

4.10 Implementing ArrayList Algorithms

Self-Study Guide · About 65 minutes · With answer key

By the end you will be able to

- Find a sum, average, minimum, or maximum of the values in an ArrayList.
- Decide whether any element, or every element, of an ArrayList has a property, and count how many do.
- Compare each element with the next one, and check a list for duplicate values.
- Insert and delete elements while traversing an ArrayList without skipping elements or looping forever.
- Shift, rotate, and reverse the elements of an ArrayList.
- Build a new list from the elements of another list, and traverse two collections side by side.

What to remember from earlier

- Standard array algorithms include a sum, a maximum, a flag, and a counter. ([Topic 4.5: Implementing Array Algorithms](#))
- `add(i, value)` inserts at index `i` and moves later elements one place right. ([Topic 4.8: ArrayList Methods, 4.8.5 Inserting in the Middle](#))
- `remove(i)` deletes the element at index `i`, moves later elements left, and returns it. ([Topic 4.8: ArrayList Methods, 4.8.6 Removing an Element](#))
- An index outside 0 through `size()` minus 1 throws an `IndexOutOfBoundsException`. ([Topic 4.9: ArrayList Traversals, 4.9.2 Staying Inside the Index Range](#))
- Adding or removing during an enhanced for loop can throw a `ConcurrentModificationException`. ([Topic 4.9: ArrayList Traversals, 4.9.5 Changing the Size in an Enhanced for Loop](#))
- Java unboxes an `Integer` to an `int` on assignment or when passed as an `int`. ([Topic 4.7: Wrapper Classes, 4.7.3 Unboxing](#))

4.10.1 Sum, Average, Minimum, and Maximum

A fitness app records your step count once a day. On day 1 there is one number; after a month there are thirty. An array would need its length fixed in advance, so the app keeps the counts in an `ArrayList<Integer>` that grows by one each day. At the end of the week it wants the total, the daily average, and the best and worst days.

You already know these algorithms. In Topic 4.5 you added up an array with an accumulator, divided by the length for an average, and tracked the largest value seen so far. Every one of those algorithms works on an `ArrayList` too. Only the way you reach the elements changes:

Job	Array (Topic 4.5)	ArrayList (Topic 4.8)
How many elements	<code>arr.length</code>	<code>list.size()</code>
Read element <code>i</code>	<code>arr[i]</code>	<code>list.get(i)</code>
Replace element <code>i</code>	<code>arr[i] = value;</code>	<code>list.set(i, value);</code>

Last valid index	<code>arr.length - 1</code>	<code>list.size() - 1</code>
------------------	-----------------------------	------------------------------

Note that `length` has no parentheses but `size()` does, because `size()` is a method call. Every algorithm in this chapter is built on a **traversal** (Topic 4.9): a loop that visits the elements of a list in order. A **standard algorithm** is a common pattern, such as a sum or a maximum, that you can reuse in many programs by changing the list and the condition.

WORKED EXAMPLE Total and average steps

Watch `total` grow on each pass, then see how the average is computed.

SumAverage.java

```

1  import java.util.ArrayList;
2
3  public class SumAverage
4  {
5      public static void main(String[] args)
6      {
7          ArrayList<Integer> steps = new ArrayList<Integer>();
8          steps.add(4200);
9          steps.add(6100);
10         steps.add(3800);
11         steps.add(7500);
12         steps.add(5400);
13
14         int total = 0;
15         for (int i = 0; i < steps.size(); i++)
16         {
17             total += steps.get(i);
18         }
19         double average = (double) total / steps.size();
20         System.out.println("Total: " + total);
21         System.out.println("Average: " + average);
22     }
23 }
```

Line 17 is `total += steps.get(i);`. The call `steps.get(i)` returns an `Integer` object, and Java **unboxes** it to an `int` (Topic 4.7) so it can be added to `total`.

i	<code>steps.get(i)</code>	total after line 17
0	4200	4200
1	6100	10300
2	3800	14100
3	7500	21600
4	5400	27000

When `i` becomes 5, `5 < steps.size()` is false and the loop ends. Line 19 casts `total` to `double` before dividing, so the division keeps its decimal part: `27000.0 / 5` is `5400.0`. Without the cast, `int / int` would throw away any remainder.

Total: 27000
Average: 5400.0

An average divides by `size()`, so it needs a list with at least one element. This is a **precondition** of the algorithm: something that must be true before it runs.

For a minimum or maximum, start with a value that really is in the list, the first element, and compare every other element against it.

WORKED EXAMPLE Coldest and warmest morning

This list holds negative temperatures, so pay attention to what `lowest` and `highest` start as.

MinMaxTemps.java

```

1  import java.util.ArrayList;
2
3  public class MinMaxTemps
4  {
5      public static void main(String[] args)
6      {
7          ArrayList<Integer> temps = new ArrayList<Integer>();
8          temps.add(-3);
9          temps.add(5);
10         temps.add(-8);
11         temps.add(2);
12         temps.add(0);
13
14         int lowest = temps.get(0);
15         int highest = temps.get(0);
16         for (int i = 1; i < temps.size(); i++)
17         {
18             int temp = temps.get(i);
19             if (temp < lowest)
20             {
21                 lowest = temp;
22             }
23             if (temp > highest)
24             {
25                 highest = temp;
26             }
27         }
28         System.out.println("Lowest: " + lowest);
29         System.out.println("Highest: " + highest);
30     }
31 }
```

Lines 14 and 15 start both variables at `temps.get(0)`, which is -3. The loop then starts at index 1, because index 0 has already been used. Line 18 unboxes each element into the `int` variable `temp`.

i	temp	lowest	highest
		-3	-3
1	5	-3	5

2	-8	-8	5
3	2	-8	5
4	0	-8	5

Lowest: -8

Highest: 5

Using `get(0)` means the list must not be empty: on an empty list, `get(0)` throws an `IndexOutOfBoundsException`.

KEY IDEA

Array algorithms carry over to an `ArrayList` by swapping `arr.length` for `list.size()` and `arr[i]` for `list.get(i)`. Start a minimum or maximum at an element of the list, never at a made-up value.

WATCH OUT

It is tempting to start a maximum at 0, because 0 feels like "nothing yet." That works only when at least one value is bigger than 0.

MaxStartZero.java

```
1 import java.util.ArrayList;
2
3 public class MaxStartZero
4 {
5     public static void main(String[] args)
6     {
7         ArrayList<Integer> temps = new ArrayList<Integer>();
8         temps.add(-4);
9         temps.add(-9);
10        temps.add(-2);
11
12        int highest = 0;
13        for (int temp : temps)
14        {
15            if (temp > highest)
16            {
17                highest = temp;
18            }
19        }
20        System.out.println("Highest: " + highest);
21    }
22 }
```

Highest: 0

Every temperature is below 0, so no comparison on line 15 is ever true, and the program reports 0, a value that is not even in the list. Starting with `int highest = temps.get(0);` fixes it: the program then prints Highest: -2.

PAUSE AND TRY

Question 1 Tracing

What does this program print? Explain why it does not print `cherry`.

LongestWord.java

```
1 import java.util.ArrayList;
2
3 public class LongestWord
4 {
5     public static void main(String[] args)
6     {
7         ArrayList<String> words = new ArrayList<String>();
8         words.add("fig");
9         words.add("banana");
10        words.add("kiwi");
11        words.add("cherry");
12        words.add("plum");
13
14        String longest = words.get(0);
15        for (int i = 1; i < words.size(); i++)
16        {
17            if (words.get(i).length() > longest.length())
18            {
19                longest = words.get(i);
20            }
21        }
22        System.out.println(longest);
23    }
24 }
```

Question 2 Multiple choice

`readings` is a non-empty `ArrayList<Integer>` whose values may all be negative. A loop from index 1 to the end will update `lowest` whenever it finds a smaller value. Which declaration should come before the loop?

- A. `int lowest = 0;`
- B. `int lowest = readings.get(0);`
- C. `int lowest = readings.size();`
- D. `int lowest = readings.get(1);`

4.10.2 Any, All, and How Many

A teacher's grade list raises three different questions. Did anyone score below 70? Did everyone pass? How many students earned an A? Each is a yes-or-no question or a count about the elements of the list, and each needs a full traversal.

The tools are the ones from Topic 4.5:

- **At least one:** a boolean flag starts `false` and switches to `true` the first time an element has the property.
- **All:** a flag starts `true` and switches to `false` the first time an element does *not* have the property. One counterexample is enough to make "all" false.
- **How many:** an `int` counter starts at 0 and goes up by 1 for each element with the property.

None of these changes the list, so an **enhanced for loop** (Topic 4.9) is a good fit: it hands you each element in turn without an index. When the list holds `Integer` objects, the loop variable can be declared `int`, and Java unboxes each element as it is assigned.

WORKED EXAMPLE Three questions about one grade list

One loop can answer all three questions at once. Watch each variable change only when its own condition is true.

GradeChecks.java

```
1 import java.util.ArrayList;
2
3 public class GradeChecks
4 {
5     public static void main(String[] args)
6     {
7         ArrayList<Integer> grades = new ArrayList<Integer>();
8         grades.add(88);
9         grades.add(92);
10        grades.add(67);
11        grades.add(95);
12        grades.add(74);
13
14        boolean anyBelow70 = false;
15        boolean allPassing = true;
16        int countA = 0;
17        for (int grade : grades)
18        {
19            if (grade < 70)
20            {
21                anyBelow70 = true;
22            }
23            if (grade < 60)
24            {
25                allPassing = false;
26            }
27            if (grade >= 90)
28            {
29                countA++;
30            }
31        }
32        System.out.println("Any below 70: " + anyBelow70);
33        System.out.println("All passing: " + allPassing);
```

```
34         System.out.println("A grades: " + countA);
35     }
36 }
```

grade	anyBelow70	allPassing	countA
(start)	false	true	0
88	false	true	0
92	false	true	1
67	true	true	1
95	true	true	2
74	true	true	2

```
Any below 70: true
All passing: true
A grades: 2
```

`anyBelow70` switched once, at 67, and never switched back. `allPassing` never found a grade below 60, so it kept its starting value.

The same patterns work on a list of strings. Here the "all" flag and the counter share one `if/else`: an element either has the property (count it) or it does not (it breaks "all").

WORKED EXAMPLE Do all tags start with a hash sign?

TagChecks.java

```
1 import java.util.ArrayList;
2
3 public class TagChecks
4 {
5     public static void main(String[] args)
6     {
7         ArrayList<String> tags = new ArrayList<String>();
8         tags.add("#java");
9         tags.add("#code");
10        tags.add("fun");
11        tags.add("#ap");
12
13        boolean allTags = true;
14        int tagCount = 0;
15        for (String tag : tags)
16        {
17            if (tag.substring(0, 1).equals("#"))
18            {
19                tagCount++;
20            }
21            else
22            {
23                allTags = false;
24            }
25        }
26        System.out.println("All hashtags: " + allTags);
27        System.out.println("Hashtags: " + tagCount);
28    }
29 }
```

`tag.substring(0, 1)` is the first character of each tag (Topic 1.15), compared with `equals`. The tags `#java` and `#code` raise `tagCount` to 2. The tag `fun` goes to the `else`, so `allTags` becomes false. Then `#ap` raises `tagCount` to 3. Setting `allTags` to false a second time would change nothing, and nothing ever sets it back to true.

```
All hashtags: false
Hashtags: 3
```

WATCH OUT

A flag must never be switched back inside the loop. This version adds an `else` that resets `allPassing`:

FlagElseBug.java

```
1 import java.util.ArrayList;
2
3 public class FlagElseBug
4 {
5     public static void main(String[] args)
6     {
7         ArrayList<Integer> grades = new ArrayList<Integer>();
8         grades.add(55);
9         grades.add(80);
10        grades.add(90);
11
12        boolean allPassing = true;
13        for (int grade : grades)
14        {
15            if (grade < 60)
16            {
17                allPassing = false;
18            }
19            else
20            {
21                allPassing = true;
22            }
23        }
24        System.out.println("All passing: " + allPassing);
25    }
26 }
```

All passing: true

The 55 sets the flag to false, but the 80 sets it back to true, so the result describes only the **last** grade. Students write the `else` because it feels like "the other case." Delete the `else` block: with only the `if`, the program prints All passing: false.

PAUSE AND TRY

Question 3 Tracing

What does this program print?

CountRange.java

```
1 import java.util.ArrayList;
2
3 public class CountRange
4 {
5     public static void main(String[] args)
6     {
7         ArrayList<Integer> values = new ArrayList<Integer>();
8         values.add(12);
9         values.add(55);
10        values.add(20);
11        values.add(9);
12        values.add(31);
13        values.add(10);
14
15        int inRange = 0;
16        boolean over50 = false;
17        for (int value : values)
18        {
19            if (value >= 10 && value <= 20)
20            {
21                inRange++;
22            }
23            if (value > 50)
24            {
25                over50 = true;
26            }
27        }
28        System.out.println(inRange + " " + over50);
29    }
30 }
```

Question 4 Multiple choice

names is an ArrayList<String>. After this code runs, what does result tell you?

```
1 boolean result = true;
2 for (String name : names)
3 {
4     if (name.length() < 3)
5     {
6         result = false;
7     }
8 }
```

- A. Whether at least one name has fewer than 3 characters
- B. Whether every name has at least 3 characters
- C. How many names have at least 3 characters
- D. Whether every name has fewer than 3 characters

4.10.3 Consecutive Pairs and Duplicates

A stock-price app keeps one closing price per day. To report "the price rose 3 times this week," it has to compare each day with the **next** day. A school database storing student ID numbers has a different worry: does any ID appear twice? Both questions look at two elements at a time instead of one.

Consecutive pairs

Consecutive elements are elements next to each other, at indices `i` and `i + 1`. To visit every consecutive pair, the loop must stop one index early. In a list of 5 elements, the pairs start at indices 0, 1, 2, and 3; a pair starting at index 4 would need index 5, which does not exist. So the condition is `i < list.size() - 1`.

WORKED EXAMPLE Counting price rises

ConsecutiveRises.java

```

1  import java.util.ArrayList;
2
3  public class ConsecutiveRises
4  {
5      public static void main(String[] args)
6      {
7          ArrayList<Integer> prices = new ArrayList<Integer>();
8          prices.add(20);
9          prices.add(22);
10         prices.add(21);
11         prices.add(25);
12         prices.add(30);
13
14         int rises = 0;
15         int biggestJump = 0;
16         for (int i = 0; i < prices.size() - 1; i++)
17         {
18             int today = prices.get(i);
19             int tomorrow = prices.get(i + 1);
20             if (tomorrow > today)
21             {
22                 rises++;
23                 if (tomorrow - today > biggestJump)
24                 {
25                     biggestJump = tomorrow - today;
26                 }
27             }
28         }
29         System.out.println("Rises: " + rises);
30         System.out.println("Biggest jump: " + biggestJump);
31     }
32 }
```

Lines 18 and 19 copy the pair into two `int` variables, which unboxes them and gives them readable names.

i	today	tomorrow	Rise?	rises	biggestJump
0	20	22	yes	1	2
1	22	21	no	1	2

2	21	25	yes	2	4
3	25	30	yes	3	5

At `i = 4`, the condition `4 < 4` is false, so `get(5)` is never called.

Rises: 3
Biggest jump: 5

WATCH OUT

Writing the usual `i < prices.size()` in a pair loop goes one step too far:

PairsTooFar.java

```

1  import java.util.ArrayList;
2
3  public class PairsTooFar
4  {
5      public static void main(String[] args)
6      {
7          ArrayList<Integer> prices = new ArrayList<Integer>();
8          prices.add(20);
9          prices.add(22);
10         prices.add(21);
11         prices.add(25);
12         prices.add(30);
13
14         for (int i = 0; i < prices.size(); i++)
15         {
16             System.out.println(prices.get(i) + " -> " + prices.get(i + 1));
17         }
18     }
19 }
```

```

20 -> 22
22 -> 21
21 -> 25
25 -> 30
Exception in thread "main" java.lang.IndexOutOfBoundsException: Index 5 out of
bounds for length 5
```

The four real pairs print, then at `i = 4` the call `prices.get(i + 1)` asks for index 5 and the program crashes. (The exact message wording can vary between Java versions.) The fix is `i < prices.size() - 1`. Another correct form starts at `i = 1` and compares `get(i - 1)` with `get(i)`.

Duplicates

To find out whether any value appears twice, every element must be compared with every other element, not only its neighbor, since the two copies could be far apart. That calls for **nested loops** (Topic 2.11): the outer loop picks an element at index `i`, and the inner loop compares it with every

element **after** it, starting at $j = i + 1$. Starting after i means an element is never compared with itself, and each pair is checked only once.

WORKED EXAMPLE Is any ID repeated?

HasDuplicate.java

```

1  import java.util.ArrayList;
2
3  public class HasDuplicate
4  {
5      public static void main(String[] args)
6      {
7          ArrayList<Integer> ids = new ArrayList<Integer>();
8          ids.add(104);
9          ids.add(317);
10         ids.add(250);
11         ids.add(317);
12         ids.add(999);
13
14         boolean duplicate = false;
15         for (int i = 0; i < ids.size(); i++)
16         {
17             for (int j = i + 1; j < ids.size(); j++)
18             {
19                 int first = ids.get(i);
20                 int second = ids.get(j);
21                 if (first == second)
22                 {
23                     duplicate = true;
24                     System.out.println("Match at " + i + " and " + j);
25                 }
26             }
27         }
28         System.out.println("Has duplicate: " + duplicate);
29     }
30 }

```

The comparisons run like this. Only one is a match.

i	j values checked	Match?
0	1, 2, 3, 4	no (104 appears once)
1	2, 3, 4	at j = 3: 317 and 317
2	3, 4	no
3	4	no
4	none	inner loop does not run

```

Match at 1 and 3
Has duplicate: true

```

Five elements take $4 + 3 + 2 + 1 = 10$ comparisons. If no match had been found, `duplicate` would have stayed false, which is how the algorithm reports the absence of duplicates.

KEY IDEA

A pair loop stops at `i < list.size() - 1` so `get(i + 1)` stays in range. A duplicate check compares each element with every later element, using an inner loop that starts at `j = i + 1`.

WATCH OUT

Lines 19 and 20 of `HasDuplicate` copy the elements into `int` variables before comparing them. Skipping that step looks harmless, but it changes what `==` compares:

`IntegerEqualsBug.java`

```
1 import java.util.ArrayList;
2
3 public class IntegerEqualsBug
4 {
5     public static void main(String[] args)
6     {
7         ArrayList<Integer> ids = new ArrayList<Integer>();
8         ids.add(1000);
9         ids.add(2000);
10        ids.add(1000);
11
12        boolean duplicate = false;
13        for (int i = 0; i < ids.size(); i++)
14        {
15            for (int j = i + 1; j < ids.size(); j++)
16            {
17                if (ids.get(i) == ids.get(j))
18                {
19                    duplicate = true;
20                }
21            }
22        }
23        System.out.println("Has duplicate: " + duplicate);
24    }
25 }
```

Has duplicate: false

`ids.get(i)` and `ids.get(j)` are both `Integer` **objects**, so `==` compares their references (Topic 2.6) and asks whether they are the same object. The two 1000s are equal values stored in different objects, so the answer is false. (For small numbers Java may reuse one object, so this bug can stay hidden until the values grow.) Unbox first, with `int first = ids.get(i);` and `int second = ids.get(j);`, and compare the `int` values; that version prints `Has duplicate: true`. For lists of `String`, compare with `equals`.

PAUSE AND TRY**Question 5** Debugging

codes holds 4, 8, and 15, so there are no duplicates, but this program prints Has duplicate: true. Find the error and write the corrected line.

DupSelfBug.java

```
1 import java.util.ArrayList;
2
3 public class DupSelfBug
4 {
5     public static void main(String[] args)
6     {
7         ArrayList<Integer> codes = new ArrayList<Integer>();
8         codes.add(4);
9         codes.add(8);
10        codes.add(15);
11
12        boolean duplicate = false;
13        for (int i = 0; i < codes.size(); i++)
14        {
15            for (int j = 0; j < codes.size(); j++)
16            {
17                int first = codes.get(i);
18                int second = codes.get(j);
19                if (first == second)
20                {
21                    duplicate = true;
22                }
23            }
24        }
25        System.out.println("Has duplicate: " + duplicate);
26    }
27 }
```

Question 6 Tracing

What does this program print? Give the comparison made for each value of *i*.

AdjacentRepeats.java

```
1 import java.util.ArrayList;
2
3 public class AdjacentRepeats
4 {
5     public static void main(String[] args)
6     {
7         ArrayList<String> words = new ArrayList<String>();
8         words.add("the");
9         words.add("the");
10        words.add("cat");
11        words.add("sat");
12        words.add("sat");
13        words.add("sat");
14
15        int repeats = 0;
16        for (int i = 1; i < words.size(); i++)
17        {
18            if (words.get(i).equals(words.get(i - 1)))
19            {
20                repeats++;
21            }
22        }
23        System.out.println(repeats);
24    }
25 }
```

4.10.4 Inserting While Traversing

A race app keeps finish times in order, fastest first. When a new runner finishes, the time has to go into the right place, not at the end. With an array this was hard, because an array's length cannot change. An `ArrayList` can grow, and its `add(index, value)` method from Topic 4.8 makes room by moving the element at `index`, and every element after it, one place to the right.

To **insert** in order, traverse until you find the first element bigger than the new value, then add the new value at that index.

WORKED EXAMPLE Inserting into a sorted list

The method `insertInOrder` takes the list as a parameter. As in Topic 3.6, the parameter refers to the same `ArrayList` object as `times` in `main`, so the insertion changes the caller's list. Printing an `ArrayList` with `println` shows its elements in order, inside square brackets, separated by commas.

InsertInOrder.java

```
1 import java.util.ArrayList;
2
3 public class InsertInOrder
4 {
5     public static void insertInOrder(ArrayList<Integer> list, int value)
```

```

6      {
7          int index = 0;
8          while (index < list.size() && list.get(index) < value)
9          {
10             index++;
11         }
12         list.add(index, value);
13     }
14
15     public static void main(String[] args)
16     {
17         ArrayList<Integer> times = new ArrayList<Integer>();
18         times.add(10);
19         times.add(25);
20         times.add(40);
21         times.add(55);
22
23         System.out.println(times);
24         insertInOrder(times, 30);
25         System.out.println(times);
26         insertInOrder(times, 70);
27         System.out.println(times);
28         insertInOrder(times, 5);
29         System.out.println(times);
30     }
31 }

```

Inserting 30 into [10, 25, 40, 55]: line 8 moves `index` forward while the element there is less than 30.

index	index < 4?	list.get(index) < 30?	Action
0	true	10, true	index = 1
1	true	25, true	index = 2
2	true	40, false	loop ends

Line 12 runs `add(2, 30)`: 40 and 55 each move one place right, and 30 lands at index 2.

Inserting 70: every element is less than 70, so `index` climbs to 5, the size of the list. Now `5 < 5` is false, and because `&&` short-circuits (Topic 2.5), `list.get(5)` is never evaluated. `add(5, 70)` is allowed: an insertion index may go from 0 up to and including `size()`, and `size()` means "at the end."

Inserting 5: `10 < 5` is false right away, so `add(0, 5)` puts 5 at the front.

```

[10, 25, 40, 55]
[10, 25, 30, 40, 55]
[10, 25, 30, 40, 55, 70]
[5, 10, 25, 30, 40, 55, 70]

```

Inserting **during** a traversal needs one more step. Say a to-do list should get a `study` task in front of every `test`. When the loop finds `test` at index `i` and inserts `study` there, `test` moves to index `i + 1`. The loop's own `i++` would land right back on that same `test`, insert again, and repeat

forever. The fix is an extra `i++` after inserting, which moves past the new element so the loop's `i++` then moves past the test.

WORKED EXAMPLE Adding a study task before each test

InsertStudy.java

```

1  import java.util.ArrayList;
2
3  public class InsertStudy
4  {
5      public static void main(String[] args)
6      {
7          ArrayList<String> tasks = new ArrayList<String>();
8          tasks.add("laundry");
9          tasks.add("test");
10         tasks.add("dishes");
11         tasks.add("test");
12
13         for (int i = 0; i < tasks.size(); i++)
14         {
15             if (tasks.get(i).equals("test"))
16             {
17                 tasks.add(i, "study");
18                 i++;
19             }
20         }
21         System.out.println(tasks);
22     }
23 }
```

i at check	tasks.get(i)	Action	List afterward
0	laundry	none	[laundry, test, dishes, test]
1	test	insert at 1; i becomes 2	[laundry, study, test, dishes, test]
3	dishes	none	(same)
4	test	insert at 4; i becomes 5	[laundry, study, test, dishes, study, test]

After the second insertion, the loop's `i++` makes `i` 6, and `6 < 6` is false. Notice that line 13 calls `tasks.size()` on every check. The list grew from 4 to 6 elements, and the loop kept up because it asked for the current size each time.

```
[laundry, study, test, dishes, study, test]
```

WATCH OUT

Without the extra `i++` on line 18, the loop never ends:

```
1 for (int i = 0; i < tasks.size(); i++)
2 {
3     if (tasks.get(i).equals("test"))
4     {
5         tasks.add(i, "study");
6     }
7 }
```

At `i = 1` it inserts `study`, and `test` moves to index 2. The loop's `i++` makes `i` 2, which finds the same `test` again, inserts again, and so on. The list keeps growing and the program never finishes on its own. When you insert at or before the current index, step `i` past what you inserted.

PAUSE AND TRY

Question 7 Tracing

What does this program print? Show the list after each insertion.

InsertAfterEven.java

```
1 import java.util.ArrayList;
2
3 public class InsertAfterEven
4 {
5     public static void main(String[] args)
6     {
7         ArrayList<Integer> nums = new ArrayList<Integer>();
8         nums.add(3);
9         nums.add(4);
10        nums.add(7);
11        nums.add(8);
12
13        for (int i = 0; i < nums.size(); i++)
14        {
15            if (nums.get(i) % 2 == 0)
16            {
17                nums.add(i + 1, 0);
18                i++;
19            }
20        }
21        System.out.println(nums);
22    }
23 }
```

Question 8 Multiple choice

In the program from Question 7, suppose the `i++` inside the `if` (line 18) is deleted. What happens?

- A. The output is the same, `[3, 4, 0, 7, 8, 0]`
- B. A 0 is inserted after every element: `[3, 0, 4, 0, 7, 0, 8, 0]`
- C. The loop never ends, because each inserted 0 is itself even

D. An `IndexOutOfBoundsException` is thrown at `add(i + 1, 0)`

4.10.5 Deleting While Traversing

The teacher from 4.10.2 wants to drop every score below 60 from a list before computing a class average. Removing elements one at a time during a traversal sounds simple, but `remove(i)` from Topic 4.8 moves every later element one place **left**. The element that was at `i + 1` is now at `i`, and a loop that goes on to `i + 1` never looks at it.

WORKED EXAMPLE The skipped element

This code tries to remove every score below 60 from `[55, 48, 90, 30]`. The correct answer is `[90]`.

RemoveSkipBug.java

```
1 import java.util.ArrayList;
2
3 public class RemoveSkipBug
4 {
5     public static void main(String[] args)
6     {
7         ArrayList<Integer> scores = new ArrayList<Integer>();
8         scores.add(55);
9         scores.add(48);
10        scores.add(90);
11        scores.add(30);
12
13        for (int i = 0; i < scores.size(); i++)
14        {
15            if (scores.get(i) < 60)
16            {
17                scores.remove(i);
18            }
19        }
20        System.out.println(scores);
21    }
22 }
```

i	scores.get(i)	Action	List afterward
0	55	remove 0	[48, 90, 30]
1	90	none	[48, 90, 30]
2	30	remove 2	[48, 90]

[48, 90]

When 55 was removed, 48 slid into index 0. The loop then moved on to index 1, so 48 was never checked. No error appears; the answer is just wrong, which makes this bug easy to miss. It strikes whenever two elements to delete sit next to each other.

There are two standard fixes. The first moves forward but adds 1 to the index **only when nothing was removed**. A while loop makes that easy to write.

RemoveForwardFixed.java

```

1  import java.util.ArrayList;
2
3  public class RemoveForwardFixed
4  {
5      public static void main(String[] args)
6      {
7          ArrayList<Integer> scores = new ArrayList<Integer>();
8          scores.add(55);
9          scores.add(48);
10         scores.add(90);
11         scores.add(30);
12
13         int i = 0;
14         while (i < scores.size())
15         {
16             if (scores.get(i) < 60)
17             {
18                 scores.remove(i);
19             }
20             else
21             {
22                 i++;
23             }
24         }
25         System.out.println(scores);
26     }
27 }

```

i	scores.get(i)	Action	List afterward
0	55	remove 0	[48, 90, 30]
0	48	remove 0	[90, 30]
0	90	i becomes 1	[90, 30]
1	30	remove 1	[90]

The check at `i = 1` then finds `1 < 1` false. The program prints `[90]`.

The second fix **traverses backward**, from the last index down to 0. Removing at `i` moves only the elements to the **right** of `i`, and those have already been checked, so nothing that still needs checking moves.

RemoveBackward.java

```

1  import java.util.ArrayList;
2
3  public class RemoveBackward
4  {
5      public static void main(String[] args)
6      {
7          ArrayList<Integer> scores = new ArrayList<Integer>();
8          scores.add(55);
9          scores.add(48);
10         scores.add(90);
11         scores.add(30);
12
13         for (int i = scores.size() - 1; i >= 0; i--)
14         {
15             if (scores.get(i) < 60)
16             {
17                 scores.remove(i);
18             }
19         }
20         System.out.println(scores);
21     }
22 }

```

i	scores.get(i)	Action	List afterward
3	30	remove 3	[55, 48, 90]
2	90	none	[55, 48, 90]
1	48	remove 1	[55, 90]
0	55	remove 0	[90]

[90]

KEY IDEA

After `remove(i)`, the next element slides into index `i`. To delete during a traversal, either add 1 to `i` only when nothing was removed, or traverse backward from `size() - 1` to 0.

WATCH OUT

An enhanced for loop cannot be used to delete, even with your own index counter, as Topic 4.9 warned:

RemoveEnhancedFor.java

```
1 import java.util.ArrayList;
2
3 public class RemoveEnhancedFor
4 {
5     public static void main(String[] args)
6     {
7         ArrayList<Integer> scores = new ArrayList<Integer>();
8         scores.add(55);
9         scores.add(90);
10        scores.add(48);
11
12        int index = 0;
13        for (int score : scores)
14        {
15            if (score < 60)
16            {
17                scores.remove(index);
18            }
19            index++;
20        }
21        System.out.println(scores);
22    }
23 }
```

Exception in thread "main" java.util.ConcurrentModificationException

The 55 is removed on the first pass. When the loop asks for the next element, Java notices that the list changed size during the loop and throws a `ConcurrentModificationException`, so nothing prints. The same is true of adding. Use an indexed `for` or `while` loop whenever the list's size changes.

PAUSE AND TRY**Question 9** Tracing

What does this program print? Show the list after each removal.

RemoveShortWords.java

```
1 import java.util.ArrayList;
2
3 public class RemoveShortWords
4 {
5     public static void main(String[] args)
6     {
7         ArrayList<String> words = new ArrayList<String>();
8         words.add("a");
9         words.add("be");
10        words.add("cat");
11        words.add("do");
12        words.add("eagle");
13
14        for (int i = words.size() - 1; i >= 0; i--)
15        {
16            if (words.get(i).length() < 3)
17            {
18                words.remove(i);
19            }
20        }
21        System.out.println(words);
22    }
23 }
```

Question 10 Multiple choice

Each segment below starts with its list holding [2, 4, 5, 6] and is meant to remove every even number, leaving [5]. Which segments work?

```
1 // I
2 for (int i = 0; i < list.size(); i++)
3 {
4     if (list.get(i) % 2 == 0)
5     {
6         list.remove(i);
7     }
8 }
9
10 // II
11 for (int i = list.size() - 1; i >= 0; i--)
12 {
13     if (list.get(i) % 2 == 0)
14     {
15         list.remove(i);
16     }
17 }
18
19 // III
20 int k = 0;
21 while (k < list.size())
22 {
23     if (list.get(k) % 2 == 0)
```

```
24     {  
25         list.remove(k);  
26     }  
27     else  
28     {  
29         k++;  
30     }  
31 }
```

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

4.10.6 Shifting, Rotating, and Reversing

A weather display shows the last four temperature readings. When a new reading arrives, the oldest one drops off the front and the new one joins at the back. A board game passes the turn around a table, and after the last player it is the first player's turn again. In Topic 4.5, moving every element of an array one place took a loop that copied element after element. With an `ArrayList`, `remove` and `add` do the moving for you.

Two terms describe these moves:

- To **shift** a list is to move every element one place left or right. One element falls off one end, and a new value fills the empty spot at the other end.
- To **rotate** a list is to shift it with wraparound: the element that falls off one end comes back in at the other end, so no element is lost.

WORKED EXAMPLE Shifting a window of readings

ShiftWindow.java

```
1 import java.util.ArrayList;
2
3 public class ShiftWindow
4 {
5     public static void main(String[] args)
6     {
7         ArrayList<Integer> window = new ArrayList<Integer>();
8         window.add(3);
9         window.add(5);
10        window.add(4);
11        window.add(6);
12
13        System.out.println(window);
14        window.remove(0);
15        window.add(8);
16        System.out.println(window);
17        window.remove(window.size() - 1);
18        window.add(0, 0);
19        System.out.println(window);
20    }
21 }
```

- **Shift left** (lines 14–15): `remove(0)` drops the oldest reading, 3, and moves the rest left. `add(8)` puts the new reading at the end. The list still has 4 elements.
- **Shift right** (lines 17–18): `remove(window.size() - 1)` drops the last element, 8. `add(0, 0)` inserts the filler value 0 at the front, which moves everything else right.

```
[3, 5, 4, 6]
[5, 4, 6, 8]
[0, 5, 4, 6]
```

A rotation uses the fact that `remove` **returns** the element it removed (Topic 4.8). Save it, then add it back at the other end.

WORKED EXAMPLE Passing the turn

RotateTurns.java

```
1 import java.util.ArrayList;
2
3 public class RotateTurns
4 {
5     public static void main(String[] args)
6     {
7         ArrayList<String> turns = new ArrayList<String>();
8         turns.add("Ana");
9         turns.add("Ben");
10        turns.add("Cy");
11        turns.add("Dee");
12
13        System.out.println(turns);
14        String first = turns.remove(0);
15        turns.add(first);
16        System.out.println(turns);
17        String last = turns.remove(turns.size() - 1);
18        turns.add(0, last);
19        System.out.println(turns);
20    }
21 }
```

- **Rotate left** (lines 14–15): `remove(0)` takes Ana off the front and returns her name, which is stored in `first`. `add(first)` puts her at the back.
- **Rotate right** (lines 17–18): `remove(turns.size() - 1)` takes Ana off the back, and `add(0, last)` puts her at the front again, undoing the left rotation.

```
[Ana, Ben, Cy, Dee]
[Ben, Cy, Dee, Ana]
[Ana, Ben, Cy, Dee]
```

The two steps can also be written as one statement, `turns.add(turns.remove(0));`. Java runs the inner call first, and the element it returns becomes the argument of `add`.

To **reverse** a list in place, swap the first and last elements, then the second and second-to-last, and so on toward the middle. A swap needs a temporary variable, just as with arrays in Topic 4.5, because the first `set` overwrites one of the two values.

WORKED EXAMPLE Reversing a list by swapping

ReverseList.java

```

1  import java.util.ArrayList;
2
3  public class ReverseList
4  {
5      public static void main(String[] args)
6      {
7          ArrayList<Integer> nums = new ArrayList<Integer>();
8          nums.add(1);
9          nums.add(2);
10         nums.add(3);
11         nums.add(4);
12         nums.add(5);
13
14         int last = nums.size() - 1;
15         for (int i = 0; i < nums.size() / 2; i++)
16         {
17             int temp = nums.get(i);
18             nums.set(i, nums.get(last - i));
19             nums.set(last - i, temp);
20         }
21         System.out.println(nums);
22     }
23 }

```

last is 4. The partner of index i is $\text{last} - i$: 0 pairs with 4, and 1 pairs with 3.

i	temp	After set(i, ...)	After set(last - i, temp)
0	1	[5, 2, 3, 4, 5]	[5, 2, 3, 4, 1]
1	2	[5, 4, 3, 4, 1]	[5, 4, 3, 2, 1]

At $i = 2$ the check $2 < 5 / 2$ is $2 < 2$, false, so the loop ends. The middle element, 3, has no partner and stays put. For a list of 6, $6 / 2$ is 3 and the loop swaps three pairs.

[5, 4, 3, 2, 1]

WATCH OUT

A reverse loop that runs over the **whole** list swaps every pair twice:

ReverseTwice.java

```
1 import java.util.ArrayList;
2
3 public class ReverseTwice
4 {
5     public static void main(String[] args)
6     {
7         ArrayList<Integer> nums = new ArrayList<Integer>();
8         nums.add(1);
9         nums.add(2);
10        nums.add(3);
11        nums.add(4);
12        nums.add(5);
13
14        int last = nums.size() - 1;
15        for (int i = 0; i < nums.size(); i++)
16        {
17            int temp = nums.get(i);
18            nums.set(i, nums.get(last - i));
19            nums.set(last - i, temp);
20        }
21        System.out.println(nums);
22    }
23 }
```

[1, 2, 3, 4, 5]

By the end of `i = 1` the list is reversed. Then `i = 2` swaps the middle with itself, `i = 3` swaps indices 3 and 1 back, and `i = 4` swaps indices 4 and 0 back, restoring the original order. Students write `i < nums.size()` from habit, because most traversals use it. A reverse must stop halfway, at `i < nums.size() / 2`.

PAUSE AND TRY

Question 11 Tracing

What does this program print? Show the list after each pass.

RotateLoop.java

```
1 import java.util.ArrayList;
2
3 public class RotateLoop
4 {
5     public static void main(String[] args)
6     {
7         ArrayList<Integer> nums = new ArrayList<Integer>();
8         nums.add(1);
9         nums.add(2);
10        nums.add(3);
11        nums.add(4);
12        nums.add(5);
13
14        for (int k = 0; k < 2; k++)
15        {
16            nums.add(0, nums.remove(nums.size() - 1));
17        }
18        System.out.println(nums);
19    }
20 }
```

Question 12 Tracing

This reverse leaves out the temporary variable. What does it print?

SwapNoTemp.java

```
1 import java.util.ArrayList;
2
3 public class SwapNoTemp
4 {
5     public static void main(String[] args)
6     {
7         ArrayList<Integer> nums = new ArrayList<Integer>();
8         nums.add(10);
9         nums.add(20);
10        nums.add(30);
11        nums.add(40);
12        nums.add(50);
13        nums.add(60);
14
15        int last = nums.size() - 1;
16        for (int i = 0; i < nums.size() / 2; i++)
17        {
18            nums.set(i, nums.get(last - i));
19            nums.set(last - i, nums.get(i));
20        }
21        System.out.println(nums);
22    }
23 }
```

4.10.7 Building a Filtered List

Deleting from a list changes it for good. Often a program needs the original data later, such as every score for the class average **and** a separate list of just the failing scores to send home. Instead of removing elements from the original, you can build a second list.

The pattern: create a new, empty `ArrayList`, traverse the original, and add each element that has the property. The result is a **filtered list**. The original is only read, never changed, so an enhanced for loop is safe, and no element can be skipped.

WORKED EXAMPLE Fruits that come before m

BeforeM.java

```

1  import java.util.ArrayList;
2
3  public class BeforeM
4  {
5      public static void main(String[] args)
6      {
7          ArrayList<String> fruits = new ArrayList<String>();
8          fruits.add("apple");
9          fruits.add("kiwi");
10         fruits.add("mango");
11         fruits.add("fig");
12         fruits.add("peach");
13
14         ArrayList<String> early = new ArrayList<String>();
15         for (String fruit : fruits)
16         {
17             if (fruit.compareTo("m") < 0)
18             {
19                 early.add(fruit);
20             }
21         }
22         System.out.println(early);
23         System.out.println(fruits.size() + " " + early.size());
24     }
25 }
```

`compareTo` (Topic 1.15) returns a negative number when the first string comes before the second in dictionary order.

fruit	fruit.compareTo("m") < 0?	early afterward
apple	yes	[apple]
kiwi	yes	[apple, kiwi]
mango	no	[apple, kiwi]
fig	yes	[apple, kiwi, fig]
peach	no	[apple, kiwi, fig]

`mango` starts with `m`, but it is longer than `"m"`, so it comes **after** `"m"` in dictionary order. The last line shows that `fruits` still has all 5 elements.

```
[apple, kiwi, fig]
5 3
```

WATCH OUT

Adding to the wrong list is a one-word mistake with a loud result:

FilterAddBug.java

```
1 import java.util.ArrayList;
2
3 public class FilterAddBug
4 {
5     public static void main(String[] args)
6     {
7         ArrayList<Integer> nums = new ArrayList<Integer>();
8         nums.add(4);
9         nums.add(12);
10        nums.add(7);
11        nums.add(15);
12
13        ArrayList<Integer> big = new ArrayList<Integer>();
14        for (int n : nums)
15        {
16            if (n > 10)
17            {
18                nums.add(n);
19            }
20        }
21        System.out.println(big);
22    }
23 }
```

```
Exception in thread "main" java.util.ConcurrentModificationException
```

Line 18 adds to `nums`, the list being traversed, instead of `big`. When 12 is added, `nums` changes size in the middle of the enhanced for loop, and the next step of the loop throws. Change line 18 to `big.add(n);` and the program prints `[12, 15]`.

PAUSE AND TRY**Question 13** Short answer

A student wants the scores below 60 in their own list, and also needs the full list of scores for an average afterward. They plan to copy the failing scores into a new list and then delete them from the original. Explain why building the filtered list alone is the better plan here, and name one mistake the deleting step could add.

4.10.8 Traversing Several Collections at Once

A class keeps student names in an array and their GPAs in an `ArrayList<Double>`, with the GPA at index `i` belonging to the name at index `i`. To print the honor roll, the program has to read **both** collections together. Some algorithms need several strings, arrays, or lists traversed at the same time.

When the collections line up by position, one loop index can serve all of them: `names[i]` and `gpas.get(i)` refer to the same student. The one rule is that the loop bound must keep `i` valid in **every** collection it reads.

WORKED EXAMPLE Building an honor roll from two collections

HonorRoll.java

```

1  import java.util.ArrayList;
2
3  public class HonorRoll
4  {
5      public static void main(String[] args)
6      {
7          String[] names = {"Ava", "Ben", "Cleo", "Dev"};
8          ArrayList<Double> gpas = new ArrayList<Double>();
9          gpas.add(3.8);
10         gpas.add(3.2);
11         gpas.add(3.9);
12         gpas.add(3.5);
13
14         ArrayList<String> honorRoll = new ArrayList<String>();
15         for (int i = 0; i < names.length; i++)
16         {
17             if (gpas.get(i) >= 3.5)
18             {
19                 honorRoll.add(names[i]);
20             }
21         }
22         System.out.println(honorRoll);
23         System.out.println(gpas);
24     }
25 }
```

The array and the list both have 4 elements, so `i < names.length` keeps `i` in range for both. Line 17 reads a `Double` from the list and compares it with 3.5; line 19 adds the matching name to a new list.

i	names[i]	gpas.get(i)	3.5 or higher?	honorRoll afterward
0	Ava	3.8	yes	[Ava]
1	Ben	3.2	no	[Ava]
2	Cleo	3.9	yes	[Ava, Cleo]
3	Dev	3.5	yes	[Ava, Cleo, Dev]

```

[Ava, Cleo, Dev]
[3.8, 3.2, 3.9, 3.5]
```

A string can be one of the collections, too. Here a quiz's answer key is a `String`, one letter per question, and a student's answers are in an `ArrayList<String>`.

WORKED EXAMPLE Grading a quiz against a key

QuizKey.java

```

1  import java.util.ArrayList;
2
3  public class QuizKey
4  {
5      public static void main(String[] args)
6      {
7          String key = "BDAC";
8          ArrayList<String> answers = new ArrayList<String>();
9          answers.add("B");
10         answers.add("A");
11         answers.add("A");
12         answers.add("C");
13
14         int correct = 0;
15         ArrayList<Integer> missed = new ArrayList<Integer>();
16         for (int i = 0; i < answers.size(); i++)
17         {
18             if (answers.get(i).equals(key.substring(i, i + 1)))
19             {
20                 correct++;
21             }
22             else
23             {
24                 missed.add(i + 1);
25             }
26         }
27         System.out.println("Correct: " + correct);
28         System.out.println("Missed: " + missed);
29     }
30 }

```

On each pass, `answers.get(i)` is compared with `key.substring(i, i + 1)`, the letter at index `i` of the key. A match adds to `correct`; a miss adds the question number, `i + 1`, to the new list `missed`.

i	Student	Key	Result
0	B	B	correct = 1
1	A	D	missed = [2]
2	A	A	correct = 2
3	C	C	correct = 3

Correct: 3
Missed: [2]

KEY IDEA

To traverse collections side by side, use one index for all of them, and bound the loop by a collection that every index is valid for.

WATCH OUT

If the collections have different lengths, the bound matters. This student skipped the last question, so the list has only 3 answers, but the loop is bounded by the key:

ShortAnswersBug.java

```
1 import java.util.ArrayList;
2
3 public class ShortAnswersBug
4 {
5     public static void main(String[] args)
6     {
7         String key = "BDAC";
8         ArrayList<String> answers = new ArrayList<String>();
9         answers.add("B");
10        answers.add("A");
11        answers.add("A");
12
13        for (int i = 0; i < key.length(); i++)
14        {
15            if (answers.get(i).equals(key.substring(i, i + 1)))
16            {
17                System.out.println("Question " + (i + 1) + ": right");
18            }
19            else
20            {
21                System.out.println("Question " + (i + 1) + ": wrong");
22            }
23        }
24    }
25 }
```

Question 1: right

Question 2: wrong

Question 3: right

Exception in thread "main" java.lang.IndexOutOfBoundsException: Index 3 out of bounds for length 3

At `i = 3` the key still has a letter, but `answers.get(3)` does not exist. Bound the loop by the shorter collection, `i < answers.size()`; then the program prints the three lines and stops normally. (When either one could be shorter, `Math.min(key.length(), answers.size())` from Topic 1.11 works as the bound.)

PAUSE AND TRY**Question 14** Multiple choice

What does this program print?

LargerOfPairs.java

```
1 import java.util.ArrayList;
2
3 public class LargerOfPairs
4 {
5     public static void main(String[] args)
6     {
7         ArrayList<Integer> a = new ArrayList<Integer>();
8         a.add(3);
9         a.add(8);
10        a.add(2);
11        ArrayList<Integer> b = new ArrayList<Integer>();
12        b.add(5);
13        b.add(1);
14        b.add(2);
15
16        ArrayList<Integer> larger = new ArrayList<Integer>();
17        for (int i = 0; i < a.size(); i++)
18        {
19            larger.add(Math.max(a.get(i), b.get(i)));
20        }
21        System.out.println(larger);
22    }
23 }
```

- A. [5, 8, 2]
- B. [8]
- C. [3, 1, 2]
- D. [5, 8, 2, 2]

Putting it together

A classroom sensor records a reading every minute and stores -1 whenever it fails to read. Before the data can be used, the bad readings must go. Then the program reports the average, the readings above average, and whether any reading repeated two minutes in a row. Each job is a separate method, and each method is one algorithm from this chapter.

CleanReadings.java

```
1 import java.util.ArrayList;
2
3 public class CleanReadings
4 {
5     public static void removeInvalid(ArrayList<Integer> list)
6     {
7         for (int i = list.size() - 1; i >= 0; i--)
```

```
8      {
9          if (list.get(i) < 0)
10         {
11             list.remove(i);
12         }
13     }
14 }
15
16 public static double average(ArrayList<Integer> list)
17 {
18     int total = 0;
19     for (int value : list)
20     {
21         total += value;
22     }
23     return (double) total / list.size();
24 }
25
26 public static ArrayList<Integer> aboveAverage(ArrayList<Integer> list, double
avg)
27 {
28     ArrayList<Integer> result = new ArrayList<Integer>();
29     for (int value : list)
30     {
31         if (value > avg)
32         {
33             result.add(value);
34         }
35     }
36     return result;
37 }
38
39 public static boolean hasRepeatInARow(ArrayList<Integer> list)
40 {
41     for (int i = 0; i < list.size() - 1; i++)
42     {
43         int current = list.get(i);
44         int next = list.get(i + 1);
45         if (current == next)
46         {
47             return true;
48         }
49     }
50     return false;
51 }
52
53 public static void main(String[] args)
54 {
55     ArrayList<Integer> readings = new ArrayList<Integer>();
56     readings.add(12);
57     readings.add(-1);
58     readings.add(15);
59     readings.add(15);
60     readings.add(-1);
61     readings.add(20);
```

```

62         readings.add(9);
63
64         removeInvalid(readings);
65         System.out.println("Cleaned: " + readings);
66         double avg = average(readings);
67         System.out.println("Average: " + avg);
68         System.out.println("Above average: " + aboveAverage(readings, avg));
69         System.out.println("Repeat in a row: " + hasRepeatInARow(readings));
70     }
71 }

```

Step 1, removeInvalid (lines 5–14), deletes while traversing (4.10.5). It walks backward and removes every negative reading. The list `readings` in `main` is changed, because the parameter list refers to the same object (Topic 3.6).

i	list.get(i)	Action	List afterward
6	9	none	[12, -1, 15, 15, -1, 20, 9]
5	20	none	(same)
4	-1	remove 4	[12, -1, 15, 15, 20, 9]
3	15	none	(same)
2	15	none	(same)
1	-1	remove 1	[12, 15, 15, 20, 9]
0	12	none	[12, 15, 15, 20, 9]

Step 2, average (lines 16–24), sums and divides (4.10.1). $12 + 15 + 15 + 20 + 9 = 71$, and $71.0 / 5$ is 14.2. Running this step *after* the cleanup matters: with the two -1s still in, the average would be wrong.

Step 3, aboveAverage (lines 26–37), builds a filtered list (4.10.7). It returns a new list holding 15, 15, and 20, the readings greater than 14.2, and leaves `readings` alone.

Step 4, hasRepeatInARow (lines 39–51), checks consecutive pairs (4.10.3). The pair loop stops at `i < list.size() - 1`. The first pair, 12 and 15, differs; the second, 15 and 15, matches, so the method returns `true` at once without checking the rest. That early return is an "at least one" check (4.10.2) written without a flag.

```

Cleaned: [12, 15, 15, 20, 9]
Average: 14.2
Above average: [15, 15, 20]
Repeat in a row: true

```

Common mistakes

Several ArrayList bugs look alike in the code but behave very differently when the program runs. Use this table to go from the symptom back to the cause.

What you see	Likely cause	Fix
--------------	--------------	-----

Wrong answer, no error, after deleting	Forward loop added 1 to <code>i</code> after a <code>remove(i)</code> , skipping an element	Add 1 only when nothing was removed, or traverse backward
Program never finishes	Inserted at or before <code>i</code> without moving <code>i</code> past the new element	Add an extra <code>i++</code> after inserting
<code>IndexOutOfBoundsException</code> in a pair loop	<code>get(i + 1)</code> with <code>i < size()</code>	Use <code>i < size() - 1</code>
<code>IndexOutOfBoundsException</code> with two collections	Loop bounded by the longer collection	Bound by the shorter one
<code>ConcurrentModificationException</code>	Added to or removed from the list an enhanced for loop is traversing	Use an indexed loop, or add to a different list
Equal Integer values reported as different	<code>==</code> on two Integer objects compares references	Unbox into <code>int</code> variables first

Vocabulary

Term	Definition	Example
traversal	A loop that visits the elements of a list in order	<code>for (int i = 0; i < list.size(); i++)</code>
standard algorithm	A common, reusable pattern for a job such as a sum, a maximum, or a count	Track the largest value seen so far
unboxing	Java's automatic conversion of an Integer to an <code>int</code> (or a Double to a <code>double</code>)	<code>int temp = temps.get(i);</code>
precondition	Something that must be true before an algorithm runs	The list is not empty
consecutive elements	Elements next to each other, at indices <code>i</code> and <code>i + 1</code>	<code>get(i)</code> and <code>get(i + 1)</code>
insert	Add an element at a given index, moving later elements one place right	<code>tasks.add(i, "study");</code>
shift	Move every element one place, dropping one at an end and filling the other end	<code>remove(0)</code> then <code>add(8)</code>
rotate	Shift with wraparound, so the element that leaves one end returns at the other	<code>list.add(list.remove(0));</code>
reverse	Put the elements in the opposite order, by swapping pairs toward the middle	<code>[1, 2, 3]</code> becomes <code>[3, 2, 1]</code>
filtered list	A new list holding only the elements of another list that have a property	<code>early.add(fruit);</code>

Chapter summary

- Array algorithms carry over to `ArrayList` by using `size()`, `get(i)`, and `set(i, value)`. Start a minimum or maximum at an element of the list, and remember that an average needs a non-empty list.
- "At least one" flags start false, "all" flags start true, and counters start at 0. None of them is ever switched back inside the loop.
- Pair loops stop at `i < size() - 1`. Duplicate checks use nested loops with `j = i + 1`, and `Integer` elements are unboxed before comparing with `==`.
- Inserting at index `i` moves later elements right, so step past the new element. Deleting at index `i` moves later elements left, so add 1 only when nothing was removed, or traverse backward. Never change a list's size inside an enhanced for loop.
- `remove` and `add` shift and rotate a list, since `remove` returns the element it took out. A reverse swaps pairs with a temporary variable and stops at `size() / 2`.
- A filtered list is built by adding matching elements to a new list, leaving the original unchanged. Collections traversed side by side share one index, bounded so it is valid for every one of them.

Review questions

Question 15 Tracing

This program is meant to remove every negative change and then add up what is left, which should give `[5, 4]` and `Sum: 9`. Write the exact output, and explain where it goes wrong.

RemoveThenSum.java

```
1 import java.util.ArrayList;
2
3 public class RemoveThenSum
4 {
5     public static void main(String[] args)
6     {
7         ArrayList<Integer> changes = new ArrayList<Integer>();
8         changes.add(-2);
9         changes.add(-3);
10        changes.add(5);
11        changes.add(-1);
12        changes.add(4);
13
14        for (int i = 0; i < changes.size(); i++)
15        {
16            if (changes.get(i) < 0)
17            {
18                changes.remove(i);
19            }
20        }
21        int sum = 0;
22        for (int change : changes)
23        {
24            sum += change;
25        }
26        System.out.println(changes);
27        System.out.println("Sum: " + sum);
28    }
29 }
```

Question 16 Debugging

This version of `insertInOrder` correctly prints `[10, 25, 30, 40, 55]`, then crashes with an `IndexOutOfBoundsException` when inserting 70. Find the error and write the corrected line.

InsertNoBoundCheck.java

```
1 import java.util.ArrayList;
2
3 public class InsertNoBoundCheck
4 {
5     public static void insertInOrder(ArrayList<Integer> list, int value)
6     {
7         int index = 0;
8         while (list.get(index) < value)
9         {
10             index++;
11         }
12         list.add(index, value);
13     }
14
15     public static void main(String[] args)
16     {
17         ArrayList<Integer> times = new ArrayList<Integer>();
18         times.add(10);
19         times.add(25);
20         times.add(40);
21         times.add(55);
22
23         insertInOrder(times, 30);
24         System.out.println(times);
25         insertInOrder(times, 70);
26         System.out.println(times);
27     }
28 }
```

Question 17 Multiple choice

words is an `ArrayList<String>`, and dup starts as false. Which segment sets dup to true exactly when some word appears more than once anywhere in the list?

- A. Outer loop over i; inner loop for (int j = 0; j < words.size(); j++); set dup when words.get(i).equals(words.get(j))
- B. One loop for (int i = 0; i < words.size() - 1; i++); set dup when words.get(i).equals(words.get(i + 1))
- C. Outer loop over i; inner loop for (int j = i + 1; j < words.size(); j++); set dup when words.get(i).equals(words.get(j))
- D. Outer loop over i; inner loop for (int j = i + 1; j < words.size(); j++); set dup when words.get(i) == words.get(j)

Question 18 Tracing

What does this program print? Show the list after the rotation and after each swap.

RotateReverse.java

```
1 import java.util.ArrayList;
2
3 public class RotateReverse
4 {
5     public static void main(String[] args)
6     {
7         ArrayList<String> letters = new ArrayList<String>();
8         letters.add("a");
9         letters.add("b");
10        letters.add("c");
11        letters.add("d");
12
13        letters.add(letters.remove(0));
14        int last = letters.size() - 1;
15        for (int i = 0; i < letters.size() / 2; i++)
16        {
17            String temp = letters.get(i);
18            letters.set(i, letters.get(last - i));
19            letters.set(last - i, temp);
20        }
21        System.out.println(letters);
22    }
23 }
```

Question 19 Short answer

A classmate says, "When I delete while traversing, I always go backward, and I never skip anything. But going backward is just a trick; it is luck that it works." Explain why traversing backward is guaranteed not to skip elements, using what `remove(i)` does to the other elements.

Question 20 Tracing

What does this program print? Show difference and improved for each day, and explain how best is found.

WeekCompare.java

```
1 import java.util.ArrayList;
2
3 public class WeekCompare
4 {
5     public static void main(String[] args)
6     {
7         ArrayList<Integer> thisWeek = new ArrayList<Integer>();
8         thisWeek.add(5000);
9         thisWeek.add(6200);
10        thisWeek.add(4800);
11        thisWeek.add(7000);
12        ArrayList<Integer> lastWeek = new ArrayList<Integer>();
13        lastWeek.add(5200);
14        lastWeek.add(6000);
15        lastWeek.add(4800);
16        lastWeek.add(6500);
17
18        int improved = 0;
```

```

19     ArrayList<Integer> change = new ArrayList<Integer>();
20     for (int i = 0; i < thisWeek.size(); i++)
21     {
22         int difference = thisWeek.get(i) - lastWeek.get(i);
23         change.add(difference);
24         if (difference > 0)
25         {
26             improved++;
27         }
28     }
29     int best = change.get(0);
30     for (int value : change)
31     {
32         if (value > best)
33         {
34             best = value;
35         }
36     }
37     System.out.println(change);
38     System.out.println("Days improved: " + improved);
39     System.out.println("Best change: " + best);
40 }
41 }

```

Question 21 Multiple choice

The BeforeM program in 4.10.7 builds `early`, a filtered list of the fruits that come before "m", from the list `fruits`. After the loop, when is `early.size() == fruits.size()` true?

- A. Exactly when every fruit comes before "m"
- B. Exactly when at least one fruit comes before "m"
- C. Exactly when no fruit comes before "m"
- D. Exactly when `early` and `fruits` refer to the same list object

Reflection

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

- ☐ I can find the sum, average, minimum, or maximum of an ArrayList, and explain why a maximum starts at an element of the list. Reread: 4.10.1 Sum, Average, Minimum, and Maximum
- ☐ I can use a flag or a counter to decide whether any or all elements have a property, or how many do. Reread: 4.10.2 Any, All, and How Many
- ☐ I can write the bound for a consecutive-pair loop and check a list for duplicates with nested loops. Reread: 4.10.3 Consecutive Pairs and Duplicates
- ☐ I can insert elements during a traversal without looping forever. Reread: 4.10.4 Inserting While Traversing
- ☐ I can delete elements during a traversal without skipping any, in two different ways. Reread: 4.10.5 Deleting While Traversing
- ☐ I can shift, rotate, and reverse an ArrayList, and explain why a reverse stops halfway. Reread: 4.10.6 Shifting, Rotating, and Reversing

- ☐ I can build a filtered list from another list without changing the original. Reread: 4.10.7 Building a Filtered List
- ☐ I can traverse two collections side by side with one index and choose a safe loop bound. Reread: 4.10.8 Traversing Several Collections at Once

Check Your Reasoning

Answers and explanations for every question, numbered to match.

Question 1

banana

i	words.get(i)	Length	Longer than longest?	Longest after
				fig (3)
1	banana	6	6 > 3, yes	banana
2	kiwi	4	4 > 6, no	banana
3	cherry	6	6 > 6, no	banana
4	plum	4	4 > 6, no	banana

cherry has the same length as banana, and > is false for equal values, so the first longest word found is kept. With >=, the last word of the longest length would win instead.

Reread 4.10.1 *Sum, Average, Minimum, and Maximum* if your trace differed.

Question 2

B. A minimum must start at a value from the list. Starting at the first element, and then looping from index 1, compares every element exactly once.

- **A** is wrong if every value is positive: nothing is ever below 0, so `lowest` would stay 0, a value not in the list. Made-up starting values work only by luck.
- **C** is wrong because `size()` is the number of elements, not one of the readings.
- **D** is wrong because the loop starts at index 1, so index 0 would never be compared. It also crashes on a list with one element, since index 1 does not exist.

Reread 4.10.1 *Sum, Average, Minimum, and Maximum* if you chose differently.

Question 3

3 true

value	In 10 to 20?	inRange	over50
12	yes	1	false
55	no	1	true
20	yes	2	true
9	no	2	true
31	no	2	true
10	yes	3	true

Both ends of the range count because the condition uses >= and <=. Once `over50` becomes true at 55, nothing sets it back.

Reread 4.10.2 *Any, All, and How Many* if your trace differed.

Question 4

B. The flag starts `true` and switches to `false` when a name is shorter than 3 characters. It stays true only if no name is short, which means every name has 3 or more characters. This is the "all" pattern.

- **A** describes the opposite: `result` is **false** when a short name exists. The "at least one" pattern starts false.
- **C** is wrong because a `boolean` cannot hold a count; counting needs an `int` counter.
- **D** confuses the property being searched for with the property being confirmed. Short names make `result` false, not true.

Reread 4.10.2 *Any, All, and How Many* if you chose differently.

Question 5

The error is on line 15. The inner loop starts at `j = 0`, so on its very first pass `i` and `j` are both 0, and the code compares `codes.get(0)` with **itself**. 4 equals 4, so `duplicate` becomes true even though every value is different.

Corrected line:

```
1 for (int j = i + 1; j < codes.size(); j++)
```

Now each element is compared only with the elements after it, never with itself, and the program prints `Has duplicate: false`.

Reread 4.10.3 *Consecutive Pairs and Duplicates* if you missed the error.

Question 6

3

i	words.get(i - 1)	words.get(i)	Equal?	repeats
1	the	the	yes	1
2	the	cat	no	1
3	cat	sat	no	1
4	sat	sat	yes	2
5	sat	sat	yes	3

This loop starts at 1 and looks **back** at `i - 1`, so it may run all the way to `i < words.size()`. Three `sats` in a row make two pairs, which is why the count is 3, not 2.

Reread 4.10.3 *Consecutive Pairs and Duplicates* if your trace differed.

Question 7

[3, 4, 0, 7, 8, 0]

i at check	nums.get(i)	Action	List afterward
0	3	odd, none	[3, 4, 7, 8]
1	4	insert 0 at 2; i becomes 2	[3, 4, 0, 7, 8]

3	7	odd, none	(same)
4	8	insert 0 at 5; i becomes 5	[3, 4, 0, 7, 8, 0]

Then the loop's `i++` makes `i` 6, and `6 < 6` is false. The extra `i++` skips each inserted 0, which matters here because 0 is even.

Reread 4.10.4 *Inserting While Traversing* if your trace differed.

Question 8

C. At `i = 1`, a 0 goes in at index 2. The loop's `i++` moves to index 2, which holds that new 0. It is even, so another 0 goes in at index 3, then the loop visits it, and so on. The list keeps growing and the loop never catches up to the end.

- **A** is wrong because the extra `i++` is what skipped each inserted 0.
- **B** is wrong because the code inserts only after even values, and the new 0s keep triggering more insertions.
- **D** is wrong because `i + 1` is at most `size()` here, and inserting at `size()` is allowed.

Reread 4.10.4 *Inserting While Traversing* if you chose differently.

Question 9

[cat, eagle]

i	words.get(i)	Length below 3?	List afterward
4	eagle	no	[a, be, cat, do, eagle]
3	do	yes, remove	[a, be, cat, eagle]
2	cat	no	(same)
1	be	yes, remove	[a, cat, eagle]
0	a	yes, remove	[cat, eagle]

`a` and `be` are next to each other, the case that breaks a plain forward loop, but traversing backward still checks both.

Reread 4.10.5 *Deleting While Traversing* if your trace differed.

Question 10

C. Segment II traverses backward, so removals move only elements already checked. Segment III adds 1 to `k` only when nothing was removed. Both leave [5].

- **A** and **D** are wrong because segment I skips an element: removing 2 at index 0 slides 4 into index 0, and the loop moves on to index 1. Segment I leaves [4, 5].
- **B** is wrong because it leaves out segment III, which is also correct.

Reread 4.10.5 *Deleting While Traversing* if you chose differently.

Question 11

[4, 5, 1, 2, 3]

k	Removed from the end	List afterward
0	5	[5, 1, 2, 3, 4]

1	4	[4, 5, 1, 2, 3]
---	---	-----------------

Each pass is a rotate right: the last element is removed, and `add(0, ...)` puts it at the front. Two passes rotate right by 2.

Reread 4.10.6 *Shifting, Rotating, and Reversing* if your trace differed.

Question 12

```
[60, 50, 40, 40, 50, 60]
```

`last` is 5, and the loop runs for `i = 0, 1, 2`.

i	After set(i, get(last - i))	After set(last - i, get(i))
0	[60, 20, 30, 40, 50, 60]	(no change)
1	[60, 50, 30, 40, 50, 60]	(no change)
2	[60, 50, 40, 40, 50, 60]	(no change)

The first `set` overwrites the value at `i` before it is saved anywhere, so the second `set` copies the **new** value back. The front half is lost and the list ends up a mirror of its back half. Saving `get(i)` in `temp` first fixes it.

Reread 4.10.6 *Shifting, Rotating, and Reversing* if your trace differed.

Question 13

The program needs the full list later, so deleting from the original destroys data it still needs: the average would then be computed without the failing scores. Building a new list reads the original without changing it, so both lists are available.

A complete answer mentions:

- Filtering into a new list leaves the original unchanged.
- Deleting changes the original permanently, which here makes the later average wrong.
- Deleting during a traversal can skip elements (with a forward loop that always adds 1 to the index) or throw a `ConcurrentModificationException` (with an enhanced for loop). Building a new list avoids both.

Reread 4.10.7 *Building a Filtered List* and 4.10.5 *Deleting While Traversing* if your answer differed.

Question 14

A. One index walks both lists. At each index, `Math.max` receives the two `Integer` elements, which Java unboxes to `int` because `Math.max` expects `int` values. It returns the larger: `max(3, 5)` is 5, `max(8, 1)` is 8, and `max(2, 2)` is 2.

- **B** is the single largest value overall, a maximum algorithm, not a comparison at each position.
- **C** takes the **smaller** of each pair, which is `Math.min`.
- **D** adds both 2s, but the loop adds exactly one value per index, and there are 3 indices.

Reread 4.10.8 *Traversing Several Collections at Once* if you chose differently.

Question 15

```
[-3, 5, 4]
Sum: 6
```

i	changes.get(i)	Action	List afterward
0	-2	remove 0	[-3, 5, -1, 4]
1	5	none	(same)
2	-1	remove 2	[-3, 5, 4]

Then $3 < 3$ is false. Removing -2 slid -3 into index 0, and the loop moved on to index 1, so -3 was never checked. The sum loop then adds $-3 + 5 + 4 = 6$. The sum algorithm itself is correct; it just ran on a list that the deletion step got wrong. Traversing backward in the first loop would give [5, 4] and Sum: 9.

Reread 4.10.5 *Deleting While Traversing* if your trace differed.

Question 16

The error is on line 8. Every element is less than 70, so `index` climbs to 5, the size of the list, and the condition then calls `list.get(5)`. Index 5 does not exist, so Java throws:

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

(Wording can vary by Java version.) Inserting 30 worked only because 40 stopped the loop before it reached the end.

Corrected line:

```
1 while (index < list.size() && list.get(index) < value)
```

The range check must come **first**. When `index` reaches the size, `index < list.size()` is false, && short-circuits, and `get` is never called. Then `add(5, 70)` appends 70.

Reread 4.10.4 *Inserting While Traversing* if you missed the error.

Question 17

C. Each word is compared with every later word, never with itself, and `equals` compares the characters.

- **A** compares each word with itself when `j` equals `i`, so `dup` becomes true for any non-empty list. For [red, blue], which has no duplicates, it sets `dup` to true.
- **B** checks only neighbors. For [red, blue, red] the two reds are not next to each other, so `dup` stays false.
- **D** uses `==`, which compares references (Topic 2.6), not characters. Two equal strings can be different objects, so it is not a reliable test.

Reread 4.10.3 *Consecutive Pairs and Duplicates* if you chose differently.

Question 18

```
[a, d, c, b]
```

Line 13 rotates left: `remove(0)` takes out `a` and `add` puts it at the end, giving [b, c, d, a]. Then the reverse runs with `last = 3` for `i = 0` and 1:

i	Swaps indices	List afterward
---	---------------	----------------

0	0 and 3	[a, c, d, b]
1	1 and 2	[a, d, c, b]

Reversing the rotated list does not give back the original. Order matters when algorithms are combined.

Reread 4.10.6 *Shifting, Rotating, and Reversing* if your trace differed.

Question 19

`remove(i)` moves only the elements **after** index `i`, each one place left. The elements before `i` keep their indices. A backward traversal has already checked every element after `i`, and the ones it has not checked yet are all before `i`. So a removal never moves an element that still needs checking, and the next index, `i - 1`, holds exactly the element it held before.

A complete answer mentions:

- `remove(i)` shifts only the elements to the right of `i`.
- Going backward, those elements have already been checked.
- The unchecked elements, to the left, keep their indices, so none is skipped.

Reread 4.10.5 *Deleting While Traversing* if your explanation differed.

Question 20

```
[-200, 200, 0, 500]
Days improved: 2
Best change: 500
```

i	thisWeek.get(i)	lastWeek.get(i)	difference	improved
0	5000	5200	-200	0
1	6200	6000	200	1
2	4800	4800	0	1
3	7000	6500	500	2

Both lists are traversed with one index and both elements are unboxed for the subtraction. Each difference is added to a new list, and a counter tallies the positive differences. A difference of 0 is not an improvement, because the condition is `> 0`.

The second loop is a maximum over the new list. `best` starts at its first element, -200, so it works even if every difference is negative. It becomes 200, stays 200 for the 0, and becomes 500.

Reread 4.10.8 *Traversing Several Collections at Once* or 4.10.1 *Sum, Average, Minimum, and Maximum* if your trace differed.

Question 21

A. The filter adds one element to `early` for each fruit that passes the test, so `early.size()` counts the passing fruits. It equals `fruits.size()` only if every fruit passed. Comparing a filtered list's size with the original's is another way to answer an "all" question.

- **B** would be `early.size() > 0`.
- **C** would be `early.size() == 0`.
- **D** is wrong because `==` here compares two `int` sizes, not references; `early` is always a separate, new list.

Reread *4.10.7 Building a Filtered List* and *4.10.2 Any, All, and How Many* if you chose differently.