

4.3 Array Creation and Access

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

By the end you will be able to

- Explain what an array is and why it stores values that all have the same type.
- Create an array with new and with an initializer list, and state the starting value of every element.
- Read and change one element of an array using its index in square brackets.
- Use the length attribute to find how many elements an array has, and explain why that number can never change.
- Predict when an index causes an `ArrayIndexOutOfBoundsException`, and fix code that uses an index outside the valid range.
- Trace code that stores object references in an array, including elements that are still null.

What to remember from earlier

- Each character of a `String` has an index from 0 to length minus 1. ([Topic 1.15: String Manipulation, 1.15.4 Indices and the length Method](#))
- An index outside that range throws a `StringIndexOutOfBoundsException`. ([Topic 1.15: String Manipulation, 1.15.6 Out-of-Range Indices](#))
- A reference variable that refers to no object holds null. ([Topic 1.13: Object Creation and Storage \(Instantiation\), 1.13.2 Reference Variables and null](#))
- Calling a method on null throws a `NullPointerException`. ([Topic 1.14: Calling Instance Methods, 1.14.5 Calling a Method on null](#))
- Instance variables start with default values: 0 for `int`, 0.0 for `double`, `false` for `boolean`, null for a reference. ([Topic 3.4: Constructors, 3.4.4 Default Values and the Default Constructor](#))

4.3.1 Arrays Store Many Values of One Type

Suppose a program keeps the scores from five quizzes. With only what you know so far, you would declare five variables: `quiz1`, `quiz2`, `quiz3`, `quiz4`, and `quiz5`. That works, but it causes problems quickly. A class with 30 quizzes needs 30 declarations. And if the program asks "which quiz do you want to see?" and gets back the number 3, there is no way to turn that number into the name `quiz3`; you would need a long chain of `if` statements, one per quiz.

In Topic 4.2 you saw that a data set is a collection of related values that a program works through one value at a time. Java needs a way to keep such a collection under **one name**, with each value reachable by its position. Think of a row of numbered mailboxes in an apartment lobby: one bank of boxes, all the same size, each with a number on the door. You find a box by its number, not by a separate name.

That structure is an **array**: a single object that stores a fixed number of values, all of the same type. Each value in an array is called an **element**. The type of the elements is the array's **element type**. An array of `int` holds only `int` values; an array of `String` holds only `String` references. (The

mailbox analogy stops working in one way: real mailboxes can hold letters of any shape, but every slot in a Java array holds exactly one value of the element type.)

Why must every element have the same type? Java needs to know, when your code mentions an element, what kind of value it is, so that it can check your code before it runs (the same reason every variable has a declared type, from Topic 1.2). If one array could hold a name in one slot and a score in the next, Java could not tell whether `+` on an element means addition or concatenation.

KEY IDEA

An array stores many values of one element type under a single name. The elements can be primitive values, such as `int` or `double`, or object references, such as `String`.

Declaring and creating an array

Making an array takes two parts, which are usually written in one statement:

```
1 int[] scores = new int[5];
```

Piece	What it is
<code>int[]</code>	The type of the variable: "array of <code>int</code> ". The empty square brackets mean "array of".
<code>scores</code>	The variable's name. It will hold a reference to the array.
<code>new int[5]</code>	Creates a new array object with room for 5 <code>int</code> elements. The number in the brackets is the size.
<code>=</code>	Stores the reference to the new array in <code>scores</code> , as with any object (Topic 1.13).

An array is an object, so it is created with the keyword `new`, just like the objects you made in Topic 1.13. And as with every object, the variable does not contain the array itself; it holds a reference to it. Here is a picture of memory after the statement runs (you will see in 4.3.3 why every slot holds 0):

```
scores --> [ 0 | 0 | 0 | 0 | 0 ]
```

WORKED EXAMPLE Arrays with different element types

Each statement below creates one array. Read the left side as "this variable refers to an array of ..." and the right side as "make a new array with room for ... elements".

```
1 double[] prices = new double[3];
2 boolean[] seatTaken = new boolean[40];
3 String[] names = new String[4];
```

Variable	Element type	Number of elements	Kind of element
<code>prices</code>	<code>double</code>	3	primitive value
<code>seatTaken</code>	<code>boolean</code>	40	primitive value
<code>names</code>	<code>String</code>	4	reference to a <code>String</code> object

The element type is written twice, once on each side, and the two must match. The size appears only on the right, inside `new`, because the size belongs to the array object, not to the variable. A variable of type `double[]` can refer to a `double` array of any size.

WATCH OUT

Mismatched element types. The type on the left and the type after `new` must be the same:

WrongElementType.java

```
1 public class WrongElementType
2 {
3     public static void main(String[] args)
4     {
5         int[] scores = new double[5];
6         String[] names = new String[4];
7     }
8 }
```

WrongElementType.java:5: error: incompatible types: double[] cannot be converted to int[]

```
    int[] scores = new double[5];
                        ^
```

1 error

Students sometimes expect this to work because a single `double` value can be cast to `int` (Topic 1.5). But an array of `double` is a different kind of object from an array of `int`, and Java never converts one array type into another. Line 6 is fine because both sides say `String`. Fix line 5 by making both sides agree: `int[] scores = new int[5];`.

A second common slip is putting the size on the left, as in `int[5] scores;`. Java reports error: `' ] ' expected`, because the brackets in a type are always empty. The size goes after `new`.

SizeInDeclaration.java

```
1 public class SizeInDeclaration
2 {
3     public static void main(String[] args)
4     {
5         int[5] scores;
6     }
7 }
```

PAUSE AND TRY

Question 1 Multiple choice

A weather program needs one array to hold the rainfall, in inches, for each of the 12 months. Which statement creates it correctly?

- A. `double[] rainfall = new double(12);`
- B. `double rainfall = new double[12];`

- C. `double[] rainfall = new double[12];`
- D. `double[] rainfall = new int[12];`

Question 2 Short answer

A student wants one array to store each player's name and that player's score, alternating: name, score, name, score. Explain why Java does not allow this kind of array.

4.3.2 Reading and Changing Elements with an Index

An array is only useful if you can reach the values inside it. In the mailbox picture, you open a box by its number. In Java, that number is the element's **index**: its position in the array. You met indexes in Topic 1.15, where the characters of a `String` were numbered starting from 0. Arrays use the same numbering. The first element is at index 0, the second at index 1, and so on.

To name one element, write the array variable, then the index in square brackets: `scores[2]` means "the element of `scores` at index 2". This is called accessing an element. An expression like `scores[2]` behaves exactly like a variable of the element type: you can print it, use it in arithmetic, and assign a new value to it.

KEY IDEA

`arrayName[index]` names one element. On the right side of `=` (or anywhere a value is needed) it reads the element; on the left side of `=` it stores a new value in that element.

Here is how the indexes line up for an array of three scores:

```
index:    0    1    2
value:   88   92   75
```

WORKED EXAMPLE Storing and reading quiz scores

Read each line and decide whether it stores into an element (the element is on the left of `=`) or reads one.

```
1 int[] scores = new int[3];
2 scores[0] = 88;
3 scores[1] = 92;
4 scores[2] = 75;
5 System.out.println(scores[1]);
6 scores[2] = scores[2] + 10;
7 System.out.println(scores[2]);
8 System.out.println(scores[0] + scores[1]);
```

Line	What happens	scores[0]	scores[1]	scores[2]	Printed
1	Create an array of 3 <code>int</code> elements	0	0	0	
2	Store 88 at index 0	88	0	0	
3	Store 92 at index 1	88	92	0	

4	Store 75 at index 2	88	92	75	
5	Read index 1	88	92	75	92
6	Read index 2 (75), add 10, store the 85 at index 2	88	92	85	
7	Read index 2	88	92	85	85
8	Read indexes 0 and 1 and add them as <code>int</code> values	88	92	85	180

92
85
180

Line 6 shows the two roles in one statement. Java evaluates the right side first, so `scores[2]` there **reads** 75. Then the left side `scores[2]` **receives** the result, 85. This is the same right-to-left order you learned for assignment in Topic 1.4; an element is just another place to store a value.

The index does not have to be a literal number. Any expression that produces an `int` works, including a variable or a little arithmetic. This is what solves the problem from 4.3.1: a number the program computes (or reads as input) can choose the element.

WORKED EXAMPLE Choosing the element with a variable

`stepCounts` holds the steps walked on four days. The variable `day` picks which day to look at.

```

1  int[] stepCounts = new int[4];
2  stepCounts[0] = 4200;
3  stepCounts[1] = 6100;
4  stepCounts[2] = 3900;
5  stepCounts[3] = 7500;
6  int day = 2;
7  System.out.println(stepCounts[day]);
8  System.out.println(stepCounts[day + 1]);
9  stepCounts[day] += 500;
10 System.out.println(stepCounts[day]);
11 day = 0;
12 System.out.println(stepCounts[day]);

```

Line	Index expression	Index used	Effect	Printed
7	<code>day</code>	2	Reads <code>stepCounts[2]</code>	3900
8	<code>day + 1</code>	3	Reads <code>stepCounts[3]</code>	7500
9	<code>day</code>	2	<code>+=</code> adds 500: 3900 becomes 4400	
10	<code>day</code>	2	Reads <code>stepCounts[2]</code>	4400
12	<code>day</code>	0	<code>day</code> was changed to 0 on line 11	4200

```
3900
7500
4400
4200
```

Java works out the index expression first, then goes to that element. On line 8, `day + 1` is 3, so the element at index 3 is read; `day` itself is not changed. Line 9 shows that the compound assignment operators from Topic 1.6 work on elements too: `stepCounts[day] += 500;` means `stepCounts[day] = stepCounts[day] + 500;`. Lines 10 and 12 are the same code, `stepCounts[day]`, but they read different elements because `day` holds a different value each time.

WATCH OUT

Counting from 1. People number things starting at 1, so "the first score" feels like index 1:

```
1 int[] scores = new int[3];
2 scores[0] = 88;
3 scores[1] = 92;
4 scores[2] = 75;
5 System.out.print("First score: ");
6 System.out.println(scores[1]);
```

```
First score: 92
```

The program compiles and runs, but it prints the **second** score. This is a logic error, so Java gives no message at all. The first element is always at index 0. A quick check: the element at index `k` is element number `k + 1` when you count like a person. Fix line 6 to `System.out.println(scores[0]);`, which prints `First score: 88`.

PAUSE AND TRY

Question 3 Tracing

Trace this code. Give the value of each element after every line, and the exact output.

```
1 int[] prices = new int[3];
2 prices[0] = 12;
3 prices[1] = prices[0] * 2;
4 prices[2] = prices[1] - prices[0];
5 prices[0] = prices[2] + 1;
6 System.out.println(prices[0]);
7 System.out.println(prices[1]);
8 System.out.println(prices[2]);
```

Question 4 Multiple choice

An array `temps` holds seven daily temperatures, Sunday through Saturday, in order. Which expression is Wednesday's temperature?

- A. `temps[4]`
- B. `temps[3]`
- C. `temps(3)`
- D. `temps.3`

4.3.3 New Arrays Start with Default Values

In 4.3.2 the first row of each trace showed every element as `0` before anything was stored. That was not a guess. When `new` creates an array, Java does not leave the memory holding leftover junk. It fills every element with a **default value** that depends only on the element type.

You have seen these exact values before. In Topic 3.4, instance variables got default values before a constructor set them. Array elements follow the same rules, for the same reason: memory for a new object always starts in a known state, so a program never reads an unpredictable value.

Element type	Default value
<code>int</code>	<code>0</code>
<code>double</code>	<code>0.0</code>
<code>boolean</code>	<code>false</code>
any reference type, like <code>String</code>	<code>null</code>

KEY IDEA

An array created with `new` has every element set to the default for its type: `0` for `int`, `0.0` for `double`, `false` for `boolean`, and `null` for any reference type.

WORKED EXAMPLE Reading elements nobody assigned

None of these elements is ever assigned. Each print shows the default.

```
1 int[] goals = new int[3];
2 double[] heights = new double[3];
3 boolean[] present = new boolean[3];
4 String[] names = new String[3];
5 System.out.println(goals[0]);
6 System.out.println(heights[1]);
7 System.out.println(present[2]);
8 System.out.println(names[0]);
```

```
0
0.0
false
null
```

Line 6 prints `0.0`, not `0`, because the element type is `double`, and a `double` always prints with a decimal point. Line 8 prints the word `null`: `names[0]` holds no reference at all, since no `String` object has been created for that slot. `println` shows a missing reference as `null`.

Default values are not only something to watch out for; programs rely on them. Here is a tally of votes for three candidates. Each element counts the votes for one candidate, and each count needs to start at 0, which `new` already did.

WORKED EXAMPLE A vote tally that starts at zero

```
1 int[] votes = new int[3];
2 votes[1]++;
3 votes[2]++;
4 votes[1]++;
5 System.out.println(votes[0]);
6 System.out.println(votes[1]);
7 System.out.println(votes[2]);
```

Line	Statement	votes[0]	votes[1]	votes[2]
1	create array	0	0	0
2	votes[1]++	0	1	0
3	votes[2]++	0	1	1
4	votes[1]++	0	2	1

```
0
2
1
```

`votes[1]++` adds 1 to the element at index 1 (Topic 1.6). Nobody had to write `votes[0] = 0`; first. Candidate 0 got no votes, and the default 0 correctly says so.

WATCH OUT

Treating null elements as empty strings. A new `String` array holds `null` in every slot, not `""`. A `null` element is not a `String` object, so calling a method on it fails:

NullName.java

```
1 public class NullName
2 {
3     public static void main(String[] args)
4     {
5         String[] names = new String[3];
6         names[0] = "Ava";
7         System.out.println(names[0].length());
8         System.out.println(names[1].length());
9     }
10 }
```

```
3
Exception in thread "main" java.lang.NullPointerException: Cannot invoke
"String.length()" because "<local1>[1]" is null
at NullName.main(NullName.java:8)
```

Line 7 works: `names[0]` refers to "Ava", which has 3 characters. Line 8 crashes with the `NullPointerException` from Topic 1.14, because `names[1]` was never assigned and still holds `null`. (The text after the exception name varies between Java versions.) Students expect "" because an empty string sounds like "nothing yet", but "" is a real `String` object with length 0, and `null` means no object at all. Fix it by storing a `String` in the element before calling a method on it, for example `names[1] = "Ben";`.

PAUSE AND TRY

Question 5 Tracing

Write the exact output.

```
1 double[] tips = new double[4];
2 boolean[] paid = new boolean[4];
3 tips[1] = 3.5;
4 paid[1] = true;
5 tips[2] = tips[1] + tips[3];
6 System.out.println(tips[2]);
7 System.out.println(tips[0]);
8 System.out.println(paid[0]);
9 System.out.println(paid[1]);
```

Question 6 Multiple choice

After `String[] titles = new String[5];` runs, what does `titles[4]` hold?

- A. "" (an empty string)
- B. The four-letter string "null"
- C. `null`
- D. Nothing; reading an element before assigning it is a compile error

4.3.4 Creating an Array with an Initializer List

Often you already know every value when you write the program: the days of the school week, the prices on a menu, the answer key to a quiz. Creating an array with `new` and then assigning each element on its own line works, but it takes one line per element and makes the values hard to see at a glance.

Java has a shorter way. An **initializer list** is a list of values inside curly braces, separated by commas, that creates an array and fills it in one step:

```
1 String[] days = {"Mon", "Tue", "Wed", "Thu", "Fri"};
```

Piece	What it is
<code>String[] days</code>	Declares a variable that refers to an array of <code>String</code>
<code>{</code>	Starts the list of starting values
<code>"Mon", "Tue", ...</code>	The elements, in index order: "Mon" goes to index 0, "Tue" to index 1

}	Ends the list
---	---------------

There is no `new` and no `size`. Java counts the values in the list (five here) and makes an array of exactly that many elements. Every value in the list must fit the element type.

KEY IDEA

An initializer list such as `{88, 92, 75}` creates an array, sets its size to the number of values listed, and stores the values in order starting at index 0.

WORKED EXAMPLE Three arrays from initializer lists

```
1 String[] days = {"Mon", "Tue", "Wed", "Thu", "Fri"};
2 double[] prices = {2.50, 3.75, 1.25};
3 boolean[] answers = {true, false, true, true};
4 System.out.println(days[2]);
5 System.out.println(prices[1]);
6 System.out.println(answers[1]);
7 days[0] = "Monday";
8 System.out.println(days[0]);
```

Each list sets both the size and the contents:

days	index:	0	1	2	3	4
	value:	"Mon"	"Tue"	"Wed"	"Thu"	"Fri"

prices	index:	0	1	2
	value:	2.5	3.75	1.25

answers	index:	0	1	2	3
	value:	true	false	true	true

```
Wed
3.75
false
Monday
```

Line 4 reads index 2 of `days`, the third value in its list. Line 7 shows that an array made from an initializer list is an ordinary array: you can change its elements afterward with square brackets, just like an array made with `new`.

The values in the list do not have to be literals. Any expression of the right type is allowed, and Java evaluates each one when the array is created.

WORKED EXAMPLE Expressions in the list

```
1 int base = 10;
2 int[] targets = {base, base * 2, base * 3 + 5};
3 System.out.println(targets[2]);
4 base = 50;
5 System.out.println(targets[0]);
```

Line	What happens	targets	base
1	base gets 10		10
2	base is 10, base * 2 is 20, base * 3 + 5 is 35	{10, 20, 35}	10
3	Prints index 2	{10, 20, 35}	10
4	base gets 50	{10, 20, 35}	50
5	Prints index 0	{10, 20, 35}	50

```
35
10
```

Line 5 still prints 10. On line 2 Java copied the **value** of `base` into the element. The element and base are separate storage, so changing base later does not reach back into the array.

Both ways of creating an array are useful. Here they are side by side:

Question	<code>new int[3]</code>	<code>{88, 92, 75}</code>
How is the size decided?	The number in the brackets	The number of values in the list
What do the elements start as?	Default values (0 here)	The listed values
When is it the better choice?	Values are not known yet, or start at 0	All values are known when writing the code

WATCH OUT

Using an initializer list after the declaration. An initializer list is allowed only as part of the declaration that creates the array. It cannot be assigned to an existing array variable later:

InitializerLater.java

```
1 public class InitializerLater
2 {
3     public static void main(String[] args)
4     {
5         int[] scores = {88, 92, 75};
6         scores = {90, 95, 80};
7     }
8 }
```

```
InitializerLater.java:6: error: illegal start of expression
    scores = {90, 95, 80};
            ^
```

(Java prints a few more errors on the same line, because once it is confused it misreads the rest of the line.) Students try this because line 5 worked, so line 6 looks like the same idea. But the braces only mean "array values" right after a declaration such as `int[] scores =`; anywhere else, Java does not know what type of array to build. To change the values of the existing array, assign the elements one at a time: `scores[0] = 90;`, `scores[1] = 95;`, `scores[2] = 80;`.

PAUSE AND TRY**Question 7** Tracing

Write the exact output. Work out each index expression before you read the element.

```
1 int[] laps = {4, 7, 2, 9};
2 laps[2] = laps[0] + laps[3];
3 laps[0] = laps[laps[2] - 10];
4 System.out.println(laps[0]);
5 System.out.println(laps[2]);
```

Question 8 Debugging

This program does not compile. Find both errors, explain each one, and write corrected lines.

MixedList.java

```
1 public class MixedList
2 {
3     public static void main(String[] args)
4     {
5         int[] ages = {15, 16.5, 17};
6         String[] labels = {"gold", 2, "bronze"};
7     }
8 }
```

4.3.5 Array Length Is Fixed

Code that works with an array often needs to know how many elements it has, for example to reach the last element. Hard-coding the number is risky: if someone later changes the initializer list from 4 values to 6, every 3 that meant "the last index" is now wrong.

Every array carries its own count. The number of elements is the array's **length**, and you read it with the **length attribute**: write the array variable, a dot, and `length`, with **no parentheses**. `colors.length` is a value stored with the array object, not a method call.

The length is set once, at the moment the array is created (by the number in `new int[6]` or by the count of values in the list), and it **can never change** afterward. Java gives an array one block of memory with room for exactly that many elements, side by side; there is no extra room to grow into. The `length` attribute acts like a `final` variable from Topic 3.7: you may read it, but no statement can assign to it.

KEY IDEA

An array's length is fixed when the array is created. Read it with `arrayName.length` (no parentheses). The last element is always at index `arrayName.length - 1`.

WORKED EXAMPLE Reading the length

```

1 String[] colors = {"red", "green", "blue", "gold"};
2 int[] blank = new int[6];
3 System.out.println(colors.length);
4 System.out.println(blank.length);
5 System.out.println(colors[colors.length - 1]);
6 System.out.println(colors[0].length());

```

Line	Expression	Value	Printed
3	colors.length	4 values in the list	4
4	blank.length	the size given to new, even though every element is 0	6
5	colors[colors.length - 1]	index 4 - 1 = 3, which holds "gold"	gold
6	colors[0].length()	the String "red" has 3 characters	3

```

4
6
gold
3

```

Line 5 is the standard way to reach the last element: it works for an array of any length, because it asks the array itself. Line 6 is a different thing: `colors[0]` is a `String`, and `length()` with parentheses is the `String` **method** from Topic 1.15.

Lines 5 and 6 sit next to each other on purpose, because these two are easy to mix up:

You have...	How to get its size	Why	Example
an array	<code>.length</code>	<code>length</code> is an attribute of the array	<code>colors.length</code>
a <code>String</code>	<code>.length()</code>	<code>length</code> is a method of <code>String</code>	<code>"red".length()</code>

WORKED EXAMPLE Trying to make an array bigger

A program has three scores and wants to add a fourth. A student writes `scores = new int[4];`, hoping to stretch the array:

```

1 int[] scores = {88, 92, 75};
2 scores = new int[4];
3 scores[3] = 81;
4 System.out.println(scores.length);
5 System.out.println(scores[0]);
6 System.out.println(scores[3]);

```

```

4
0
81

```

The length did go from 3 to 4, so it looks as though the array grew. It did not. Line 2 created a **brand-new** array of four elements, all 0, and stored its reference in `scores`. The old array,

holding 88, 92, and 75, still has length 3; the program simply has no reference to it anymore. Line 5 proves it: `scores[0]` is 0, not 88.

```
before line 2:  scores --> [ 88 | 92 | 75 ]

after line 2:   scores --> [ 0 | 0 | 0 | 0 ]
                  [ 88 | 92 | 75 ]   (no variable refers to this one)
```

A variable can be pointed at a different array, but an existing array's length never changes. To keep the old values in a bigger array, a program has to create the new array and then copy each value across, element by element.

WATCH OUT

Writing `length()` or assigning to `length`. Both of these come from treating an array like a String or like an ordinary variable:

LengthMistakes.java

```
1 public class LengthMistakes
2 {
3     public static void main(String[] args)
4     {
5         int[] scores = {88, 92, 75};
6         System.out.println(scores.length());
7         scores.length = 4;
8     }
9 }
```

```
LengthMistakes.java:6: error: cannot find symbol
    System.out.println(scores.length());
                        ^
    symbol:   method length()
    location: variable scores of type int[]
LengthMistakes.java:7: error: cannot assign a value to final variable length
    scores.length = 4;
                ^

2 errors
```

Line 6: arrays have no `length()` method, so Java cannot find one. Remove the parentheses: `scores.length`. Line 7: the length is final, so Java refuses to change it. There is no fix that resizes the array; create a new array of the size you need instead.

PAUSE AND TRY

Question 9 Tracing

Write the exact output.

```

1 double[] times = {12.4, 11.9, 13.2, 12.8, 12.1};
2 int last = times.length - 1;
3 System.out.println(times.length);
4 System.out.println(times[last]);
5 System.out.println(times[times.length / 2]);

```

Question 10 Multiple choice

`int[] scores` refers to an array. Which expression is the number of elements in the array?

- A. `scores.length()`
- B. `scores.length`
- C. `scores[scores.length]`
- D. `scores.length - 1`

4.3.6 Valid Indexes and Out-of-Bounds Errors

Indexes start at 0, and the last element is at `length - 1`. Put those together and you get the full rule: for an array of length `n`, the **valid indexes** are 0, 1, 2, ... up to `n - 1`, inclusive. An array of length 3 has valid indexes 0, 1, and 2, and nothing else. Index 3 is not "the end of the array"; it is a slot that does not exist.

index:	-1		0	1	2		3
			[88	92	75	]	
	out		valid indexes				out

Using any index outside that range, whether to read or to store, makes Java throw an **ArrayIndexOutOfBoundsException** while the program runs. This is the array version of the `StringIndexOutOfBoundsException` from Topic 1.15. Java checks every index at the moment it is used, because silently reading or writing memory outside the array would damage other data in the program.

Why doesn't the compiler catch it? The compiler checks only that the index is an `int`. It does not know what value a variable like `day` will hold or how long an array will be when the line runs, so a bad index can be found only while the program runs.

KEY IDEA

The valid indexes of an array are 0 through `length - 1`, inclusive. Any other index, including `length` itself or a negative number, throws an `ArrayIndexOutOfBoundsException` when that line runs.

WORKED EXAMPLE One step too far

The most common out-of-bounds index is `length` itself:

OffByOne.java

```

1 public class OffByOne
2 {
3     public static void main(String[] args)
4     {
5         int[] scores = {88, 92, 75};
6         System.out.println(scores[scores.length - 1]);
7         System.out.println(scores[scores.length]);
8         System.out.println("Done");
9     }
10 }

```

75

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 3 out of bounds for length 3
at OffByOne.main(OffByOne.java:7)

Line	Index used	Valid?	Result
6	scores.length - 1 is 2	yes	prints 75
7	scores.length is 3	no	exception; the program stops
8	(never runs)		

The program compiles without complaint. Line 6 runs normally. Line 7 throws the exception, and because nothing handles it, the program ends right there; Done is never printed. The message names the bad index and the array's length, which is the first thing to read when you see this error. (The wording after the exception name can vary between Java versions.)

Negative indexes are just as invalid, and they often come from converting a person's numbering (starting at 1) to an index by subtracting 1.

WORKED EXAMPLE A negative index

NegativeIndex.java

```

1 public class NegativeIndex
2 {
3     public static void main(String[] args)
4     {
5         String[] floors = {"Lobby", "Gym", "Pool"};
6         int choice = 1;
7         floors[choice - 1] = "Entrance";
8         System.out.println(floors[0]);
9         choice = 0;
10        floors[choice - 1] = "Basement";
11        System.out.println(floors[0]);
12    }
13 }

```

Entrance

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index -1 out of bounds for length 3
at NegativeIndex.main(NegativeIndex.java:10)

On line 7, choice - 1 is 0, a valid index, so "Lobby" is replaced by "Entrance" and line 8 prints it. On line 10, choice - 1 is -1. Storing into an element is checked exactly like reading one, so the exception is thrown before anything is stored, and line 11 never runs.

When an index comes from a calculation or from input, you can make sure it is valid **before** using it with an `if` statement and a compound condition from Topics 2.3 and 2.5:

WORKED EXAMPLE Checking an index first

GuardedLookup.java

```

1 public class GuardedLookup
2 {
3     public static void main(String[] args)
4     {
5         String[] prizes = {"sticker", "pen", "mug", "hoodie"};
6         int pick = 4;
7         if (pick >= 0 && pick < prizes.length)
8         {
9             System.out.println(prizes[pick]);
10        }
11        else
12        {
13            System.out.println("No prize at that spot");
14        }
15        pick = 3;
16        if (pick >= 0 && pick < prizes.length)
17        {
18            System.out.println(prizes[pick]);
19        }
20        else
21        {
22            System.out.println("No prize at that spot");
23        }
24    }
25 }
```

No prize at that spot
hoodie

The condition `pick >= 0 && pick < prizes.length` is exactly the valid-index rule written in Java. Notice `<`, not `<=`: with 4 prizes, `prizes.length` is 4, and 4 is not a valid index. On line 7, `pick` is 4, so `4 < 4` is false and the `else` branch runs instead of crashing. On line 16, `pick` is 3, which passes both tests, so line 18 reads `prizes[3]`.

WATCH OUT

Expecting the array to grow when you store past the end. Some students expect `scores[3] = 60`; on a three-element array to add a fourth element:

StoreOutOfBounds.java

```
1 public class StoreOutOfBounds
2 {
3     public static void main(String[] args)
4     {
5         int[] scores = new int[3];
6         scores[0] = 88;
7         scores[1] = 92;
8         scores[2] = 75;
9         System.out.println(scores.length);
10        scores[3] = 60;
11        System.out.println(scores.length);
12    }
13 }
```

```
3
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 3 out of
bounds for length 3
    at StoreOutOfBounds.main(StoreOutOfBounds.java:10)
```

Line 10 does not add anything. The length was fixed at 3 when the array was created (4.3.5), so index 3 does not exist and Java throws the exception. If the program needs room for four scores, create the array with `new int[4]` from the start.

PAUSE AND TRY**Question 11** Multiple choice

After `int[] codes = new int[8];`, which expression throws an `ArrayIndexOutOfBoundsException` when it runs?

- A. `codes[0]`
- B. `codes[7]`
- C. `codes[8]`
- D. `codes[codes.length - 8]`

Question 12 Short answer

A student says, "My program compiled with no errors, so every array index in it must be valid." Explain why the student is wrong.

4.3.7 Arrays of Object References

In 4.3.1 you read that elements can be primitive values or object references. So far most examples used `int`, `double`, and `boolean`, where each element holds the value itself. When the element type

is a class, such as `String` or a class you wrote in Unit 3, each element holds a reference to an object, exactly as a variable of that class type would (Topic 1.13).

That one fact explains everything about these arrays:

- **Creating the array does not create the objects.** `new Player[3]` makes an array with three slots for references. Each slot starts as `null` (4.3.3). No `Player` objects exist until you create them with `new Player(...)` and store them in the slots.
- **An element can be used like any variable of its class.** `team[0].getName()` follows the reference in `team[0]` and calls `getName` on that object, using the dot operator from Topic 1.14.
- **Copying an element copies the reference.** As in Topic 3.6, two elements, or an element and a variable, can refer to the same object.

KEY IDEA

In an array of a class type, each element holds a reference to an object, or `null`.
`new ClassName[n]` creates only the slots; each object must be created separately and stored in an element before its methods can be called.

WORKED EXAMPLE A team roster

This file has two classes. 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.

TeamRoster.java

```
1 class Player
2 {
3     private String name;
4     private int points;
5
6     public Player(String name)
7     {
8         this.name = name;
9         points = 0;
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 TeamRoster
```

```

29 {
30     public static void main(String[] args)
31     {
32         Player[] team = new Player[3];
33         System.out.println(team[0]);
34         team[0] = new Player("Ana");
35         team[1] = new Player("Ben");
36         team[1].addPoints(6);
37         team[0].addPoints(4);
38         team[1].addPoints(2);
39         System.out.println(team[0].getName());
40         System.out.println(team[1].getPoints());
41         System.out.println(team[2]);
42     }
43 }

```

Line	What happens	team[0]	team[1]	team[2]	Printed
32	Array of 3 references created	null	null	null	
33	Print team[0]	null	null	null	null
34	New Player Ana, 0 points	Ana, 0	null	null	
35	New Player Ben, 0 points	Ana, 0	Ben, 0	null	
36	addPoints(6) on Ben	Ana, 0	Ben, 6	null	
37	addPoints(4) on Ana	Ana, 4	Ben, 6	null	
38	addPoints(2) on Ben	Ana, 4	Ben, 8	null	
39	team[0].getName()	Ana, 4	Ben, 8	null	Ana
40	team[1].getPoints()	Ana, 4	Ben, 8	null	8
41	Print team[2]	Ana, 4	Ben, 8	null	null

```

null
Ana
8
null

```

In the table, "Ana, 4" is short for "a reference to the `Player` object whose name is Ana and whose points are 4". The objects live outside the array; the array holds only the references. `team[2]` was never given an object, so it is still `null` at the end.

The same rules apply to an array of `String`. Each element refers to a `String` object, so you can call `String` methods from Topic 1.15 on an element:

```

cities --> [ ref | ref | ref ]
           |   |   |
           v   v   v
        "Denver" "Austin" "Boston"

```

WORKED EXAMPLE Calling String methods on elements

```
1 String[] cities = {"Denver", "Austin", "Boston"};
2 String code = cities[1].substring(0, 3);
3 System.out.println(code);
4 System.out.println(cities[2].length());
5 cities[0] = cities[0] + " C0";
6 System.out.println(cities[0]);
```

```
Aus
6
Denver C0
```

Line 2 calls `substring(0, 3)` on the object `cities[1]` refers to, "Austin", giving "Aus". Line 4 asks "Boston" for its length, 6. Line 5 builds a new String by concatenation and stores its reference in `cities[0]`; strings are immutable (Topic 1.15), so the element now refers to a different object, "Denver C0".

Because elements hold references, an element and a variable can be aliases, two names for one object.

WORKED EXAMPLE Two names for one player

`SharedPlayer.java` uses the same `Player` class as `TeamRoster.java`; only `main` (starting at line 30) is different.

`SharedPlayer.java`

```
1 class Player
2 {
3     private String name;
4     private int points;
5
6     public Player(String name)
7     {
8         this.name = name;
9         points = 0;
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 SharedPlayer
29 {
30     public static void main(String[] args)
31     {
32         Player star = new Player("Cara");
33         Player[] lineup = new Player[2];
34         lineup[0] = star;
35         lineup[1] = new Player("Dev");
36         lineup[0].addPoints(10);
37         System.out.println(star.getPoints());
38         star.addPoints(3);
39         System.out.println(lineup[0].getPoints());
40         lineup[0] = lineup[1];
41         System.out.println(lineup[0].getName());
42         System.out.println(star.getName());
43     }
44 }

```

Line	What happens	Cara's points	lineup[0] refers to	Printed
32	star refers to a new Player, Cara	0		
34	The reference in star is copied into lineup[0]	0	Cara	
36	addPoints(10) through lineup[0] changes Cara's object	10	Cara	
37	star.getPoints() asks the same object	10	Cara	10
38-39	Adding 3 through star, then reading through lineup[0]	13	Cara	13
40	The reference in lineup[1] (Dev) is copied into lineup[0]	13	Dev	
41-42	lineup[0] now names Dev; star still names Cara	13	Dev	Dev, Cara

```

10
13
Dev
Cara

```

Line 34 did not copy the player; it copied the reference, so changes made through either name show up through the other. Line 40 replaced only the reference stored in `lineup[0]`. It did not change Cara's object or the variable `star`.

WATCH OUT

Calling a method on a slot that has no object. Students often assume `new Player[2]` creates two players:

EmptyRoster.java

```
1 class Player
2 {
3     private String name;
4     private int points;
5
6     public Player(String name)
7     {
8         this.name = name;
9         points = 0;
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 EmptyRoster
29 {
30     public static void main(String[] args)
31     {
32         Player[] team = new Player[2];
33         team[0] = new Player("Ana");
34         team[0].addPoints(5);
35         System.out.println(team[0].getPoints());
36         team[1].addPoints(5);
37         System.out.println(team[1].getPoints());
38     }
39 }
```

5

```
Exception in thread "main" java.lang.NullPointerException: Cannot invoke
"Player.addPoints(int)" because "<local1>[1]" is null
at EmptyRoster.main(EmptyRoster.java:36)
```

Line 32 created two empty slots. Line 33 filled `team[0]`, so lines 34 and 35 work. `team[1]` was never filled; it is still `null`, so line 36 throws a `NullPointerException`. Notice that this is not an index problem: index 1 is valid. The slot exists, but nothing is in it. The fix is to create the object first, for example `team[1] = new Player("Ben");` before line 36. (The text after the exception name varies between Java versions.)

PAUSE AND TRY**Question 13** Tracing

RosterTrace.java uses the same Player class as TeamRoster.java. Write the exact output.

RosterTrace.java

```
1 class Player
2 {
3     private String name;
4     private int points;
5
6     public Player(String name)
7     {
8         this.name = name;
9         points = 0;
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 RosterTrace
29 {
30     public static void main(String[] args)
31     {
32         Player[] squad = new Player[3];
33         squad[2] = new Player("Eli");
34         squad[0] = new Player("Fay");
35         squad[1] = squad[2];
36         squad[1].addPoints(7);
37         squad[2].addPoints(1);
38         squad[0].addPoints(squad[2].getPoints());
39         System.out.println(squad[0].getPoints());
40         System.out.println(squad[1].getName());
41         System.out.println(squad[2].getPoints());
42     }
43 }
```

Question 14 Multiple choice

After `Player[] bench = new Player[4];` runs, how many `Player` objects have been created?

A. 0

- B. 1
- C. 4
- D. 5

Putting it together

This program records lunch votes. It uses an initializer list for the menu, `new` with default values for the vote counts, the `length` attribute to keep the two arrays the same size, index arithmetic to reach the last element, and an index check before reading. Predict all five lines of output before you read the trace.

WORKED EXAMPLE Lunch vote

LunchVote.java

```
1 public class LunchVote
2 {
3     public static void main(String[] args)
4     {
5         String[] menu = {"Pizza", "Tacos", "Salad", "Soup"};
6         int[] votes = new int[menu.length];
7
8         votes[1]++;
9         votes[3]++;
10        votes[1]++;
11        votes[0] += 2;
12
13        System.out.println(menu.length + " choices");
14        System.out.println(menu[1] + ": " + votes[1]);
15        System.out.println(menu[2] + ": " + votes[2]);
16
17        int last = menu.length - 1;
18        System.out.println(menu[last] + ": " + votes[last]);
19
20        int request = 4;
21        if (request >= 0 && request < menu.length)
22        {
23            System.out.println(menu[request]);
24        }
25        else
26        {
27            System.out.println("Choice " + request + " is not on the menu");
28        }
29    }
30 }
```

Setting up (lines 5–6). The initializer list makes `menu` an array of 4 `String` references. Line 6 creates `votes` with `new int[menu.length]`, so it also has 4 elements, all starting at 0. Using `menu.length` instead of typing 4 keeps the arrays matched: if someone adds a fifth food to the list, `votes` grows with it the next time the program runs. The two arrays line up by index: `votes[k]` counts the votes for `menu[k]`.

```

index:    0      1      2      3
menu:    "Pizza" "Tacos" "Salad" "Soup"
votes:    0      0      0      0

```

Counting (lines 8–11).

Line	Statement	votes
8	<code>votes[1]++</code>	{0, 1, 0, 0}
9	<code>votes[3]++</code>	{0, 1, 0, 1}
10	<code>votes[1]++</code>	{0, 2, 0, 1}
11	<code>votes[0] += 2</code>	{2, 2, 0, 1}

Reporting (lines 13–28).

Line	Work	Printed
13	<code>menu.length</code> is 4	4 choices
14	<code>menu[1]</code> is "Tacos", <code>votes[1]</code> is 2	Tacos: 2
15	<code>menu[2]</code> is "Salad", <code>votes[2]</code> is still the default 0	Salad: 0
17–18	last is $4 - 1 = 3$; <code>menu[3]</code> is "Soup", <code>votes[3]</code> is 1	Soup: 1
20–27	request is 4; $4 < 4$ is false, so the else runs instead of crashing	Choice 4 is not on the menu

```

4 choices
Tacos: 2
Salad: 0
Soup: 1
Choice 4 is not on the menu

```

Without the check on line 21, line 23 would try `menu[4]` and throw an `ArrayIndexOutOfBoundsException`, because an array of length 4 has no index 4.

Common mistakes

Arrays produce a handful of error messages over and over. Use this table to go from the message to the cause.

Message (key words)	Usual cause	Fix
<code>incompatible types: double[] cannot be converted to int[]</code>	Element types differ on the two sides of <code>=</code>	Use the same element type on both sides
<code>']' expected</code>	A size inside the brackets of the type, as in <code>int[5] scores</code>	Leave the type's brackets empty; put the size after <code>new</code>
<code>illegal start of expression at {</code>	An initializer list assigned after the declaration	Assign elements one at a time, or use the list in the declaration
<code>cannot find symbol ... method length()</code>	<code>length()</code> written for an array	Write <code>length</code> with no parentheses

cannot assign a value to final variable length	Trying to change an array's length	Create a new array of the size you need
ArrayIndexOutOfBoundsException (while running)	An index below 0 or at least length, often <code>arr[arr.length]</code>	Use 0 through <code>length - 1</code> ; check an index before using it
NullPointerException (while running)	A method called on an element that is still <code>null</code>	Store an object in the element first

Vocabulary

Term	Definition	Example
array	An object that stores a fixed number of values, all of the same type, under one name	<code>int[] scores = new int[5];</code>
element	One value stored in an array	<code>scores[2]</code>
element type	The type every element of an array must have	<code>double</code> in <code>double[]</code>
index	An element's position in an array, starting at 0	2 in <code>scores[2]</code>
default value	The value Java puts in each element when <code>new</code> creates an array	0, 0.0, false, null
initializer list	Values in curly braces that create an array and set its size and contents in one step	<code>{88, 92, 75}</code>
length	The number of elements in an array, set at creation and never changed; read with <code>.length</code>	<code>scores.length</code>
valid index	Any index from 0 through <code>length - 1</code> , inclusive	0, 1, 2 for a length-3 array
ArrayIndexOutOfBoundsException	The run-time error thrown when code uses an index outside the valid range	<code>scores[scores.length]</code>

Chapter summary

- An array is an object that stores many values of a single element type under one name. The elements can be primitive values or object references, and the array variable holds a reference to the array.
- `new int[5]` creates an array of a given length with every element set to its default (0, 0.0, false, or null); an initializer list such as `{88, 92, 75}` creates an array whose length and contents come from the listed values, and it may appear only in the declaration.
- `arrayName[index]` reads or changes one element. The index can be any `int` expression, and the first element is at index 0.

- An array's length is fixed when it is created and is read with the `length` attribute, without parentheses. Assigning a new array to a variable makes a new array; it never resizes the old one.
- The valid indexes are 0 through `length - 1`. Any other index throws an `ArrayIndexOutOfBoundsException` when the line runs, which the compiler cannot catch.
- In an array of a class type, creating the array creates only `null` slots; each object must be created and stored before its methods are called, and copying an element copies a reference.

Review questions

Question 15 Multiple choice

Which statement about arrays is true?

- A. After `boolean[] flags = new boolean[3];`, every element of `flags` is `true`.
- B. An array created with an initializer list can have its length increased later by adding values to the list.
- C. After `double[] rates = {1.5, 2.0};`, the expression `rates[rates.length - 1]` is `2.0`.
- D. The valid indexes of an array of length 10 are 1 through 10.

Question 16 Tracing

Write the exact output. Some elements are never assigned.

```
1 double[] miles = new double[4];
2 int[] trips = {2, 5, 1};
3 miles[trips[2]] = 3.5;
4 miles[0] = miles[1] * trips.length;
5 trips[0] = trips[0] + miles.length;
6 System.out.println(miles[0]);
7 System.out.println(miles[3]);
8 System.out.println(trips[0]);
```

Question 17 Debugging

This program should print the highest badge, `Platinum`. Instead it prints `4` badges and then crashes. Explain the error and write the corrected line.

LastBadge.java

```
1 public class LastBadge
2 {
3     public static void main(String[] args)
4     {
5         String[] badges = {"Bronze", "Silver", "Gold", "Platinum"};
6         int count = badges.length;
7         System.out.println(count + " badges");
8         System.out.println("Top badge: " + badges[count]);
9     }
10 }
```

Question 18 Multiple choice

What happens when this program runs?

ShelfNull.java

```
1 class Book
2 {
3     private String title;
4
5     public Book(String title)
6     {
7         this.title = title;
8     }
9
10    public String getTitle()
11    {
12        return title;
13    }
14 }
15
16 public class ShelfNull
17 {
18     public static void main(String[] args)
19     {
20         Book[] shelf = new Book[3];
21         shelf[0] = new Book("Dune");
22         System.out.println(shelf[0].getTitle());
23         System.out.println(shelf[1].getTitle());
24     }
25 }
```

- A. It prints Dune, then null.
- B. It prints Dune, then throws a NullPointerException.
- C. It prints Dune, then throws an ArrayIndexOutOfBoundsException.
- D. It does not compile, because shelf[1] was never assigned.

Question 19 Short answer

A program declares `int[] scores = new int[30];` for a class of 30 students. Midyear, 2 students join. A student suggests writing `scores.length = 32;`. Explain why this does not work, and describe what `scores = new int[32];` would do instead.

Question 20 Tracing

Here the index of each element comes from another element of the same array. Write the exact output.

```
1 int[] code = {3, 0, 2, 1};
2 System.out.println(code[code[0]]);
3 code[code[3]] = code[code[2]] + 5;
4 System.out.println(code[1]);
5 System.out.println(code[code[1] - 5]);
```

Question 21 Multiple choice

Consider these three code segments.

```
1 // I
2 double[] d = new double[2];
3 d[2] = 1.5;
4
5 // II
6 boolean[] b = {true, false};
7 b[1] = !b[0];
8 System.out.println(b[1]);
9
10 // III
11 String[] s = new String[2];
12 System.out.println(s[0]);
```

Which segments compile and run without throwing an exception?

- A. I only
- B. II only
- C. II and III only
- D. I, II, and III

Reflection

☐ I can explain what an array is, why all its elements share one type, and declare and create one with `new`. Reread: 4.3.1 Arrays Store Many Values of One Type

☐ I can read and change one element using an index in square brackets, starting from index 0.

Reread: 4.3.2 Reading and Changing Elements with an Index

☐ I can state the default value of each element of a new `int`, `double`, `boolean`, or reference-type array. Reread: 4.3.3 New Arrays Start with Default Values

☐ I can create an array with an initializer list and say how its length is decided. Reread: 4.3.4

Creating an Array with an Initializer List

☐ I can use the `length` attribute, reach the last element, and explain why an array cannot grow.

Reread: 4.3.5 Array Length Is Fixed

☐ I can list the valid indexes of an array and predict when an `ArrayIndexOutOfBoundsException` is thrown. Reread: 4.3.6 Valid Indexes and Out-of-Bounds Errors

☐ I can trace an array of object references, including null slots and two names for one object.

Reread: 4.3.7 Arrays of Object References

Check Your Reasoning

Answers and explanations for every question, numbered to match.

Question 1

C. The variable's type is `double[]` (array of `double`), and `new double[12]` creates a `double` array with 12 elements. The element types match on both sides.

- **A** is wrong: an array's size goes in square brackets, not parentheses. Java reports `'[' expected.`
- **B** is wrong: the variable is declared as a single `double`, not an array of `double`, so it cannot hold a reference to an array. Java reports `incompatible types: double[] cannot be converted to double.`
- **D** is wrong: the element types disagree. Java reports `incompatible types: int[] cannot be converted to double[]; it never converts one array type into another.`

Reread 4.3.1 *Arrays Store Many Values of One Type* if you chose differently.

Question 2

Model answer: Every element of an array has the same element type, which is fixed when the array is declared. An array of `String` can hold only `String` references, and an array of `int` can hold only `int` values, so names and scores cannot share one array. Java requires this so that it always knows what type an element is and can check the code before it runs. The usual solution is two arrays, one of `String` for the names and one of `int` for the scores.

A complete answer says:

- All elements of one array have the same type.
- Names (`String`) and scores (`int`) are different types.

Reread 4.3.1 *Arrays Store Many Values of One Type* if your answer was missing one of these.

Question 3

```
13
24
12
```

Line	Right side worked out	prices[0]	prices[1]	prices[2]
1	(array created)	0	0	0
2	12	12	0	0
3	$12 * 2 = 24$	12	24	0
4	$24 - 12 = 12$	12	24	12
5	$12 + 1 = 13$	13	24	12

Line 5 changes only `prices[0]`. Changing one element never changes the others, even though `prices[1]` was originally computed from `prices[0]`; line 3 stored the number 24, not a formula.

Reread 4.3.2 *Reading and Changing Elements with an Index* if your trace differed.

Question 4

B. Sunday is index 0, Monday 1, Tuesday 2, and Wednesday 3. An element is accessed with square brackets.

- **A** is wrong: Wednesday is the fourth day, but counting from 1 instead of 0 gives the index of Thursday.
- **C** is wrong: parentheses are for method calls; elements use square brackets.
- **D** is wrong: the dot operator reaches a method or attribute by name, not an element by position.

Reread 4.3.2 *Reading and Changing Elements with an Index* if you chose differently.

Question 5

```
3.5
0.0
false
true
```

Both arrays start with defaults: every `tips` element is `0.0` and every `paid` element is `false`. Line 3 sets `tips[1]` to `3.5` and line 4 sets `paid[1]` to `true`. On line 5, `tips[3]` was never assigned, so it is still `0.0`, and `3.5 + 0.0` is `3.5`. Line 7 prints the untouched `tips[0]` (`0.0`) and line 8 the untouched `paid[0]` (`false`).

Reread 4.3.3 *New Arrays Start with Default Values* if your output differed.

Question 6

C. `String` is a reference type, and every element of a new reference-type array starts as `null`, meaning it refers to no object.

- **A** is wrong: `""` is an actual `String` object. Java does not create any `String` objects for the elements.
- **B** is wrong: `println` shows a `null` element as the letters `null`, but the element holds no string at all.
- **D** is wrong: elements always have a value, the default, so reading one is legal. (Calling a method on it would throw a `NullPointerException`.)

Reread 4.3.3 *New Arrays Start with Default Values* if you chose differently.

Question 7

```
9
13
```

Line	Work	<code>laps</code>
1	The list makes four elements	{4, 7, 2, 9}
2	<code>laps[0] + laps[3]</code> is <code>4 + 9 = 13</code> ; store at index 2	{4, 7, 13, 9}
3	Index: <code>laps[2] - 10</code> is <code>13 - 10 = 3</code> ; <code>laps[3]</code> is 9; store at index 0	{9, 7, 13, 9}

On line 3, the index expression uses the **new** value of `laps[2]` (13), because line 2 already ran. Using the original 2 would give the index `2 - 10 = -8`, which is not a position in the array at all; a trace that reaches that point should make you recheck which value the element holds now.

Reread 4.3.4 *Creating an Array with an Initializer List* and 4.3.2 *Reading and Changing Elements with an Index* if your trace differed.

Question 8

Every value in an initializer list must fit the array's element type.

- **Line 5:** 16.5 is a `double`, but `ages` is an array of `int`. Java will not silently drop the .5 (Topic 1.5), so it reports incompatible types: possible lossy conversion from `double` to `int`. If ages are whole numbers, use `int[] ages = {15, 16, 17};`. If half-years matter, change the array type: `double[] ages = {15, 16.5, 17};`.
- **Line 6:** 2 is an `int`, but `labels` is an array of `String`. Java reports incompatible types: `int` cannot be converted to `String`. Use a string: `String[] labels = {"gold", "silver", "bronze"};`.

Reread 4.3.4 *Creating an Array with an Initializer List* and 4.3.1 *Arrays Store Many Values of One Type* if you missed either error.

Question 9

```
5
12.1
13.2
```

The list has 5 values, so `times.length` is 5 and line 3 prints 5. Line 2 stores $5 - 1 = 4$ in `last`, and `times[4]` is the last value, 12.1. On line 5, `times.length / 2` is $5 / 2$, which is `int` division (Topic 1.3), giving 2; `times[2]` is 13.2, the middle value.

Reread 4.3.5 *Array Length Is Fixed* if your output differed.

Question 10

B. An array's number of elements is its `length` attribute, written without parentheses.

- **A** is wrong: `length()` with parentheses is the `String` method. Arrays have no such method, so this is a compile error (`cannot find symbol`).
- **C** is wrong: this tries to read an element at index `scores.length`, which is one past the last valid index.
- **D** is wrong: this is the index of the last element, which is one less than the number of elements.

Reread 4.3.5 *Array Length Is Fixed* if you chose differently.

Question 11

C. The array has length 8, so its valid indexes are 0 through 7. Index 8 is one past the end.

- **A** is wrong: 0 is always the first valid index of an array that has any elements.
- **B** is wrong: 7 is `length - 1`, the last valid index.
- **D** is wrong: `codes.length - 8` is $8 - 8 = 0$, a valid index.

Reread 4.3.6 *Valid Indexes and Out-of-Bounds Errors* if you chose differently.

Question 12

Model answer: The compiler checks only that each index is an `int`; it does not work out what value the index will have. Indexes often come from variables or calculations whose values are known only while the program runs, so the compiler cannot tell whether they will be in the range 0

to `length - 1`. Java checks each index when the line actually runs and throws an `ArrayIndexOutOfBoundsException` then, as `OffByOne.java` did after compiling cleanly.

A complete answer says:

- The compiler checks the type of the index, not its value.
- An out-of-range index is found only while the program runs, as an `ArrayIndexOutOfBoundsException`.

Reread 4.3.6 *Valid Indexes and Out-of-Bounds Errors* if your answer was missing one of these.

Question 13

```
8
Eli
8
```

Line	What happens	Eli's points	Fay's points
33	<code>squad[2]</code> refers to a new <code>Player</code> , Eli	0	
34	<code>squad[0]</code> refers to a new <code>Player</code> , Fay	0	0
35	The reference in <code>squad[2]</code> is copied into <code>squad[1]</code> : both name Eli	0	0
36	<code>squad[1].addPoints(7)</code> changes Eli	7	0
37	<code>squad[2].addPoints(1)</code> changes Eli again	8	0
38	<code>squad[2].getPoints()</code> is 8; Fay gets 8	8	8

Only two `Player` objects exist. `squad[1]` and `squad[2]` are aliases for Eli, so line 40 prints `Eli` and line 41 prints Eli's 8 points. If you printed 7 and 1 separately, you treated `squad[1]` and `squad[2]` as different players; line 35 copied a reference, not an object.

Reread 4.3.7 *Arrays of Object References* if your trace differed.

Question 14

A. `new Player[4]` creates one **array** object with four slots, each holding `null`. No `Player` constructor runs, so no `Player` objects exist yet.

- **B** is wrong: it counts the array as a `Player`. The array is an object, but it is an array of references, not a player.
- **C** is wrong: it assumes Java fills each slot with a new object. Slots start as `null` (the default for reference types).
- **D** is wrong: it counts four players plus the array; neither the players nor this total is right.

Reread 4.3.7 *Arrays of Object References* if you chose differently.

Question 15

C. `rates.length` is 2, so `rates.length - 1` is 1, the last valid index, and `rates[1]` is 2.0.

- **A** is wrong: the default value for `boolean` elements is `false`, not `true`.
- **B** is wrong: the list is used once, when the array is created. After that the length is fixed and cannot be increased.
- **D** is wrong: indexes start at 0, so the valid indexes are 0 through 9. Index 10 would throw an `ArrayIndexOutOfBoundsException`.

Reread 4.3.3 New Arrays Start with Default Values, 4.3.5 Array Length Is Fixed, and 4.3.6 Valid Indexes and Out-of-Bounds Errors if you chose differently.

Question 16

```
10.5
0.0
6
```

Line	Work	miles	trips
1	Four double elements, all default	{0.0, 0.0, 0.0, 0.0}	
2	List of three int values	{0.0, 0.0, 0.0, 0.0}	{2, 5, 1}
3	Index trips[2] is 1; store 3.5 at miles[1]	{0.0, 3.5, 0.0, 0.0}	{2, 5, 1}
4	miles[1] * trips.length is 3.5 × 3 = 10.5	{10.5, 3.5, 0.0, 0.0}	{2, 5, 1}
5	trips[0] + miles.length is 2 + 4 = 6	{10.5, 3.5, 0.0, 0.0}	{6, 5, 1}

Line 7 prints miles[3], which was never assigned and still holds the default 0.0. Each array has its own length: trips.length is 3 and miles.length is 4.

Reread 4.3.2 Reading and Changing Elements with an Index, 4.3.3 New Arrays Start with Default Values, and 4.3.5 Array Length Is Fixed if your trace differed.

Question 17

Line 8 uses badges[count], and count holds badges.length, which is 4. The array has 4 elements at indexes 0 through 3, so index 4 does not exist and Java throws

ArrayIndexOutOfBoundsException: Index 4 out of bounds for length 4. Line 7 already ran, which is why 4 badges appears first. The length counts elements, but the last index is one less than that.

Corrected line 8:

```
1      System.out.println("Top badge: " + badges[count - 1]);
```

Now the program prints 4 badges and then Top badge: Platinum.

Reread 4.3.6 Valid Indexes and Out-of-Bounds Errors if you missed it.

Question 18

B. Line 20 creates three null slots, and line 21 fills only shelf[0], so line 22 prints Dune. On line 23, shelf[1] is still null, and calling getTitle() on null throws a NullPointerException.

- **A** is wrong: printing shelf[1] by itself would show null, but line 23 calls a method on it, and there is no object to call it on.
- **C** is wrong: index 1 is valid for an array of length 3; the slot exists but holds no object.
- **D** is wrong: elements always have a value (here the default null), so the compiler accepts the code. The problem appears only when line 23 runs.

Reread 4.3.7 Arrays of Object References if you chose differently.

Question 19

Model answer: An array's length is set when the array is created and can never change; the length attribute is final, so scores.length = 32; is a compile error (cannot assign a value to final variable length). Writing scores = new int[32]; does compile,

but it does not enlarge the old array. It creates a separate new array of 32 elements, all 0, and makes `scores` refer to it. The 30 scores in the old array are no longer reachable through `scores`, so they would need to be copied into the new array one element at a time before the variable is changed.

A complete answer says:

- The length is fixed at creation; `length` cannot be assigned.
- `new int[32]` makes a new array with default values; the old values are not carried over.

Reread 4.3.5 *Array Length Is Fixed* if your answer was missing one of these.

Question 20

```
1
7
2
```

Work from the inside out: evaluate the inner element to get an index, then read the outer one.

Line	Inner work	Outer work	code
1			{3, 0, 2, 1}
2	<code>code[0]</code> is 3	<code>code[3]</code> is 1; prints 1	{3, 0, 2, 1}
3	Left: <code>code[3]</code> is 1. Right: <code>code[2]</code> is 2	<code>code[2] + 5</code> is 7; store at index 1	{3, 7, 2, 1}
4		<code>code[1]</code> is 7; prints 7	{3, 7, 2, 1}
5	<code>code[1] - 5</code> is 7 - 5 = 2	<code>code[2]</code> is 2; prints 2	{3, 7, 2, 1}

On line 3, both index expressions are worked out before the value is stored, so the store goes to index 1 (the value of `code[3]`).

Reread 4.3.2 *Reading and Changing Elements with an Index* if your trace differed.

Question 21

C. Segment II stores `!true`, which is `false`, in `b[1]` and prints `false`. Segment III prints `null`, the default value of a `String` element; printing `null` is fine, only calling a method on it fails. Segment I compiles but throws an `ArrayIndexOutOfBoundsException`, because an array of length 2 has valid indexes 0 and 1 only.

- **A** is wrong: segment I is the one that fails, and II and III both work.
- **B** is wrong: it leaves out III, probably by confusing printing a `null` element with calling a method on it.
- **D** is wrong: it treats index 2 as valid for an array of length 2; the last valid index is `length - 1`.

Reread 4.3.3 *New Arrays Start with Default Values* and 4.3.6 *Valid Indexes and Out-of-Bounds Errors* if you chose differently.