

4.4 Array Traversals

Self-Study Guide · About 60 minutes · With answer key

By the end you will be able to

- Use a for loop and the length of an array to visit every element by its index, and avoid going one step past the end.
- Traverse part of an array, traverse it backward, or skip elements by changing the loop's start, test, and update.
- Traverse an array with a while loop that uses an index variable.
- Read and trace an enhanced for loop, and explain what its loop variable holds on each pass.
- Explain why assigning to the enhanced for loop variable never changes the array, while calling a mutator on it can change an object in the array.
- Rewrite an enhanced for loop as an indexed for loop or a while loop, and choose the right kind of loop for a task.

What to remember from earlier

- An array holds a fixed number of values of one type, with indices 0 through length minus 1. ([Topic 4.3: Array Creation and Access](#))
- An index outside 0 to `arr.length - 1` throws an `ArrayIndexOutOfBoundsException`. ([Topic 4.3: Array Creation and Access, 4.3.6 Valid Indexes and Out-of-Bounds Errors](#))
- A for loop puts the setup, test, and update in one header. ([Topic 2.8: for Loops, 2.8.1 The Three Parts of a for Loop Header](#))
- A loop that runs one time too many or too few has an off-by-one error. ([Topic 2.7: while Loops, 2.7.5 Off-by-One Errors](#))
- With a `&& b`, Java skips `b` when `a` is false. ([Topic 2.5: Compound Boolean Expressions, 2.5.5 Short-Circuit Evaluation](#))

4.4.1 Visiting Every Element with a for Loop

A weather app stores a week of daily high temperatures in an array. To show them, it has to print every one. With four temperatures you could write `temps[0]`, `temps[1]`, `temps[2]`, and `temps[3]` on four lines, but a year of temperatures would need 365 nearly identical lines, and the code would break the moment the array had a different length.

Look at what changes from one of those lines to the next: only the index, which goes 0, 1, 2, 3. From Topic 2.8 you know a `for` loop is built for exactly that, a counter that starts somewhere, grows by 1, and stops at a limit. If the counter is used as the index, one loop body can reach every element in turn.

Using a loop to reach the elements of an array one after another is called **traversing** the array, and one complete run through it is a **traversal**. A traversal can visit every element, or only some of them in a planned order (that is 4.4.2). When the loop counter is the index, as here, the loop is called an **indexed loop**.

The standard header for visiting every element of an array named `temps` is:

```
1 for (int i = 0; i < temps.length; i++)
```

Piece	What it does	Why it is written this way
<code>int i = 0</code>	Starts the index at the first element	The first index of every array is 0 (Topic 4.3)
<code>i < temps.length</code>	Keeps looping while <code>i</code> is a valid index	The last valid index is <code>temps.length - 1</code> , so <code>i</code> must stay less than the length
<code>i++</code>	Moves to the next index	Neighboring elements have indexes that differ by 1
<code>temps[i]</code> in body	Reads (or changes) the element at the current index	The body works on one element per pass

Because the test uses `temps.length` instead of a typed number like `4`, the same loop works for an array of any length.

KEY IDEA

An indexed loop reaches each element through its index. To visit every element, start at 0, continue while the index is less than the array's length, and add 1 each pass.

WORKED EXAMPLE Printing every temperature

Watch how `i` does two jobs: it picks the element, and it is also printed as the day number.

```
1 int[] temps = {68, 71, 75, 70};
2 for (int i = 0; i < temps.length; i++)
3 {
4     System.out.println("Day " + i + ": " + temps[i]);
5 }
```

`temps.length` is 4, so the test on line 2 is `i < 4`.

<code>i</code>	<code>i < 4?</code>	<code>temps[i]</code>	Printed
0	true	68	Day 0: 68
1	true	71	Day 1: 71
2	true	75	Day 2: 75
3	true	70	Day 3: 70
4	false		(loop ends)

The body ran 4 times, once per element. When `i` became 4 the test was false, so `temps[4]`, which does not exist, was never touched.

```
Day 0: 68
Day 1: 71
Day 2: 75
Day 3: 70
```

An indexed loop can also **change** elements, because `points[i]` on the left of `=` stores into the array (Topic 4.3). This program doubles every score, then uses a second traversal to print them.

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

On line 4, the right side reads the current element and the `=` stores the doubled value back into the same slot. After the first loop the array holds `{6, 10, 16}`, and the second loop prints:

```
6
10
16
```

Remember this example; 4.4.5 shows a different loop that looks like it does the same thing but does not.

WATCH OUT

Using `<=` instead of `<`. Students often think "the loop should go up to the length," and write this:

```
1 int[] temps = {68, 71, 75, 70};
2 for (int i = 0; i <= temps.length; i++)
3 {
4     System.out.println(temps[i]);
5 }
```

The first four passes print all four temperatures. Then `i` becomes 4, the test `4 <= 4` is still true, and line 4 asks for `temps[4]`. The valid indexes are 0 through 3, so the program crashes. The console shows something like this (the wording can vary between Java versions):

```
68
71
75
70
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 4 out of
bounds for length 4
```

This is an off by one error (Topic 2.7) that the compiler cannot catch, because the code is legal; it fails only when it runs. The fix is `i < temps.length`. If you prefer to think "up to the last index," write `i <= temps.length - 1`, which means the same thing.

PAUSE AND TRY

Question 1 Tracing

Write the exact output.

```
1 String[] words = {"map", "tree", "go"};
2 for (int k = 0; k < words.length; k++)
3 {
4     System.out.println(k + " " + words[k].length());
5 }
```

Question 2 Multiple choice

What happens with this program?

LengthParens.java

```
1 public class LengthParens
2 {
3     public static void main(String[] args)
4     {
5         int[] scores = {90, 85, 77};
6         for (int i = 0; i < scores.length(); i++)
7         {
8             System.out.println(scores[i]);
9         }
10    }
11 }
```

- A. It prints 90, 85, and 77
- B. It prints 90, 85, and 77, then crashes with an `ArrayIndexOutOfBoundsException`
- C. It does not compile, because `length` for an array is not a method
- D. It prints nothing, because the loop test is false at the start

4.4.2 Other Starting Points, Directions, and Steps

A bus app lists the stops on a route. Riders going the other way want the list from the last stop to the first. A coach wants to call every other runner in a lineup. A track timer wants every lap except the first and last. None of these visits the elements from index 0 to the end one at a time, yet each still visits elements in a planned order.

The indexed `for` loop from 4.4.1 handles all of them, because each of its three parts can change, just as they did in Topic 2.8. The rule that never changes is that every index the body uses must be between 0 and `length - 1`.

To do this	Start	Test	Update
------------	-------	------	--------

Visit every element, forward	0	<code>i < arr.length</code>	<code>i++</code>
Visit every element, backward	<code>arr.length - 1</code>	<code>i >= 0</code>	<code>i--</code>
Visit every other element	0	<code>i < arr.length</code>	<code>i += 2</code>
Skip the first and last	1	<code>i < arr.length - 1</code>	<code>i++</code>

KEY IDEA

Changing where an indexed loop starts, when it stops, and how it steps lets you visit part of an array, go backward, or skip elements, as long as every index used stays from 0 to length minus 1.

WORKED EXAMPLE Stops in reverse

The loop starts at the **last** index and counts down. Look at the start value and the test.

```
1 String[] stops = {"Elm", "Oak", "Pine", "Main"};
2 for (int i = stops.length - 1; i >= 0; i--)
3 {
4     System.out.println(stops[i]);
5 }
```

`stops.length` is 4, so the loop starts at `i = 3`, the last valid index.

i	i >= 0?	stops[i]
3	true	"Main"
2	true	"Pine"
1	true	"Oak"
0	true	"Elm"
-1	false	(loop ends)

The test is `>= 0`, not `> 0`, because index 0 holds a real element that must be printed.

```
Main
Pine
Oak
Elm
```

Now change only the update. With `i += 2`, the loop jumps two indexes at a time:

```
1 String[] lineup = {"Ana", "Ben", "Cy", "Dee", "Eli"};
2 for (int i = 0; i < lineup.length; i += 2)
3 {
4     System.out.println(lineup[i]);
5 }
```

`i` takes the values 0, 2, and 4, then becomes 6, and `6 < 5` is false. The program prints:

Ana
Cy
Eli

The test `i < lineup.length` still protects the array. If the lineup had 4 names instead of 5, `i` would go 0, 2, then 4, and `4 < 4` would stop the loop before reaching a bad index.

WATCH OUT

Starting a backward loop at `length`. It is easy to write the start as "the length" when counting down:

```
1 String[] stops = {"Elm", "Oak", "Pine", "Main"};
2 for (int i = stops.length; i >= 0; i--)
3 {
4     System.out.println(stops[i]);
5 }
```

On the very first pass, `i` is 4 and line 4 asks for `stops[4]`. Nothing prints at all; the program crashes right away with an `ArrayIndexOutOfBoundsException` saying index 4 is out of bounds for length 4. The length counts the elements, but the last **index** is one less. Start at `stops.length - 1`.

PAUSE AND TRY

Question 3 Tracing

Write the exact output.

```
1 int[] codes = {4, 9, 2, 7, 5, 1, 8};
2 for (int i = codes.length - 1; i > 0; i -= 3)
3 {
4     System.out.println(codes[i]);
5 }
```

Question 4 Multiple choice

`laps` holds six lap times. What does this code print?

```
1 int[] laps = {52, 49, 47, 48, 50, 55};
2 for (int i = 1; i < laps.length - 1; i++)
3 {
4     System.out.println("Lap " + i + ": " + laps[i]);
5 }
```

- A. Lap 1: 49 through Lap 4: 50, four lines
- B. Lap 1: 49 through Lap 5: 55, five lines
- C. Lap 0: 52 through Lap 4: 50, five lines
- D. Lap 1: 49 through Lap 5: 55, then an `ArrayIndexOutOfBoundsException`

4.4.3 Traversing with a while Loop

Every traversal so far has used a `for` loop. In Topic 2.8 you learned that any `for` loop can be written as a `while` loop with the same three parts placed on separate lines. The same is true for traversals, and sometimes a `while` loop reads more naturally, for example when the loop should also stop early for a reason other than reaching the end.

A `while` traversal is still an indexed traversal: you need your own index variable, declared before the loop, tested in the condition, used inside the brackets, and updated at the end of the body.

```
1 int i = 0;           // start
2 while (i < temps.length) // test
3 {
4     // use temps[i]
5     i++;             // update
6 }
```

KEY IDEA

A while loop traverses an array the same way an indexed `for` loop does: declare an index before the loop, test it against the length, use it in brackets, and update it at the end of the body.

WORKED EXAMPLE The temperature report as a while loop

This produces exactly the same output as `PrintTemps` in 4.4.1. Match each part to the `for` header.

```
1 int[] temps = {68, 71, 75, 70};
2 int i = 0;
3 while (i < temps.length)
4 {
5     System.out.println("Day " + i + ": " + temps[i]);
6     i++;
7 }
```

for header part	Where it is in the while version
<code>int i = 0</code>	Line 2, before the loop
<code>i < temps.length</code>	Line 3, the while condition
<code>i++</code>	Line 6, the last line of the body

The trace is the same as before: `i` goes 0, 1, 2, 3, the body prints one line each time, and when `i` reaches 4 the condition is false.

```
Day 0: 68
Day 1: 71
Day 2: 75
Day 3: 70
```

Now vary the condition. A to-do list ends with a marker "done", and anything after the marker should be ignored. The `while` condition can check both "still inside the array" and "not at the marker yet":

```
1 String[] tasks = {"wash", "dry", "done", "fold"};
2 int i = 0;
3 while (i < tasks.length && !tasks[i].equals("done"))
4 {
5     System.out.println(tasks[i]);
6     i++;
7 }
8 System.out.println("Stopped at index " + i);
```

```
wash
dry
Stopped at index 2
```

i	i < 4?	tasks[i]	!tasks[i].equals("done")?	Loop runs?
0	true	"wash"	true	yes
1	true	"dry"	true	yes
2	true	"done"	false	no

This traversal visits an ordered part of the array, from the start up to the marker. Because `i` is declared before the loop, it still exists afterward, so line 8 can report where the loop stopped.

The **order** of the two tests matters. If the array has no "done" at all, `i` eventually reaches the length. Then `i < tasks.length` is false, and by short-circuit evaluation (Topic 2.5) Java never evaluates `tasks[i]`. Swap the two tests and that protection disappears:

```
1 String[] tasks = {"wash", "dry"};
2 int i = 0;
3 while (!tasks[i].equals("done") && i < tasks.length)
4 {
5     System.out.println(tasks[i]);
6     i++;
7 }
8 System.out.println("Stopped at index " + i);
```

This prints `wash` and `dry`, then crashes with an `ArrayIndexOutOfBoundsException` for index 2, because when `i` is 2 Java evaluates `tasks[2]` first. With the tests in the original order, the same two-element array prints `wash`, `dry`, and `Stopped at index 2`.

WATCH OUT

Forgetting the update. In a `for` loop the update sits in the header where it is hard to forget. In a `while` loop it is a separate statement at the bottom of the body:

```

1 int i = 0;
2 while (i < temps.length)
3 {
4     System.out.println(temps[i]);
5 }

```

`i` stays 0 forever, so the condition is always true and the loop never ends; it keeps printing the first temperature until someone stops the program. Add `i++` as the last statement of the body.

PAUSE AND TRY

Question 5 Tracing

Write the exact output.

```

1 int[] seats = {12, 15, 9, 20, 11};
2 int n = seats.length - 1;
3 while (n >= 0)
4 {
5     System.out.println(seats[n]);
6     n -= 2;
7 }

```

Question 6 Short answer

In `UntilDone`, the condition is `i < tasks.length && !tasks[i].equals("done")`. Explain why the test `i < tasks.length` has to come first.

4.4.4 The Enhanced for Loop

Look back at `PrintTemps` without the day numbers. If all you want is each temperature, the index variable is just bookkeeping: you set it up, test it, add 1 to it, and use it in brackets, all to get at values you could describe simply as "each temperature in the array." Java has a shorter loop for exactly that situation.

The **enhanced for loop** (sometimes called a for-each loop) visits every element of an array from first to last without any index. Its header names a variable, and on each pass that variable is given a copy of the next element.

```

1 for (int t : temps)

```

Read it aloud as "for each `int t` in `temps`."

Piece	What it is
<code>int</code>	The variable's type, which must match the type of the array's elements
<code>t</code>	The enhanced for loop variable ; it holds a copy of one element during each pass
<code>:</code>	Separates the variable from the array; read it as "in"

temps	The array to traverse
-------	-----------------------

Behind the scenes Java still walks through the indexes 0, 1, 2, and so on, but you never see them. That means three things the indexed loop could do are impossible here: the loop always starts at the first element, always moves forward one element at a time, and never tells you the current index.

KEY IDEA

An enhanced for loop gives its loop variable a copy of each element in order, from first to last, with no index involved.

WORKED EXAMPLE Each temperature, no index

Compare this to `PrintTemps`: there is no `i`, no `.length`, and no brackets.

```
1 int[] temps = {68, 71, 75, 70};
2 for (int t : temps)
3 {
4     System.out.println(t);
5 }
```

Pass	t holds a copy of	Printed
1	temps[0], 68	68
2	temps[1], 71	71
3	temps[2], 75	75
4	temps[3], 70	70

After the fourth element there are no more, so the loop ends on its own. An enhanced for loop cannot run past the end of the array, because it never uses an index you could get wrong.

```
68
71
75
70
```

The element type does not have to be `int`. For an array of `String`, the loop variable is a `String`, so you can call `String` methods on it:

```
1 String[] guests = {"Rosa", "Kenji", "Amara"};
2 for (String guest : guests)
3 {
4     System.out.println(guest + " has " + guest.length() + " letters");
5 }
```

```
Rosa has 4 letters
Kenji has 5 letters
Amara has 5 letters
```

Each pass, `guest` refers to the next name, and `guest.length()` is that name's number of letters.

WATCH OUT

Treating the loop variable as an index. Students used to indexed loops sometimes write `temps[t]`:

```
1 int[] temps = {68, 71, 75, 70};
2 for (int t : temps)
3 {
4     System.out.println(temps[t]);
5 }
```

On the first pass `t` is 68, a temperature, not a position, so line 4 asks for `temps[68]`. Nothing prints; the program crashes with an `ArrayIndexOutOfBoundsException` for index 68. In an enhanced for loop the variable already **is** the element. Print `t` itself.

PAUSE AND TRY**Question 7** Tracing

This array happens to hold only values that are also valid indexes. Write the exact output.

```
1 int[] nums = {2, 0, 3, 1};
2 for (int value : nums)
3 {
4     System.out.println(nums[value]);
5 }
```

Question 8 Multiple choice

What happens with this program?

`WrongVarType.java`

```
1 public class WrongVarType
2 {
3     public static void main(String[] args)
4     {
5         int[] temps = {68, 71, 75, 70};
6         for (String t : temps)
7         {
8             System.out.println(t);
9         }
10    }
11 }
```

- A. It prints 68, 71, 75, and 70
- B. It does not compile, because the loop variable's type does not match the array's element type
- C. It compiles but crashes, because an `int` cannot be stored in a `String` while the program runs
- D. It does not compile, because an enhanced for loop cannot be used on an `int` array

4.4.5 Assigning to the Loop Variable

In 4.4.1, an indexed loop doubled every score with `points[i] = points[i] * 2`. The enhanced for loop is shorter, so it is tempting to write the same job as "for each `p` in `points`, set `p` to `p * 2`." Does that change the array?

The key word in 4.4.4 was **copy**. On each pass the loop variable gets a copy of the element, the same way a parameter gets a copy of its argument (Topic 3.5). Changing the copy does not reach back to the original. Think of copying a phone number from a contact list onto a sticky note: if you scribble a new number on the note, the contact list does not change.

KEY IDEA

Assigning a new value to the enhanced for loop variable changes only the variable. The value stored in the array stays the same.

WORKED EXAMPLE Doubling that does not stick

The first loop assigns to `p`. The second loop shows what the array actually holds.

```
1 int[] points = {3, 5, 8};
2 for (int p : points)
3 {
4     p = p * 2;
5     System.out.println("p is now " + p);
6 }
7 for (int p : points)
8 {
9     System.out.println("array holds " + p);
10 }
```

Pass	Copied into <code>p</code>	<code>p</code> after line 4	<code>points</code> array
1	3	6	{3, 5, 8}
2	5	10	{3, 5, 8}
3	8	16	{3, 5, 8}

Line 4 really does double `p`, and line 5 proves it. But `p` is a separate variable that held a copy, so the array column never changes. The second loop starts fresh, copies the original values, and prints them.

```
p is now 6
p is now 10
p is now 16
array holds 3
array holds 5
array holds 8
```

The same thing happens with an array of `String`. Here the loop tries to make each name uppercase:

```
1 String[] guests = {"Rosa", "Kenji"};
2 for (String guest : guests)
3 {
4     guest = guest.toUpperCase();
5 }
6 System.out.println(guests[0] + " " + guests[1]);
```

Rosa Kenji

`guest.toUpperCase()` builds a **new** string, "ROSA", and line 4 makes `guest` refer to it. The array element still refers to the original "Rosa". As with a `String` parameter in Topic 3.6, assigning to the variable moves only the variable.

WATCH OUT

Using an enhanced for loop to change array elements. `EnhancedAssign` and `ShoutNames` compile and run with no error message, which makes the bug hard to spot: the array just quietly stays the same. To store new values in an array, you need the index, so use an indexed loop and assign to the element:

```
1 for (int i = 0; i < guests.length; i++)
2 {
3     guests[i] = guests[i].toUpperCase();
4 }
```

With this loop in place of lines 2 to 5, the program prints ROSA KENJI. For the scores, `DoublePoints` in 4.4.1 is the correct version.

PAUSE AND TRY

Question 9 Tracing

Write the exact output.

```
1 int[] quiz = {7, 9, 6};
2 for (int q : quiz)
3 {
4     q += 1;
5     System.out.println(q);
6 }
7 System.out.println(quiz[0] + " " + quiz[1] + " " + quiz[2]);
```

Question 10 Multiple choice

`guests` is a `String` array holding "Rosa" and "Kenji". After this loop runs, what does `guests` hold?

```

1 for (String g : guests)
2 {
3     g = g + "!";
4 }

```

- A. "Rosa!" and "Kenji!"
- B. "Rosa" and "Kenji"
- C. "Rosa" and "Kenji!", because only the last assignment is kept
- D. Nothing; the loop does not compile because `g` cannot be assigned

4.4.6 Enhanced for Loops and Objects

A game keeps its players in an array of `Player` objects. At the end of a round, every player earns a 5-point bonus. From 4.4.5 you might expect an enhanced for loop to be useless for this, since it only hands out copies. But what exactly is being copied when the elements are objects?

From Topic 3.6, a variable of a class type holds a **reference**, the address of an object. An array of `Player` holds references too. So when the enhanced for loop copies an element into its variable, it copies the address, not the object. The loop variable becomes an alias of the object the array element refers to, exactly like a parameter that receives an object. That leads to two different outcomes:

Statement in the loop body	What it changes	Does the array see it?
<code>p.addPoints(5);</code>	The object that <code>p</code> and the element both refer to	Yes: the element still refers to that same, now-changed object
<code>p = new Player("Rookie", 0);</code>	Only <code>p</code> , which now refers to a different object	No: the element still holds its original reference

In the sticky-note picture, the note now has a house address on it. Going to the house and painting the door changes the house that the array's address also leads to. Writing a different address on the note changes only the note.

KEY IDEA

When an array holds objects, calling a mutator on the enhanced for loop variable changes the object the element refers to. It never changes which object each element refers to.

The programs in this subsection use a small `Player` class with a name, a point total, two accessors, and the mutator `addPoints`. 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 A bonus for every player

Lines 1 to 26 are the `Player` class. Watch line 35, which calls a mutator through the loop variable.

TeamBonus.java

```

1  class Player
2  {
3      private String name;
4      private int points;
5
6      public Player(String playerName, int startPoints)
7      {
8          name = playerName;
9          points = startPoints;
10     }
11
12     public String getName()
13     {
14         return name;
15     }
16
17     public int getPoints()
18     {
19         return points;
20     }
21
22     public void addPoints(int amount)
23     {
24         points += amount;
25     }
26 }
27
28 public class TeamBonus
29 {
30     public static void main(String[] args)
31     {
32         Player[] team = {new Player("Ada", 10), new Player("Bo", 7)};
33         for (Player p : team)
34         {
35             p.addPoints(5);
36         }
37         for (Player p : team)
38         {
39             System.out.println(p.getName() + " " + p.getPoints());
40         }
41     }
42 }

```

Line 32 builds two `Player` objects. Call their addresses A and B; `team[0]` holds A and `team[1]` holds B.

Pass	p holds a copy of	Line 35 does	Ada's object	Bo's object
1	address A	adds 5 to the object at A	15	7
2	address B	adds 5 to the object at B	15	12

The array still holds addresses A and B, unchanged, but the objects at those addresses have new point totals. The second loop reads them:

Ada 15
Bo 12

Now vary the body: instead of calling a mutator, assign a new object to `p`.

TeamReplace.java

```
1 class Player
2 {
3     private String name;
4     private int points;
5
6     public Player(String playerName, int startPoints)
7     {
8         name = playerName;
9         points = startPoints;
10    }
11
12    public String getName()
13    {
14        return name;
15    }
16
17    public int getPoints()
18    {
19        return points;
20    }
21
22    public void addPoints(int amount)
23    {
24        points += amount;
25    }
26 }
27
28 public class TeamReplace
29 {
30     public static void main(String[] args)
31     {
32         Player[] team = {new Player("Ada", 10), new Player("Bo", 7)};
33         for (Player p : team)
34         {
35             p = new Player("Rookie", 0);
36         }
37         for (Player p : team)
38         {
39             System.out.println(p.getName() + " " + p.getPoints());
40         }
41     }
42 }
```

Ada 10
Bo 7

Each pass, line 35 builds a new rookie and moves `p` to it. That changes only `p.team[0]` and `team[1]` still hold addresses A and B, and nothing changed those objects, so Ada and Bo keep their original points. The two rookies are lost when the loop moves on, because nothing holds their addresses.

WATCH OUT

Expecting the enhanced for loop to replace objects in the array. To put a different object into each slot, you have to assign to the element itself, which needs an index:

```
1 for (int i = 0; i < team.length; i++)
2 {
3     team[i] = new Player("Rookie", 0);
4 }
```

With this loop in place of lines 33 to 36, the program prints `Rookie 0` twice. Use the enhanced for loop to **use or update** each object; use an indexed loop to **replace** elements.

PAUSE AND TRY

Question 11 Tracing

Write the exact output. (`Player` is the same class as in `TeamBonus`.)

MixedLoop.java

```
1 class Player
2 {
3     private String name;
4     private int points;
5
6     public Player(String playerName, int startPoints)
7     {
8         name = playerName;
9         points = startPoints;
10    }
11
12    public String getName()
13    {
14        return name;
15    }
16
17    public int getPoints()
18    {
19        return points;
20    }
21
22    public void addPoints(int amount)
23    {
24        points += amount;
25    }
26 }
27
28 public class MixedLoop
```

```
29 {
30     public static void main(String[] args)
31     {
32         Player[] team = {new Player("Kai", 4), new Player("Lee", 6)};
33         for (Player p : team)
34         {
35             p.addPoints(2);
36             p = new Player("Temp", 100);
37             p.addPoints(2);
38         }
39         System.out.println(team[0].getName() + " " + team[0].getPoints());
40         System.out.println(team[1].getName() + " " + team[1].getPoints());
41     }
42 }
```

4.4.7 Choosing and Rewriting a Traversal

You now have three ways to traverse an array: an indexed `for` loop, a `while` loop, and an enhanced `for` loop. How do you choose? And if a program uses one kind, can you always switch to another?

Switching from an enhanced `for` loop to an indexed loop always works. The enhanced loop visits every element from first to last, and an indexed loop from 0 while `i < arr.length` does exactly the same. The trick is to recreate the loop variable: inside the body, wherever the enhanced loop used its variable, use `arr[i]`. The easiest way is to declare a variable with the same name and set it to `arr[i]` as the first line of the body; then the rest of the body does not change at all.

KEY IDEA

Any enhanced `for` loop over an array can be rewritten as an indexed `for` loop or a `while` loop that visits indexes 0 through length minus 1 and uses `arr[i]` wherever the loop variable appeared.

WORKED EXAMPLE One traversal, three ways

All three loops print every color in order. Notice line 10 and line 18, which recreate `c` so the print statements stay the same.

```

1 String[] colors = {"red", "teal", "gold"};
2
3 for (String c : colors)
4 {
5     System.out.println("enhanced: " + c);
6 }
7
8 for (int i = 0; i < colors.length; i++)
9 {
10    String c = colors[i];
11    System.out.println("indexed: " + c);
12 }
13
14 int j = 0;
15 while (j < colors.length)
16 {
17    String c = colors[j];
18    System.out.println("while: " + c);
19    j++;
20 }

```

Loop	Where c comes from	Lines
Enhanced	The loop gives c a copy of each element	3–6
Indexed	String c = colors[i]; copies element i	8–12
while	String c = colors[j]; copies element j	14–20

Each `c` inside a loop body is a separate local variable that exists only during its own loop, so reusing the name is allowed.

```

enhanced: red
enhanced: teal
enhanced: gold
indexed: red
indexed: teal
indexed: gold
while: red
while: teal
while: gold

```

The other direction does not always work. An enhanced for loop cannot replace an indexed loop that needs the index. Use this to choose:

The task needs to...	Use	Why
Read or use every element, first to last	Enhanced for (simplest)	No index to get wrong
Know the position (for example, print "Day 2")	Indexed for or while	The enhanced loop never reveals the index
Store new values or new objects in the array	Indexed for or while	Only <code>arr[i] = ...</code> changes an element

Go backward, skip elements, or visit only part	Indexed <code>for</code> or <code>while</code>	The enhanced loop always goes forward through all
Stop when a condition is met before the end	<code>while</code> with two tests	The condition can check the index and the value

Calling a mutator on objects in the array works with any of the three, as 4.4.6 showed.

WATCH OUT

Rewriting with the index instead of the element. A student rewrites `for (int age : ages) { System.out.println(age); }` as an indexed loop but prints the counter:

```
1 int[] ages = {15, 17, 16};
2 for (int i = 0; i < ages.length; i++)
3 {
4     System.out.println(i);
5 }
```

```
0
1
2
```

It prints the positions 0, 1, and 2 instead of the ages 15, 17, and 16. In an indexed loop, `i` is a position; the element is `ages[i]`. Printing `ages[i]` fixes it.

PAUSE AND TRY

Question 12 Short answer

For each task on an array `prices`, say whether an enhanced for loop can do it, and explain why or why not.

1. Print every price, first to last.
2. Add 1 to every price stored in the array.
3. Print the prices from last to first.

Question 13 Debugging

`RewriteWrong` was supposed to print the ages 15, 17, and 16 but printed 0, 1, and 2. Correct line 4, and explain the error.

Putting it together

This program uses every kind of traversal from the chapter on one array of `Player` objects (the same `Player` class as in 4.4.6). Before reading the explanation, predict the output.

RosterDay.java

```
1 class Player
2 {
```

```
3     private String name;
4     private int points;
5
6     public Player(String playerName, int startPoints)
7     {
8         name = playerName;
9         points = startPoints;
10    }
11
12    public String getName()
13    {
14        return name;
15    }
16
17    public int getPoints()
18    {
19        return points;
20    }
21
22    public void addPoints(int amount)
23    {
24        points += amount;
25    }
26 }
27
28 public class RosterDay
29 {
30     public static void main(String[] args)
31     {
32         Player[] roster = {new Player("Ada", 12), new Player("Bo", 9), new
Player("Cy", 15)};
33
34         for (Player p : roster)
35         {
36             p.addPoints(3);
37         }
38
39         roster[1] = new Player("Dee", 0);
40
41         for (int i = 0; i < roster.length; i++)
42         {
43             System.out.println((i + 1) + ". " + roster[i].getName() + " " +
roster[i].getPoints());
44         }
45
46         int k = roster.length - 1;
47         while (k >= 0)
48         {
49             System.out.println(roster[k].getName());
50             k--;
51         }
52     }
53 }
```

Call the three original objects A (Ada, 12), B (Bo, 9), and C (Cy, 15).

Lines	What happens	roster holds	Points
32	An initializer list builds the array	A, B, C	12, 9, 15
34–37	Enhanced for: <code>p</code> is an alias of each object in turn, and the mutator adds 3 to each	A, B, C	15, 12, 18
39	Assigning to an element replaces Bo's reference with a new object D (Dee, 0)	A, D, C	15, 0, 18
41–44	Indexed for: <code>i</code> goes 0, 1, 2; the index is needed to print the place numbers <code>i + 1</code>	A, D, C	15, 0, 18
46–51	while loop, backward: <code>k</code> goes 2, 1, 0, printing names from last to first; at <code>k = -1</code> the condition is false	A, D, C	15, 0, 18

```

1. Ada 15
2. Dee 0
3. Cy 18
Cy
Dee
Ada

```

Each loop was chosen for its job. The bonus only needed each object, so the enhanced for loop was simplest; it worked because `addPoints` changes the objects, not the array. Replacing Bo needed an assignment to `roster[1]`. The numbered list needed the index, and printing in reverse needed an index that counts down. Bo's object, now with 12 points, is lost because no element refers to it anymore.

Vocabulary

Term	Definition	Example
traversing (an array)	Using a loop to reach the elements of an array one after another, either all of them or a planned part	Printing every temperature
traversal	One complete run of a loop through an array's elements	A backward pass over <code>stops</code>
indexed loop	A for or while loop whose counter is used as the index to reach each element	<code>for (int i = 0; i < a.length; i++)</code>
enhanced for loop	A loop that visits every element of an array from first to last without an index	<code>for (int t : temps)</code>
enhanced for loop variable	The variable in an enhanced for loop's header, which holds a copy of one element on each pass	<code>t in for (int t : temps)</code>

ArrayIndexOutOfBoundsException (review)	The runtime error caused by using an index below 0 or at least the array's length	<code>temps[4]</code> when the length is 4
--	---	--

Chapter summary

- Traversing an array means using a loop to reach its elements in order, either all of them or a planned part.
- An indexed `for` or `while` loop reaches elements through their indexes. To visit every element, go from 0 while the index is less than the length; any other start, test, or update is fine as long as every index stays between 0 and length minus 1, or the program throws an `ArrayIndexOutOfBoundsException`.
- An enhanced `for` loop gives its variable a copy of each element, first to last, with no index. Assigning to that variable never changes the array.
- When the elements are object references, the enhanced `for` loop variable is an alias of each object, so calling a mutator on it changes the object, but the references stored in the array stay the same.
- Any enhanced `for` loop can be rewritten as an indexed `for` or `while` loop using `arr[i]`; tasks that need positions, a different order, or changes to the elements themselves require an indexed loop.

Review questions

Question 14 Tracing

Write the exact output.

```

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

Question 15 Multiple choice

`Player` is the same class as in `TeamBonus`. What happens when this program runs?

`NullTeam.java`

```

1  class Player
2  {
3      private String name;
```

```
4     private int points;
5
6     public Player(String playerName, int startPoints)
7     {
8         name = playerName;
9         points = startPoints;
10    }
11
12    public String getName()
13    {
14        return name;
15    }
16
17    public int getPoints()
18    {
19        return points;
20    }
21
22    public void addPoints(int amount)
23    {
24        points += amount;
25    }
26 }
27
28 public class NullTeam
29 {
30     public static void main(String[] args)
31     {
32         Player[] team = new Player[3];
33         team[0] = new Player("Ada", 10);
34         for (Player p : team)
35         {
36             p.addPoints(1);
37             System.out.println(p.getName() + " " + p.getPoints());
38         }
39     }
40 }
```

- A. It prints Ada 11 and nothing else, because the loop skips empty elements
- B. It prints Ada 11, then crashes with a `NullPointerException`
- C. It prints Ada 11, null 1, and null 1
- D. It crashes before printing anything, with an `ArrayIndexOutOfBoundsException`

Question 16 Debugging

`resetAll` is supposed to set every team's score back to 0, but the program prints 4 0 7. Explain the error and write corrected code for lines 17 to 20.

ScoreboardReset.java

```
1 class Scoreboard
2 {
3     private int[] scores;
4
5     public Scoreboard(int teams)
```

```

6      {
7          scores = new int[teams];
8      }
9
10     public void addScore(int team, int amount)
11     {
12         scores[team] += amount;
13     }
14
15     public void resetAll()
16     {
17         for (int s : scores)
18         {
19             s = 0;
20         }
21     }
22
23     public int getScore(int team)
24     {
25         return scores[team];
26     }
27 }
28
29 public class ScoreboardReset
30 {
31     public static void main(String[] args)
32     {
33         Scoreboard board = new Scoreboard(3);
34         board.addScore(0, 4);
35         board.addScore(2, 7);
36         board.resetAll();
37         System.out.println(board.getScore(0) + " " + board.getScore(1) + " " +
board.getScore(2));
38     }
39 }

```

Question 17 Short answer

TeamBonus used an enhanced for loop to change every player's points, but ShoutNames could not use one to change every guest's name. Both arrays hold references. Explain why one works and the other does not.

Question 18 Multiple choice

Which `while` loop produces exactly the same output as this enhanced for loop?

```

1  double[] prices = {2.5, 4.0, 1.25};
2  for (double price : prices)
3  {
4      System.out.println(price);
5  }

```

- A. `int i = 0; while (i < prices.length) { System.out.println(prices[i]); i++; }`
- B. `int i = 0; while (i < prices.length) { i++; System.out.println(prices[i]); }`
- C. `int i = 0; while (i <= prices.length - 1) { System.out.println(prices[i]); }`

D. `int i = prices.length; while (i > 0) { System.out.println(prices[i - 1]); i--; }`

Question 19 Tracing

A sensor records readings until it sends -1 to say it has stopped. Write the exact output.

```
1 int[] readings = {5, 8, -1, 4};
2 int i = 0;
3 while (i < readings.length && readings[i] >= 0)
4 {
5     System.out.println(readings[i] * 2);
6     i++;
7 }
8 System.out.println("i = " + i);
```

Question 20 Multiple choice

data has length 7. How many times does the body of this loop run, and which indexes does it visit?

```
1 int[] data = {10, 20, 30, 40, 50, 60, 70};
2 for (int i = data.length - 2; i >= 1; i -= 2)
3 {
4     System.out.println(i + ": " + data[i]);
5 }
```

- A. 3 times: indexes 5, 3, and 1
- B. 4 times: indexes 6, 4, 2, and 0
- C. 3 times: indexes 6, 4, and 2
- D. 4 times: indexes 5, 3, 1, and then -1, which crashes

Reflection

- ☐ I can write the header of an indexed for loop that visits every element, and explain why the test uses less than the length. Reread: 4.4.1 Visiting Every Element with a for Loop
- ☐ I can trace a traversal that goes backward, skips elements, or visits only part of an array. Reread: 4.4.2 Other Starting Points, Directions, and Steps
- ☐ I can traverse an array with a while loop and explain why the bounds test comes first in a two-part condition. Reread: 4.4.3 Traversing with a while Loop
- ☐ I can read an enhanced for loop and say what its variable holds on each pass. Reread: 4.4.4 The Enhanced for Loop
- ☐ I can explain why assigning to the enhanced for loop variable does not change the array. Reread: 4.4.5 Assigning to the Loop Variable
- ☐ I can predict whether a change made through the loop variable shows up in an array of objects. Reread: 4.4.6 Enhanced for Loops and Objects
- ☐ I can rewrite an enhanced for loop as an indexed loop and choose the right loop for a task. Reread: 4.4.7 Choosing and Rewriting a Traversal

Check Your Reasoning

Answers and explanations for every question, numbered to match.

Question 1

```
0 3
1 4
2 2
```

`words.length` is 3, so `k` takes the values 0, 1, and 2. Each pass prints the index, a space, and the length of the string at that index.

k	words[k]	words[k].length()	Printed
0	"map"	3	0 3
1	"tree"	4	1 4
2	"go"	2	2 2

At `k = 3`, the test `3 < 3` is false and the loop ends. Notice the two kinds of `length`: `words.length` (no parentheses) is the array's length, and `words[k].length()` (with parentheses) is a `String` method.

Reread 4.4.1 *Visiting Every Element with a for Loop* if your output differed.

Question 2

C. An array's length is reached as `scores.length`, with no parentheses (Topic 4.3). Writing `scores.length()` asks for a method that arrays do not have, so Java reports something like error: cannot find symbol with symbol: method length() for line 6.

- **A** is wrong: it would be right with `scores.length`, but the program never gets to run.
- **B** is wrong: that is what `<=` would cause, and this test uses `<`. A compile error stops everything before any output.
- **D** is wrong: a program that does not compile never evaluates its loop test.

Students mix this up because `String` uses `length()` with parentheses. Arrays use `length` without them.

Reread 4.4.1 *Visiting Every Element with a for Loop* if you chose differently.

Question 3

```
8
7
```

`codes.length` is 7, so `i` starts at 6.

i	i > 0?	codes[i]	Printed
6	true	8	8
3	true	7	7
0	false		(loop ends)

The test is `i > 0`, so index 0 is **not** visited even though it is a valid index. That is fine here: the loop was written to stop before it.

Reread 4.4.2 *Other Starting Points, Directions, and Steps* if your output differed.

Question 4

A. `laps.length - 1` is 5, so the test is `i < 5` and `i` takes the values 1, 2, 3, and 4. The loop skips index 0 (it starts at 1) and index 5 (the test stops before it), so it prints Lap 1: 49, Lap 2: 47, Lap 3: 48, and Lap 4: 50.

- **B** is wrong: that would need the test `i < laps.length`. Subtracting 1 in the test removes the last element.
- **C** is wrong: the loop starts at `i = 1`, not 0.
- **D** is wrong: every index used, 1 through 4, is valid, so nothing crashes.

Reread 4.4.2 *Other Starting Points, Directions, and Steps* if you chose differently.

Question 5

```
11
9
12
```

`seats.length` is 5, so `n` starts at 4, the last index.

n	n >= 0?	seats[n]	Printed
4	true	11	11
2	true	9	9
0	true	12	12
-2	false		(loop ends)

This is a backward, every-other traversal written as a `while` loop. The update `n -= 2` is on line 6, inside the body.

Reread 4.4.3 *Traversing with a while Loop* if your output differed.

Question 6

Model answer: When the array has no "done" marker, `i` eventually equals `tasks.length`, which is not a valid index. With `i < tasks.length` first, that test is false, and `&&` short-circuits, so Java never evaluates `tasks[i]`. If the marker check came first, Java would evaluate `tasks[i]` with `i` equal to the length and throw an `ArrayIndexOutOfBoundsException`, as `GuardOrderWrong` did.

A complete answer mentions:

- When `i` reaches the length, `tasks[i]` is out of bounds.
- `&&` skips its right side when the left side is false, so the bounds test protects the array access only if it is on the left.

Reread 4.4.3 *Traversing with a while Loop* if your answer differed.

Question 7

```
3
2
1
0
```

`value` holds each element in turn, and line 4 uses it as an index.

Pass	value	nums[value]
1	2	3
2	0	2
3	3	1
4	1	0

It does not crash only because every element is between 0 and 3. This is the same mistake as in the Watch Out: `value` is an element, not a position, so the program prints the wrong things even though it runs.

Reread 4.4.4 *The Enhanced for Loop* if your output differed.

Question 8

B. The loop variable receives each element, and the elements of `temps` are `int` values. A `String` variable cannot hold an `int`, so Java reports something like

error: incompatible types: `int` cannot be converted to `String` for line 6. The fix is `for (int t : temps).`

- **A** is wrong: the program never runs.
- **C** is wrong: Java checks the types when it compiles, so this is a compile error, not a runtime crash.
- **D** is wrong: enhanced for loops work on arrays of any type, including `int`, as `EnhancedTemps` showed.

Reread 4.4.4 *The Enhanced for Loop* if you chose differently.

Question 9

```
8
10
7
7 9 6
```

Pass	Copied into q	q after q += 1	Printed
1	7	8	8
2	9	10	10
3	6	7	7

Inside the loop, `q` really is 1 larger, so lines 4 and 5 print 8, 10, and 7. But `q += 1` changes only the copy. Line 7 reads the array itself, which still holds `7 9 6`.

Reread 4.4.5 *Assigning to the Loop Variable* if your output differed.

Question 10

B. Each pass, `g` gets a copy of the reference stored in the array. `g + "!"` builds a new string, and the assignment makes `g` refer to it. The array's elements are never assigned, so they still refer to "Rosa" and "Kenji". This is the same behavior as `ShoutNames`.

- **A** is wrong: that would require assigning to `guests[i]` in an indexed loop. Assigning to the loop variable does not store into the array.
- **C** is wrong: no assignment reaches the array, including the last one.
- **D** is wrong: the loop variable can be assigned; Java allows it. It just has no effect on the array.

Reread 4.4.5 *Assigning to the Loop Variable* if you chose differently.

Question 11

```
Kai 6
Lee 8
```

`team[0]` holds address A (Kai, 4) and `team[1]` holds address B (Lee, 6).

Pass	Line	p refers to	Kai	Lee	Temp object
1	35	A (Kai)	6	6	
1	36	new Temp object	6	6	100
1	37	Temp object	6	6	102
2	35	B (Lee)	6	8	
2	36	new Temp object	6	8	100
2	37	Temp object	6	8	102

Line 35 runs while `p` is still an alias of the array's object, so Kai and Lee each gain 2. Line 36 moves `p` to a brand-new object, so line 37 adds to that object instead. The array's references never change, so lines 39 and 40 print Kai with 6 and Lee with 8.

Reread 4.4.6 *Enhanced for Loops and Objects* if your output differed.

Question 12

Model answer:

1. Yes. It only needs each element in order, which is exactly what an enhanced for loop provides.
2. No. The loop variable holds a copy of each price, so assigning to it does not change the array. Storing a new value needs `prices[i] = prices[i] + 1` in an indexed loop.
3. No. An enhanced for loop always goes from the first element to the last. Going backward needs an indexed loop that starts at `prices.length - 1` and counts down.

A complete answer gives a reason for each: the enhanced loop works on copies, always goes forward through every element, and has no index.

Reread 4.4.7 *Choosing and Rewriting a Traversal* if your answer differed.

Question 13

Corrected line 4:

```
1    System.out.println(ages[i]);
```

The loop counter `i` holds a position (0, 1, 2), not an element. In the enhanced loop, the variable `age` was the element itself; in the indexed version, the element at position `i` is `ages[i]`. With the fix, the program prints 15, 17, and 16. The original compiled and ran without any error, because printing an `int` counter is legal; it just printed the wrong values.

Reread 4.4.7 *Choosing and Rewriting a Traversal* if your fix differed.

Question 14

```
1
20
3
40
5
60
```

The first loop starts at index 1 and steps by 2, so `i` takes the values 1, 3, and 5 (then 7, and `7 < 6` is false). It multiplies only those elements by 10:

i	Before	After
1	2	20
3	4	40
5	6	60

The array is now {1, 20, 3, 40, 5, 60}. The second loop assigns 0 to its loop variable, which is only a copy, so the array does not change. The third loop prints each element.

Reread 4.4.2 *Other Starting Points, Directions, and Steps* and 4.4.5 *Assigning to the Loop Variable* if your output differed.

Question 15

B. `new Player[3]` makes an array of three elements, each starting as `null` (Topic 4.3). Line 33 fills only `team[0]`. On the first pass, `p` refers to Ada, so line 36 adds 1 and line 37 prints Ada 11. On the second pass, `p` is a copy of `team[1]`, which is `null`, and calling `p.addPoints(1)` on `null` throws a `NullPointerException` (Topic 1.14). The message looks something like `java.lang.NullPointerException: Cannot invoke "Player.addPoints(int)"...`, and its wording can vary between Java versions.

- **A** is wrong: an enhanced for loop visits every element, including `null` ones. It does not check for them.
- **C** is wrong: `null` is not an object, so there is no name or point total to print. Calling a method on it crashes.
- **D** is wrong: the enhanced for loop never uses an index outside the array, and the crash happens on the second pass, after Ada was printed.

Reread 4.4.6 *Enhanced for Loops and Objects* if you chose differently.

Question 16

On each pass, `s` receives a copy of one element of the `scores` array. Line 19 sets that copy to 0, but the array's elements are never assigned, so the scores stay 4, 0, and 7. The code compiles and runs without any error message, which is why the bug is easy to miss.

Corrected lines 17 to 20:

```
1 for (int i = 0; i < scores.length; i++)
2 {
3     scores[i] = 0;
4 }
```

Assigning to `scores[i]` stores into the array itself. With this change the program prints `0 0 0`. The fact that `scores` is a private instance variable does not matter here; `resetAll` is inside `Scoreboard`, so it may use the array either way.

Reread 4.4.5 *Assigning to the Loop Variable* if your fix differed.

Question 17

Model answer: In both loops, the loop variable gets a copy of a reference, so it refers to the same object as the array element. In `TeamBonus`, `p.addPoints(5)` is a mutator: it goes to the shared `Player` object and changes it, so the array's elements, which refer to the same objects, see the new points. A `String` has no mutators, because strings are immutable, so `ShoutNames` could only build a new string with `toUpperCase()` and assign it to `guest`. Assigning to the loop variable changes only that variable, never the reference stored in the array.

A complete answer includes:

- Both loop variables are aliases of the objects in the array.
- Calling a mutator changes the shared object; assigning to the loop variable does not change the array.
- Strings are immutable, so the only option for a `String` is assignment, which does not stick.

Reread 4.4.5 *Assigning to the Loop Variable* and 4.4.6 *Enhanced for Loops and Objects* if your answer differed.

Question 18

A. The enhanced loop prints `2.5`, `4.0`, and `1.25`, every element from first to last. Choice A starts at 0, continues while `i` is a valid index, prints `prices[i]`, and moves to the next index, so it prints the same three lines in the same order.

- **B** is wrong: the update happens **before** the print, so the first element is skipped. It prints `4.0` and `1.25`, then, when `i` becomes 3 inside the body, `prices[3]` throws an `ArrayIndexOutOfBoundsException`.
- **C** is wrong: its test is fine (`<= prices.length - 1` means the same as `< prices.length`), but it never updates `i`, so `i` stays 0 and the loop never ends.
- **D** is wrong: it visits every element safely, but backward, printing `1.25`, `4.0`, `2.5`. An enhanced for loop always goes first to last, so the order does not match.

Reread 4.4.3 *Traversing with a while Loop* and 4.4.7 *Choosing and Rewriting a Traversal* if you chose differently.

Question 19

```
10
16
i = 2
```

i	i < 4?	readings[i]	readings[i] >= 0?	Printed
0	true	5	true	10
1	true	8	true	16
2	true	-1	false	(loop ends)

The loop stops at the first negative reading, so the 4 after it is never visited. `i` still exists after the loop because it was declared before it, and line 8 prints `i = 2`.

Reread 4.4.3 *Traversing with a while Loop* if your output differed.

Question 20

A. `data.length - 2` is 5, so `i` starts at 5. It then becomes 3 and 1, and each of those passes the test `i >= 1`. The next value, -1, fails the test, so the loop ends after 3 passes. The program prints 5: 60, 3: 40, and 1: 20.

- **B** is wrong: that would need a start of `data.length - 1` and a test of `i >= 0`.
- **C** is wrong: the start is `data.length - 2`, which is 5, not 6.
- **D** is wrong: the test is checked before each pass, and `-1 >= 1` is false, so the body never runs with `i = -1`.

Reread 4.4.2 *Other Starting Points, Directions, and Steps* if you chose differently.