the opposite order	<code>{1, 4, 9} → {9, 4, 1}</code>
mirror	The index the same distance from the other end: <code>length - 1 - i</code>	Index 1's mirror in length 5 is index 3
swap	Trade the values in two places, using a temporary variable	<code>temp = a[i]; a[i] = a[j]; a[j] = temp;</code>

Chapter summary

- Sums, averages, and counts use an accumulator that starts before the traversal and changes inside it. An array's `length` gives the number of elements for an average, and a traversal must start at index `0` to include every element.
- A minimum or maximum starts its best so far at `values[0]`. Keeping the best **index** instead of the best value tells you both where the best element is and what it is; compare elements with elements, never with an index.
- "At least one" starts a flag at `false` and "all" starts it at `true`; in a method, a `return` inside the loop can end early, but only for the answer that one element can prove.
- Consecutive pairs use `values[i]` and `values[i + 1]` with `i < values.length - 1`. Duplicates need a nested loop with `j` starting at `i + 1`, so every pair of different indices is compared exactly once.
- Shifting and rotating copy each element into a neighbor's spot, moving through the array opposite to the direction the elements travel; rotating saves the element that falls off first. Reversing swaps each element in the first half with its mirror, using a temporary variable, and stops at `length / 2`.

Review questions

Question 16 Multiple choice

What does `mystery` determine about its array parameter?

MysteryIncreasing.java

```
1 public class MysteryIncreasing
2 {
3     public static boolean mystery(int[] values)
4     {
5         for (int i = 1; i < values.length; i++)
6         {
7             if (values[i] <= values[i - 1])
8             {
9                 return false;
10            }
11        }
12        return true;
13    }
14
15    public static void main(String[] args)
16    {
17        int[] a = {2, 5, 8, 9};
18        int[] b = {2, 5, 5, 9};
19        int[] c = {1, 5, 3};
20        int[] d = {3, 1, 2};
21        System.out.println(mystery(a));
22        System.out.println(mystery(b));
23        System.out.println(mystery(c));
24        System.out.println(mystery(d));
25    }
26 }
```

- A. Whether each element is greater than the element before it
- B. Whether each element is greater than or equal to the element before it
- C. Whether the first element is the smallest element
- D. Whether the array has no duplicates

Question 17 Tracing

This code moves the elements into a new array. What does it print, and what has it done to the order?

```
1 int[] seats = {5, 8, 1, 6};
2 int[] moved = new int[seats.length];
3 for (int i = 0; i < seats.length; i++)
4 {
5     moved[(i + 1) % seats.length] = seats[i];
6 }
7 for (int value : moved)
8 {
9     System.out.print(value + " ");
10 }
11 System.out.println();
```

Question 18 Debugging

This code is supposed to report whether `ids` contains a duplicate. For `{6, 2, 6}` it prints `false`. Find the error and correct it.

```
1 int[] ids = {6, 2, 6};
2 boolean hasDuplicate = false;
3 for (int i = 0; i < ids.length; i++)
4 {
5     for (int j = i + 1; j < ids.length - 1; j++)
6     {
7         if (ids[i] == ids[j])
8         {
9             hasDuplicate = true;
10        }
11    }
12 }
13 System.out.println(hasDuplicate);
```

Question 19 Tracing

An element is a "peak" when it is greater than both of its neighbors. What does this code print?

```
1 int[] heights = {3, 7, 4, 8, 8, 2, 5};
2 int peaks = 0;
3 for (int i = 1; i < heights.length - 1; i++)
4 {
5     if (heights[i] > heights[i - 1] && heights[i] > heights[i + 1])
6     {
7         peaks++;
8     }
9 }
10 System.out.println(peaks);
```

Question 20 Multiple choice

What does this code print?

```
1 String[] names = {"Maya", "Leo", "Zoe", "Ana", "Kai"};
2 String pick = names[0];
3 for (int i = 1; i < names.length; i++)
4 {
5     if (names[i].compareTo(pick) > 0)
6     {
7         pick = names[i];
8     }
9 }
10 System.out.println(pick);
```

- A. Ana
- B. Zoe
- C. Maya
- D. Kai

Question 21 Short answer

RotateRight in 4.5.7 saves the last element before its loop, and its loop runs from the last index down to 1. Explain why each of those two choices is needed. What would go wrong without each one?

Question 22 Tracing

What does this program print?

DropFirstTwice.java

```
1 public class DropFirstTwice
2 {
3     public static void dropFirst(int[] list)
4     {
5         for (int i = 0; i < list.length - 1; i++)
6         {
7             list[i] = list[i + 1];
8         }
9         list[list.length - 1] = 0;
10    }
11
12    public static void main(String[] args)
13    {
14        int[] orders = {4, 9, 2, 7};
15        dropFirst(orders);
16        dropFirst(orders);
17        int total = 0;
18        for (int order : orders)
19        {
20            System.out.print(order + " ");
21            total += order;
22        }
23        System.out.println();
24        System.out.println(total);
25    }
26 }
```

Question 23 Multiple choice

What does this code print? (Math.abs returns the absolute value, Topic 1.11.)

```
1 double[] readings = {20.0, 22.5, 19.0, 26.5};
2 double sum = 0.0;
3 for (double r : readings)
4 {
5     sum += r;
6 }
7 double average = sum / readings.length;
8 boolean allClose = true;
9 for (double r : readings)
10 {
11     if (Math.abs(r - average) > 4.0)
12     {
13         allClose = false;
14     }
15 }
16 System.out.println(average);
17 System.out.println(allClose);
```

- A. 22.0 and then true
- B. 22.0 and then false

- C. 88.0 and then false
- D. 22.0 and then 4.5

Reflection

- ☐ I can trace the sum and average of an array and explain why the traversal starts at index 0.
Reread: 4.5.1 Sum and Average of an Array
- ☐ I can find the minimum or maximum of an array, and its index, without mixing up indices and elements. Reread: 4.5.2 Minimum and Maximum
- ☐ I can count the elements that have a property, including at the boundary value. Reread: 4.5.3 Counting Elements with a Property
- ☐ I can decide whether at least one element or every element has a property, and explain where each return belongs. Reread: 4.5.4 At Least One or All
- ☐ I can visit every consecutive pair without going past the end of the array. Reread: 4.5.5 Consecutive Pairs
- ☐ I can explain why the duplicate check uses a nested loop with j starting at $i + 1$. Reread: 4.5.6 Detecting Duplicates
- ☐ I can predict the result of shifting or rotating an array left or right. Reread: 4.5.7 Shifting and Rotating
- ☐ I can reverse an array with swaps and explain why the loop stops halfway. Reread: 4.5.8 Reversing an Array
- ☐ I can trace a program that runs several array algorithms on the same array. Reread: Putting it together

Check Your Reasoning

Answers and explanations for every question, numbered to match.

Question 1

It prints 11.

i	points[i]	total after line 5
0	4	4
2	1	5
4	6	11

The update `i += 2` makes the loop visit only indices 0, 2, and 4. After index 4, `i` becomes 6, and `6 < 5` is false. The elements at indices 1 and 3 (7 and 9) are never added. If you got 27, you added every element; the update decides which indices are visited.

Reread 4.5.1 *Sum and Average of an Array* if your answer differed.

Question 2

C. The cast makes `total` the `double` 30.0 before the division, and `scores.length` is 4, so the result is 7.5.

- **A** is wrong: both values are `int`, so the division drops the decimal part and gives 7.
- **B** is wrong: the parentheses make `30 / 4` happen first, which is already 7; the cast then gives 7.0.
- **D** is wrong: it does not compile. For an array, `length` is not a method, so it has no parentheses. Java reports `cannot find symbol` because there is no method named `length()` for an array. (A `String` uses `length()` with parentheses; an array uses `length` without them.)

Reread 4.5.1 *Sum and Average of an Array* if you chose differently.

Question 3

B. Every element of {3, 8, 5} is greater than 0, so `v < smallest` is never true and `smallest` stays 0, a value that is not in the array. The real minimum is 3. Starting at `values[0]` instead of 0 fixes it.

- **A** is wrong: every element is below 0, so the made-up start is replaced; `smallest` ends at -9, which is correct.
- **C** is wrong: the answer 0 is correct here, because 0 really is the smallest element.
- **D** is wrong: -1 and then -3 replace the start, ending at -3, which is correct.

Reread 4.5.2 *Minimum and Maximum* if you chose differently.

Question 4

It prints 3.

age	age > 10	age <= 20	Counted?
12	true	true	yes (1)
25	true	false	no
10	false	(skipped)	no

19	true	true	yes (2)
31	true	false	no
20	true	true	yes (3)

10 is not counted because `10 > 10` is false; 20 is counted because `20 <= 20` is true. If you got 4, you counted 10; if you got 2, you left out 20.

Reread 4.5.3 *Counting Elements with a Property* if your answer differed.

Question 5

Move the declaration `int shutouts = 0;` from line 8 to just before the loop (after line 5):

```

1  int shutouts = 0;
2  for (int g : goals)
3  {
4      if (g == 0)
5      {
6          shutouts++;
7      }
8  }
9  System.out.println(shutouts);

```

Declared inside the loop body, `shutouts` is a local variable of that block (Topic 3.8), so it does not exist after the loop. Line 14 then uses a name Java cannot see, and the compiler reports `cannot find symbol`. There is a second problem hiding behind the first: even inside the loop, the declaration would reset the counter to 0 on every pass, so it could never get past 1. A counter must be created once, before the loop. With the fix, the program prints 3.

Reread 4.5.3 *Counting Elements with a Property* if your answer differed.

Question 6

B. It starts at `true` and switches to `false` when it finds an odd element. Nothing switches it back, so one odd element anywhere makes the answer `false`. The `main` method prints each choice's answers for {4, 2, 7} (should be false), {4, 6, 2} (should be true), and {3, 6, 2} (should be false):

```

true true true
false true false
false true true
false false false

```

Only the second line, `choiceB`, is right all three times.

- **A** is wrong: it is the "at least one" pattern. It returns `true` if **any** element is even, so it says `true` for all three arrays.
- **C** is wrong: the `else` lets every pass overwrite the flag, so it reports only whether the **last** element is even. It is wrong for {3, 6, 2}.
- **D** is wrong: the flag starts at `false` and can only be set to `false`, so it always returns `false`, even for {4, 6, 2}.

Reread 4.5.4 *At Least One or All* if you chose differently.

Question 7

```
checking cat
checking giraffe
true
checking cat
checking giraffe
checking ox
checking elephant
false
```

First call, limit 5: "cat" has length 3, not more than 5. "giraffe" has length 7, so line 10 returns `true` right away; "ox" and "elephant" are never checked. Second call, limit 8: no word is longer than 8 ("elephant" has exactly 8), so all four are checked and line 13 returns `false`. An "at least one" method can stop at the first match, but it can say `false` only after checking everything.

Reread 4.5.4 *At Least One or All* if your answer differed.

Question 8

It prints 3.

i	Pair	Equal?	repeats
0	3 and 3	yes	1
1	3 and 5	no	1
2	5 and 5	yes	2
3	5 and 5	yes	3
4	5 and 2	no	3

The three 5s form two overlapping pairs (indices 2–3 and 3–4), and each pair is checked on its own. Six elements give five pairs. If you got 2, you counted the run of 5s only once.

Reread 4.5.5 *Consecutive Pairs* if your answer differed.

Question 9

A. `inOrder` combines consecutive pairs with the "all" pattern: it returns `false` as soon as an element is smaller than the one before it.

- For {2, 5, 5, 9}, no element is smaller than the one before it (`5 < 5` is false), so it returns `true`.
- For {2, 5, 4, 9}, at `i = 2`, `4 < 5` is true, so it returns `false`.
- For {7}, the loop starts at `i = 1`, and `1 < 1` is false, so the body never runs and the method returns `true`.

The other choices:

- B** is wrong: equal neighbors do not fail the test, because the condition is `<`, not `<=`.
- C** is wrong: the loop never touches an index outside the array. With one element it does not run at all.
- D** is wrong: 4 comes after 5 in the second array, which fails the test.

Reread 4.5.5 *Consecutive Pairs* if you chose differently.

Question 10

It prints 3.

i	j compared	Matching pairs
0	1, 2, 3	(0, 2) and (0, 3)
1	2, 3	none (7 vs 2s)
2	3	(2, 3)
3	none	—

The three 2s at indices 0, 2, and 3 can be paired three ways, and each pair is counted once. So this code counts matching **pairs**, not repeated values: only one value repeats, but it forms three pairs. If you got 2, you missed the pair (2, 3).

Reread 4.5.6 *Detecting Duplicates* if your answer differed.

Question 11

Model answer: No. The array has a duplicate, the two 4s, but they are at indices 0 and 2, which are not neighbors. The pair loop only compares (0, 1) and (1, 2), so it never compares the two 4s. It answers a different question, "are two equal values next to each other?" To find any duplicate, use the nested loop with `j` starting at `i + 1`, which compares every pair of indices, including (0, 2).

A complete answer says:

- The answer `false` is wrong: 4 appears twice.
- Consecutive pairs compare only neighbors, and the copies are not next to each other.
- The nested-loop duplicate check compares every pair, so it finds them.

Reread 4.5.6 *Detecting Duplicates* if your reasoning differed.

Question 12

It prints 6 9 12 3.

After	queue
line 2	3, 6, 9, 12 (first is 3)
<code>i = 0</code>	6, 6, 9, 12
<code>i = 1</code>	6, 9, 9, 12
<code>i = 2</code>	6, 9, 12, 12
line 7	6, 9, 12, 3

This is a left rotation: the loop shifts left, and line 7 puts the saved first element at the end. If you got 6 9 12 0 or 6 9 12 12, you treated it as a shift and forgot that `first` wraps around.

Reread 4.5.7 *Shifting and Rotating* if your answer differed.

Question 13

B. The loop shifts left, giving 20, 30, 40, 50, 50. Then line 6 copies `line[0]` into the last spot, but `line[0]` is no longer 10: the first pass overwrote it with 20. So the last spot becomes 20.

- **A** is wrong: that is the correct rotation, which needs the 10 saved in a variable before the loop.
- **C** is wrong: that is the array after the loop but before line 6.
- **D** is wrong: the loop does change the array; nothing restores the original order.

Reread 4.5.7 *Shifting and Rotating* if you chose differently.

Question 14

Lines 5 and 6 trade the two elements without a temporary variable. Replace them with a three-step swap:

```
1 int temp = nums[i];
2 nums[i] = nums[mirror];
3 nums[mirror] = temp;
```

When `i` is 0, line 5 copies 4 into index 0, which destroys the 1. Line 6 then copies `nums[0]`, now 4, into index 3, so both ends hold 4. The same happens in the middle with 3. The original values 1 and 2 are lost. With `temp` holding the first value before it is overwritten, the code prints 4 3 2 1.

Reread 4.5.8 *Reversing an Array* if your answer differed.

Question 15

A. `nums.length / 2` is `7 / 2`, which is 3, so the loop swaps the pairs (0, 6), (1, 5), and (2, 4). The middle element at index 3 stays put. Result: 7 6 5 4 3 2 1.

- **B** is wrong: every pair is swapped twice, so the array ends in its original order, 1 2 3 4 5 6 7.
- **C** is wrong: it swaps only two pairs, (0, 6) and (1, 5), leaving the pair (2, 4) unswapped:
7 6 3 4 5 2 1.
- **D** is wrong: it goes past the middle. The swaps for `i = 4` and 5 undo the swaps for `i = 2` and 1, giving 7 2 3 4 5 6 1.

Reread 4.5.8 *Reversing an Array* if you chose differently.

Question 16

A. The loop visits each consecutive pair as `values[i - 1]` and `values[i]`, and returns `false` as soon as an element is less than **or equal to** the one before it. So it returns `true` only when every element is strictly greater than its neighbor on the left. `main` prints `true` for {2, 5, 8, 9} and `false` for the other three arrays.

- **B** is wrong: the condition `<=` fails on equal neighbors, so {2, 5, 5, 9} gives `false`. Choice B would need `<` in the condition.
- **C** is wrong: in {1, 5, 3}, the first element is the smallest, but `3 <= 5` makes the method return `false`.
- **D** is wrong: {3, 1, 2} has no duplicates, but `1 <= 3` makes the method return `false`. Checking for duplicates needs a nested loop, not neighbors.

Reread 4.5.5 *Consecutive Pairs* and 4.5.4 *At Least One or All* if you chose differently.

Question 17

It prints 6 5 8 1: a rotation to the right by one.

i	seats[i]	(i + 1) % 4	moved after line 5
0	5	1	0, 5, 0, 0
1	8	2	0, 5, 8, 0
2	1	3	0, 5, 8, 1
3	6	0	6, 5, 8, 1

Each element goes one index to the right. For the last element, $(3 + 1) \% 4$ is 0, so the remainder operator (Topic 1.3) wraps it around to the front, which is exactly what makes this a rotation and not a shift. Because the results go into a separate array, nothing is overwritten, so no saved value or backward loop is needed. If you got 8 1 6 5, you rotated left.

Reread 4.5.7 *Shifting and Rotating* if your answer differed.

Question 18

The inner loop's condition on line 5 stops one index too early. Change it to:

```
1 for (int j = i + 1; j < ids.length; j++)
```

With `j < ids.length - 1`, `j` never reaches the last index, 2. When `i` is 0, the inner loop compares only index 1, so the two 6s at indices 0 and 2 are never compared. The `- 1` belongs in a consecutive-pair loop, which uses `i + 1`; here `j` itself is the index being read, so it must be allowed to reach the last index. With the fix, the pair (0, 2) is compared and the program prints `true`.

Reread 4.5.6 *Detecting Duplicates* if your answer differed.

Question 19

It prints 1.

i	Left, element, right	Greater than both?	peaks
1	3, 7, 4	yes	1
2	7, 4, 8	no	1
3	4, 8, 8	no (8 > 8 false)	1
4	8, 8, 2	no (8 > 8 false)	1
5	8, 2, 5	no	1

The loop runs from index 1 to index 5, because the first and last elements do not have two neighbors; this keeps both `heights[i - 1]` and `heights[i + 1]` inside the array. Neither 8 counts, because each is equal to, not greater than, the other. If you got 2 or 3, you counted an 8.

Reread 4.5.5 *Consecutive Pairs* and 4.5.3 *Counting Elements with a Property* if your answer differed.

Question 20

B. This is the maximum algorithm on `String` elements. From Topic 1.15, `a.compareTo(b)` is positive when `a` comes after `b` in alphabetical order. So `pick` is replaced whenever a name comes later than the best so far: it starts at "Maya", skips "Leo", changes to "Zoe", and then nothing comes after "Zoe". The result is the name that comes last alphabetically.

- **A** is wrong: "Ana" comes first alphabetically; it would be the answer with `< 0`, the minimum.
- **C** is wrong: "Maya" is only the starting value; "Zoe" replaces it.
- **D** is wrong: "Kai" is just the last element. The loop compares, not just copies.

Reread 4.5.2 *Minimum and Maximum* if you chose differently.

Question 21

Model answer: The loop copies each element into the spot to its right, and every copy overwrites what was there. The last element is overwritten on the very first pass (by the element before it), so if it were not saved first, it would be lost and could not wrap around to index 0. The loop must go backward, from the end toward the front, so that each element is copied out of its spot before that spot is overwritten. Going forward would copy the first element into index 1, then copy that same value into index 2, and so on, filling the array with copies of the first element.

A complete answer says:

- Copies overwrite values, so the element that falls off must be saved before the loop, or it is lost.
- A right move must loop from right to left, or the first value is copied over and over.

Reread 4.5.7 *Shifting and Rotating* if your reasoning differed.

Question 22

```
2 7 0 0
9
```

`dropFirst` shifts left and fills the end with 0. Its parameter `list` receives a copy of the reference, so it changes the same array that `orders` refers to (Topic 3.6).

After	orders
start	4, 9, 2, 7
first <code>dropFirst</code>	9, 2, 7, 0
second <code>dropFirst</code>	2, 7, 0, 0

The final loop prints each element and totals them: $2 + 7 + 0 + 0$ is 9. If you got 4 9 2 7 and 22, you assumed a `void` method cannot change the caller's array; it cannot change which array `orders` refers to, but it can change the elements.

Reread 4.5.7 *Shifting and Rotating* and 4.5.1 *Sum and Average of an Array* if your answer differed.

Question 23

B. The first loop sums the readings to 88.0, and $88.0 / 4$ is 22.0. The second loop is an "all" check: the flag starts `true` and becomes `false` if any reading is more than 4.0 away from the average. The distances are 2.0, 0.5, 3.0, and 4.5; the last one is more than 4.0, so `allClose` ends `false`.

- **A** is wrong: three readings are close, but "all" fails as soon as one reading is too far away.
- **C** is wrong: 88.0 is the sum; the code prints `average`, the sum divided by the length.
- **D** is wrong: `allClose` is a boolean, so it prints `true` or `false`; 4.5 is only the distance of the last reading.

Reread 4.5.4 *At Least One or All* and 4.5.1 *Sum and Average of an Array* if you chose differently.