UNIT 4 · DATA COLLECTIONS

4.9 ArrayList Traversals

Self-Study Guide · About 50 minutes

By the end you will be able to

- Traverse an ArrayList with an indexed for loop or a while loop, forward, backward, or by steps.
- Explain why an index outside 0 through size() minus 1 throws an IndexOutOfBoundsException, and fix the loop that causes it.
- Traverse an ArrayList with an enhanced for loop, and predict when the loop can and cannot change the list.
- Explain why removing elements in a forward indexed loop skips elements, and use a traversal that does not skip.
- Explain why adding or removing elements inside an enhanced for loop can throw a ConcurrentModificationException.

What to remember from earlier

- An ArrayList can grow and shrink, and its indices run from 0 to size() minus 1. (Topic 4.8: ArrayList Methods, 4.8.3 Indexes and get)
- get(i) reads an element and set(i, value) replaces it. (Topic 4.8: ArrayList Methods)
- remove(i) takes out an element and slides later elements one place left. (Topic 4.8: ArrayList Methods, 4.8.6 Removing an Element)
- An ArrayList of Integer works with int values through boxing and unboxing. (Topic 4.7: Wrapper Classes)
- Assigning to an enhanced for loop variable never changes the array. (Topic 4.4: Array Traversals, 4.4.5 Assigning to the Loop Variable)

4.9.1 Visiting Every Element by Index

A music app keeps a playlist in an `ArrayList<String>`. To show the playlist on screen, the app has to look at every song, one after another. In Topic 4.8 you could reach one element at a time with `get`, but writing `songs.get(0)`, `songs.get(1)`, `songs.get(2)`, ... by hand only works if you know exactly how many songs there are. A playlist can have 4 songs today and 40 tomorrow.

You solved the same problem for arrays in Topic 4.4: let a loop variable count through the indices, and use it to reach each element in turn. The same plan works for an `ArrayList`. Only two details change:

- An array tells you its length with `.length` (no parentheses). An `ArrayList` tells you how many elements it holds with the method call `size()`.
- An array element is `scores[i]`. An `ArrayList` element is `songs.get(i)`.

Using a loop to visit all the elements of a list, or an ordered run of them (for example, every other one, or the last three), is called **traversing** the list. One such pass is a **traversal**.

KEY IDEA

To traverse an ArrayList by index, count `i` from `0` while `i < list.size()`, and reach each element with `list.get(i)`.

WORKED EXAMPLE **Printing a playlist with its positions**

Look at the loop header on line 13. It starts `i` at the first index and stops before `songs.size()`.

PlaylistIndexed.java

```
 1 import java.util.ArrayList;
 2
 3 public class PlaylistIndexed
 4 {
 5     public static void main(String[] args)
 6     {
 7         ArrayList<String> songs = new ArrayList<String>();
 8         songs.add("Sunrise");
 9         songs.add("Rain Song");
10         songs.add("Night Drive");
11         songs.add("Echoes");
12
13         for (int i = 0; i < songs.size(); i++)
14         {
15             System.out.println(i + ": " + songs.get(i));
16         }
17     }
18 }
```

The list holds 4 songs, so `songs.size()` returns `4` and the valid indices are 0, 1, 2, and 3.

i	i < songs.size()	songs.get(i)	Printed
0	`0 < 4` is `true`	`"Sunrise"`	`0: Sunrise`
1	`1 < 4` is `true`	`"Rain Song"`	`1: Rain Song`
2	`2 < 4` is `true`	`"Night Drive"`	`2: Night Drive`
3	`3 < 4` is `true`	`"Echoes"`	`3: Echoes`
4	`4 < 4` is `false`		loop ends

```
0: Sunrise
1: Rain Song
2: Night Drive
3: Echoes
```

Two things make this loop safe for a playlist of any length. First, the test calls `songs.size()` each time it runs, so it always uses the current number of songs. Second, `<` stops the loop at the last valid index, `size() - 1`, because the indices start at 0.

A traversal does not have to start at the front or visit every element. Because the loop header is an ordinary for loop from Topic 2.8, you can change the start, the test, and the update to walk through the list in a different order.

WORKED EXAMPLE **Backward, and every other song**

To play the list in reverse, start at the last index, `songs.size() - 1`, and count down while `i >= 0`:

PlaylistBackward.java

```java
import java.util.ArrayList;

public class PlaylistBackward
{
    public static void main(String[] args)
    {
        ArrayList<String> songs = new ArrayList<String>();
        songs.add("Sunrise");
        songs.add("Rain Song");
        songs.add("Night Drive");
        songs.add("Echoes");

        for (int i = songs.size() - 1; i >= 0; i--)
        {
            System.out.println(songs.get(i));
        }
    }
}
```

With 4 songs, `i` takes the values 3, 2, 1, 0, so this prints:

```
Echoes
Night Drive
Rain Song
Sunrise
```

The test is `i >= 0`, not `i > 0`, because index 0 holds a real song, `"Sunrise"`, which must not be left out.

A while loop can traverse a list too, as it could an array in Topic 4.4. This one has five songs and adds 2 to `i` each pass, so it visits only the songs at even indices:

PlaylistWhile.java

```java
import java.util.ArrayList;

public class PlaylistWhile
{
    public static void main(String[] args)
    {
        ArrayList<String> songs = new ArrayList<String>();
        songs.add("Sunrise");
        songs.add("Rain Song");
        songs.add("Night Drive");
        songs.add("Echoes");
        songs.add("Last Call");

        int i = 0;
        while (i < songs.size())
        {
            System.out.println(songs.get(i));
            i += 2;
        }
    }
}
```

i	i < songs.size()	Printed
0	`0 < 5` is `true`	`Sunrise`
2	`2 < 5` is `true`	`Night Drive`
4	`4 < 5` is `true`	`Last Call`
6	`6 < 5` is `false`	loop ends

The pieces of the for loop are all still there, just spread out: `int i = 0;` before the loop, the test as the while condition, and `i += 2;` as the last line of the body.

WATCH OUT

Writing array syntax on an ArrayList. Arrays and ArrayLists look alike, so students often reach for the array forms by habit:

ListLength.java

```
 1 import java.util.ArrayList;
 2
 3 public class ListLength
 4 {
 5     public static void main(String[] args)
 6     {
 7         ArrayList<String> songs = new ArrayList<String>();
 8         songs.add("Sunrise");
 9         songs.add("Echoes");
10
11         for (int i = 0; i < songs.length; i++)
12         {
13             System.out.println(songs[i]);
14         }
15     }
16 }
```

This does not compile. Java reports two errors:

```
ListLength.java:11: error: cannot find symbol
  symbol:   variable length
ListLength.java:13: error: array required, but ArrayList<String> found
```

(Exact wording can vary between Java versions.) An `ArrayList` is an object, not an array. It has no `length` variable, and square brackets only work on arrays. The fix is `songs.size()` on line 11 and `songs.get(i)` on line 13. A memory aid: arrays use `.length` and `[ ]`; ArrayLists use method calls with parentheses, `size()` and `get(i)`.

PAUSE AND TRY

Question 1 Multiple choice

`names` is an `ArrayList<String>` holding at least two names. Which loop header makes `names.get(i)` visit every name exactly once, from first to last?

A. `for (int i = 0; i < names.size(); i++)`
B. `for (int i = 0; i < names.length; i++)`
C. `for (int i = 0; i < names.size() - 1; i++)`
D. `for (int i = 1; i < names.size(); i++)`

Question 2 Tracing

What does this program print? Show the value of `i` for each pass.

PrizeTrace.java

```java
import java.util.ArrayList;

public class PrizeTrace
{
    public static void main(String[] args)
    {
        ArrayList<String> prizes = new ArrayList<String>();
        prizes.add("pen");
        prizes.add("mug");
        prizes.add("hat");
        prizes.add("bag");
        prizes.add("cap");

        for (int i = 1; i < prizes.size(); i += 2)
        {
            System.out.println(i + " " + prizes.get(i));
        }
    }
}
```

4.9.2 Staying Inside the Index Range

The loop in 4.9.1 stopped at exactly the right moment. What happens if a loop goes one step too far? With a `String` in Topic 1.15, asking for a character past the end crashed the program. With arrays in Topic 4.3, asking for a missing index crashed it too. An `ArrayList` follows the same rule, for the same reason: there is no element there to return, and Java refuses to make one up.

When a program calls `get`, `set`, or `remove` with an index less than 0 or greater than `size() - 1`, Java stops the program with an **IndexOutOfBoundsException**, a runtime error meaning "that index is outside the list." The program compiles, because the compiler cannot know how big the list will be when the line runs. The problem only appears when the line actually runs with a bad index.

KEY IDEA

The valid indices of an ArrayList are 0 through `size() - 1`. Any other index makes `get`, `set`, or `remove` throw an IndexOutOfBoundsException.

WORKED EXAMPLE **One pass too many**

This loop is meant to print three scores. Compare its test on line 12 with the one in 4.9.1.

ScoresTooFar.java

```
 1 import java.util.ArrayList;
 2
 3 public class ScoresTooFar
 4 {
 5     public static void main(String[] args)
 6     {
 7         ArrayList<Integer> scores = new ArrayList<Integer>();
 8         scores.add(88);
 9         scores.add(73);
10         scores.add(95);
11
12         for (int i = 0; i <= scores.size(); i++)
13         {
14             System.out.println(scores.get(i));
15         }
16         System.out.println("Done");
17     }
18 }
```

i	i <= scores.size()	What happens
0	`0 <= 3` is `true`	prints `88`
1	`1 <= 3` is `true`	prints `73`
2	`2 <= 3` is `true`	prints `95`
3	`3 <= 3` is `true`	`scores.get(3)`: there is no index 3. Crash

The output is the three scores, then an error message:

```
88
73
95
Exception in thread "main" java.lang.IndexOutOfBoundsException: Index 3 out of bounds
for length 3
```

(The exact message can vary between Java versions, and several more lines follow it that name the line where the crash happened.) Notice that `Done` never prints. When an exception stops a program, nothing after the crash runs.

The cause is `<=`. It lets `i` reach `size()`, which is one past the last index. The fix is the test from 4.9.1, `i < scores.size()`.

The same mistake can happen at the other end of a backward traversal. Change one thing from the backward loop in 4.9.1: start at `size()` instead of `size() - 1`.

WORKED EXAMPLE **Starting a backward loop one too high**

BackwardFromSize.java

```
1  import java.util.ArrayList;
2
3  public class BackwardFromSize
4  {
5      public static void main(String[] args)
6      {
7          ArrayList<Integer> scores = new ArrayList<Integer>();
8          scores.add(88);
9          scores.add(73);
10         scores.add(95);
11
12         for (int i = scores.size(); i >= 0; i--)
13         {
14             System.out.println(scores.get(i));
15         }
16     }
17 }
```

The list holds 3 scores, so the first value of `i` is `3`. The very first call, `scores.get(3)`, asks for an index that does not exist. Nothing prints at all before the program stops with `java.lang.IndexOutOfBoundsException: Index 3 out of bounds for length 3`. The fix is to start at the last real index: `int i = scores.size() - 1`.

WATCH OUT

Treating the size as an index. `size()` counts elements; it is not a position. A list of 3 elements has positions 0, 1, and 2, so `size()` is always one more than the last index. Students who write `<= list.size()`, or start a backward loop at `list.size()`, are using the count where they need a position. Two quick checks before you run a traversal: the first index used should be `0` (or `size() - 1` going backward), and the last one should be `size() - 1` (or `0` going backward).

PAUSE AND TRY

Question 3 Multiple choice

What happens when this program runs?

ShiftedStart.java

```java
import java.util.ArrayList;

public class ShiftedStart
{
    public static void main(String[] args)
    {
        ArrayList<String> colors = new ArrayList<String>();
        colors.add("red");
        colors.add("blue");
        colors.add("green");
        colors.add("gold");

        for (int i = 1; i <= colors.size(); i++)
        {
            System.out.println(colors.get(i));
        }
    }
}
```

A. It prints `red`, `blue`, `green`, and `gold`.
B. It prints `blue`, `green`, and `gold`, and then stops with an IndexOutOfBoundsException.
C. It prints `blue`, `green`, and `gold`, and then ends normally.
D. It does not compile, because `colors.size()` cannot be used in a loop test.

Question 4 Short answer

A student says, "`list.get(list.size())` gets the last element, because `size()` is how far the list goes." Explain what that call really does, and what the student should write instead.

4.9.3 Traversing with an Enhanced for Loop

Many traversals never use the index at all. Printing each song only needs the song, not its position. In Topic 4.4 you met a shorter loop for exactly this job on arrays, the **enhanced for loop**. It works on an `ArrayList` in the same way, and it removes the chance of an index mistake, because there is no index to get wrong.

Here is the form, read aloud as "for each `song` in `songs`":

```java
for (String song : songs)
```

- `String` is the type of each element. It must match the list's element type, the `E` in `ArrayList<E>`.
- `song` is the **enhanced for loop variable**. On each pass, Java assigns it the next element of the list, starting with the element at index 0.

- The colon separates the variable from the list being traversed.

WORKED EXAMPLE **The playlist again, with no index**

PlaylistEnhanced.java

```
import java.util.ArrayList;

public class PlaylistEnhanced
{
    public static void main(String[] args)
    {
        ArrayList<String> songs = new ArrayList<String>();
        songs.add("Sunrise");
        songs.add("Rain Song");
        songs.add("Night Drive");
        songs.add("Echoes");

        for (String song : songs)
        {
            System.out.println("Now playing: " + song);
        }
    }
}
```

Pass	`song` holds	Printed
1	`"Sunrise"`	`Now playing: Sunrise`
2	`"Rain Song"`	`Now playing: Rain Song`
3	`"Night Drive"`	`Now playing: Night Drive`
4	`"Echoes"`	`Now playing: Echoes`

The loop stops by itself after the last element. There is no `size()`, no `get`, and no `i`. The trade-off is that the loop always goes front to back, one element at a time; for backward or every-other traversals, use an indexed loop.

For a list of `Integer`, the loop variable can be an `int`. Each `Integer` element is unboxed into an `int` as it is assigned, which is the unboxing from Topic 4.7.

WORKED EXAMPLE **Unboxing in the loop header**

TempsEnhanced.java

```
1  import java.util.ArrayList;
2
3  public class TempsEnhanced
4  {
5      public static void main(String[] args)
6      {
7          ArrayList<Integer> temps = new ArrayList<Integer>();
8          temps.add(64);
9          temps.add(71);
10         temps.add(58);
11
12         for (int temp : temps)
13         {
14             int fahrenheitAboveFreezing = temp - 32;
15             System.out.println(temp + " is " + fahrenheitAboveFreezing + " above
   freezing");
16         }
17     }
18 }
```

On each pass, `temp` gets a plain `int` copy of the next element, so line 14 can do arithmetic with it directly:

```
64 is 32 above freezing
71 is 39 above freezing
58 is 26 above freezing
```

Writing `for (Integer temp : temps)` would also work; the arithmetic would unbox `temp` instead.

What the loop variable can and cannot change

The loop variable is a separate variable that is handed a copy of each element. That one fact explains two behaviors that look contradictory until you see it.

Assigning to the loop variable changes only the variable. In the next program, the goal is to make every word uppercase. (Printing an `ArrayList` with `println` shows its elements in square brackets, separated by commas.)

ShoutWrong.java

```
 1 import java.util.ArrayList;
 2
 3 public class ShoutWrong
 4 {
 5     public static void main(String[] args)
 6     {
 7         ArrayList<String> words = new ArrayList<String>();
 8         words.add("go");
 9         words.add("team");
10         words.add("win");
11
12         for (String word : words)
13         {
14             word = word.toUpperCase();
15         }
16         System.out.println(words);
17     }
18 }
```

```
[go, team, win]
```

Nothing changed. On each pass, line 14 builds a new uppercase `String` (Strings are immutable, Topic 1.15) and stores its reference in `word`. The list still refers to the original lowercase strings; `word` was only a copy of the reference, so pointing it at a new string leaves the list's slot alone. This matches the array rule from Topic 4.4.

To replace what is stored in the list, you need the slot's index so you can call `set`. That calls for an indexed loop:

ShoutFixed.java

```
 1 import java.util.ArrayList;
 2
 3 public class ShoutFixed
 4 {
 5     public static void main(String[] args)
 6     {
 7         ArrayList<String> words = new ArrayList<String>();
 8         words.add("go");
 9         words.add("team");
10         words.add("win");
11
12         for (int i = 0; i < words.size(); i++)
13         {
14             words.set(i, words.get(i).toUpperCase());
15         }
16         System.out.println(words);
17     }
18 }
```

```
[GO, TEAM, WIN]
```

Calling a method on the loop variable can change the object. When the list holds references to mutable objects, the loop variable refers to the very same object as the list does. They are aliases, as when a reference is passed to a method in Topic 3.6. Calling a mutator through the loop variable changes that shared object.

Java allows only one `public` class per file, so the helper class here leaves off `public`; in a real project each class gets its own file.

WORKED EXAMPLE **Giving every player a bonus**

`Player` has a private point total and a mutator, `addPoints`. The first loop in `main` calls the mutator on each player.

TeamBonus.java

```
import java.util.ArrayList;

class Player
{
    private String name;
    private int points;

    public Player(String playerName, int startPoints)
    {
        name = playerName;
        points = startPoints;
    }

    public String getName()
    {
        return name;
    }

    public int getPoints()
    {
        return points;
    }

    public void addPoints(int amount)
    {
        points = points + amount;
    }
}

public class TeamBonus
{
    public static void main(String[] args)
    {
        ArrayList<Player> team = new ArrayList<Player>();
        team.add(new Player("Ada", 10));
        team.add(new Player("Ben", 7));

        for (Player p : team)
        {
            p.addPoints(5);
        }
```

```
42
43          for (Player p : team)
44          {
45              System.out.println(p.getName() + " " + p.getPoints());
46          }
47      }
48  }
```

On the first pass of the loop on line 38, `p` refers to the same object as the list's element 0, Ada's `Player`. Line 40 adds 5 to that object's `points`. On the second pass, `p` refers to Ben's object. The list's references never change, but the objects they lead to do:

```
Ada 15
Ben 12
```

Inside `for (Player p : team)`	Changes the list?	Why
`p = new Player("Zed", 0);`	No	Only the copy in `p` points somewhere new
`p.addPoints(5);`	The object, yes	`p` and the list refer to the same object
`team.set(...)` to replace an element	Needs an index	Use an indexed loop; the enhanced loop has no `i`

KEY IDEA

An enhanced for loop gives its variable a copy of each element. Assigning to that variable never changes the list, but calling a mutator through it changes the object the list refers to.

WATCH OUT

Expecting the loop variable to write back. Students write `word = word.toUpperCase();` or `n = n * 2;` inside an enhanced for loop and expect the list to change, because the variable "is" the element. It is only a copy. The code compiles and runs, but the list is unchanged afterward. When you need to replace elements, use an indexed loop with `set(i, …)`.

PAUSE AND TRY

Question 5 Tracing

What does this program print?

DoubleLoopVar.java

```java
import java.util.ArrayList;

public class DoubleLoopVar
{
    public static void main(String[] args)
    {
        ArrayList<Integer> nums = new ArrayList<Integer>();
        nums.add(4);
        nums.add(3);
        nums.add(5);

        for (int n : nums)
        {
            n = n * 2;
            System.out.println(n);
        }
        System.out.println(nums);
    }
}
```

Question 6 Multiple choice

`team` is an `ArrayList<Player>`, using the `Player` class from 4.9.3. Which loop body, placed inside `for (Player p : team)`, changes what a later traversal of `team` will see?

A. `p = new Player(p.getName(), 100);`
B. `p.addPoints(100);`
C. `int total = p.getPoints() + 100;`
D. None of them; an enhanced for loop can never affect the list.

4.9.4 Removing Elements Without Skipping

A teacher keeps quiz scores in an `ArrayList<Integer>` and wants to drop every score below 70. In Topic 4.8 you removed one element at a known index with `remove(i)`. Here the program has to look at each score and decide, so `remove` goes inside a traversal. The obvious loop looks right, compiles, and runs without crashing, yet it sometimes gives the wrong answer.

The problem comes from what `remove` does to the list, which you saw in Topic 4.8: every element after the removed one slides one place to the left, and `size()` drops by 1. So the element that was

at index `i + 1` is now at index `i`. If the loop then does `i++`, it moves on to index `i + 1`, and the element that just slid into index `i` is never checked. It has been **skipped**.

Picture a line of people being checked by a guard who steps forward one position after each check. If the person in front of the guard leaves, everyone behind shuffles forward. The guard still steps forward one place, so the person who shuffled into the spot in front of the guard is never checked.

WORKED EXAMPLE **A score that slips past**

Two low scores, `62` and `58`, sit next to each other. Watch what happens to `58`.

RemoveSkip.java

```
 1 import java.util.ArrayList;
 2
 3 public class RemoveSkip
 4 {
 5     public static void main(String[] args)
 6     {
 7         ArrayList<Integer> scores = new ArrayList<Integer>();
 8         scores.add(85);
 9         scores.add(62);
10         scores.add(58);
11         scores.add(90);
12
13         for (int i = 0; i < scores.size(); i++)
14         {
15             if (scores.get(i) < 70)
16             {
17                 scores.remove(i);
18             }
19         }
20         System.out.println(scores);
21     }
22 }
```

`i`	List before this pass	`scores.get(i)`	Action
0	`[85, 62, 58, 90]`	85	keep
1	`[85, 62, 58, 90]`	62	remove index 1; `58` slides to index 1
2	`[85, 58, 90]`	90	keep (`58` at index 1 was never checked)
3	`[85, 58, 90]`		`3 < 3` is `false`; loop ends

```
[85, 58, 90]
```

`58` is below 70 but survived. After the removal on line 17, the loop's `i++` jumped straight from index 1 to index 2, past the score that had just moved into index 1.

There are two common ways to remove without skipping. Both make sure that every element is checked exactly once, even after a shift.

WORKED EXAMPLE **Fix 1: traverse backward**

RemoveBackward.java

```java
import java.util.ArrayList;

public class RemoveBackward
{
    public static void main(String[] args)
    {
        ArrayList<Integer> scores = new ArrayList<Integer>();
        scores.add(85);
        scores.add(62);
        scores.add(58);
        scores.add(90);

        for (int i = scores.size() - 1; i >= 0; i--)
        {
            if (scores.get(i) < 70)
            {
                scores.remove(i);
            }
        }
        System.out.println(scores);
    }
}
```

`i`	List before this pass	`scores.get(i)`	Action
3	`[85, 62, 58, 90]`	90	keep
2	`[85, 62, 58, 90]`	58	remove index 2
1	`[85, 62, 90]`	62	remove index 1
0	`[85, 90]`	85	keep

```
[85, 90]
```

When `remove(i)` shifts elements, only the elements **after** index `i` move, and going backward the loop has already checked all of them. The elements it has not checked yet, at smaller indices, stay exactly where they were. So nothing is skipped.

WORKED EXAMPLE **Fix 2: move forward only when nothing was removed**

RemoveStayPut.java

```java
import java.util.ArrayList;

public class RemoveStayPut
{
    public static void main(String[] args)
    {
        ArrayList<Integer> scores = new ArrayList<Integer>();
        scores.add(85);
        scores.add(62);
        scores.add(58);
        scores.add(90);

        int i = 0;
        while (i < scores.size())
        {
            if (scores.get(i) < 70)
            {
                scores.remove(i);
            }
            else
            {
                i++;
            }
        }
        System.out.println(scores);
    }
}
```

This while loop adds 1 to `i` only in the `else` branch, when the score was kept. When a score is removed, `i` stays put, so the next pass checks the element that just slid into index `i`.

`i`	List before this pass	`scores.get(i)`	Action
0	`[85, 62, 58, 90]`	85	keep; `i` becomes 1
1	`[85, 62, 58, 90]`	62	remove; `i` stays 1
1	`[85, 58, 90]`	58	remove; `i` stays 1
1	`[85, 90]`	90	keep; `i` becomes 2
2	`[85, 90]`		`2 < 2` is `false`; loop ends

```
[85, 90]
```

The test `i < scores.size()` calls `size()` every time, so it always uses the list's current, shrinking length.

KEY IDEA

Removing at index `i` slides the later elements left. A forward loop that still does `i++` after a removal skips the element that moved into index `i`. Traverse backward, or advance `i` only when nothing was removed.

WATCH OUT

Trusting one test run. The skipping bug only shows up when two elements to be removed sit next to each other. With the scores `[62, 85, 58, 90]`, the forward loop from the first example happens to produce the right answer, `[85, 90]`, because the element that slides into place after each removal is a score that should stay. A student who tests only that list will think the loop works. Always ask: "What if two elements in a row need to be removed?"

PAUSE AND TRY

Question 7 Tracing

This loop is supposed to remove every word shorter than 4 letters. What does it print? Show the list before each pass.

WordsSkip.java

```java
import java.util.ArrayList;

public class WordsSkip
{
    public static void main(String[] args)
    {
        ArrayList<String> words = new ArrayList<String>();
        words.add("the");
        words.add("an");
        words.add("apple");
        words.add("of");
        words.add("pie");

        for (int i = 0; i < words.size(); i++)
        {
            if (words.get(i).length() < 4)
            {
                words.remove(i);
            }
        }
        System.out.println(words);
    }
}
```

Question 8 Debugging

Correct the loop in Question 7 so that it removes every word shorter than 4 letters. Change only the loop header.

4.9.5 Changing the Size in an Enhanced for Loop

An enhanced for loop would seem like a tidy way to remove items: "for each task, do it, then take it off the list." But the enhanced for loop keeps track of where it is in the list behind the scenes, and it assumes the list keeps the same length for the whole traversal. If the loop body adds or removes an element, the loop's idea of "the next element" no longer matches the real list.

Rather than quietly skipping or repeating elements, Java checks for this. When the enhanced for loop goes to fetch the next element and finds that the list's size was changed during the traversal, it stops the program with a **ConcurrentModificationException**. "Concurrent modification" means the list was modified while it was being traversed.

WORKED EXAMPLE **Removing during an enhanced for loop**

The plan is to print each task and then remove it from the front of the list.

TaskListCrash.java

```
1  import java.util.ArrayList;
2
3  public class TaskListCrash
4  {
5      public static void main(String[] args)
6      {
7          ArrayList<String> tasks = new ArrayList<String>();
8          tasks.add("email");
9          tasks.add("laundry");
10         tasks.add("homework");
11
12         for (String task : tasks)
13         {
14             System.out.println("Done: " + task);
15             tasks.remove(0);
16         }
17         System.out.println("All tasks finished");
18     }
19 }
```

On the first pass, `task` is `"email"`, so line 14 prints `Done: email`. Line 15 removes element 0, so the list is now `[laundry, homework]`. The body ends, and the loop goes back to line 12 to fetch the next element. It sees that the list changed size and throws:

```
Done: email
Exception in thread "main" java.util.ConcurrentModificationException
```

(More lines follow, ending with one that names line 12, the loop header, since that is where the next element was being fetched. Wording can vary.) `All tasks finished` never prints.

Since the job is "keep handling the first task until none are left," a while loop that checks the size and uses indices does it correctly:

TaskListFixed.java

```
 1 import java.util.ArrayList;
 2
 3 public class TaskListFixed
 4 {
 5     public static void main(String[] args)
 6     {
 7         ArrayList<String> tasks = new ArrayList<String>();
 8         tasks.add("email");
 9         tasks.add("laundry");
10         tasks.add("homework");
11
12         while (tasks.size() > 0)
13         {
14             System.out.println("Done: " + tasks.get(0));
15             tasks.remove(0);
16         }
17         System.out.println("All tasks finished");
18     }
19 }
```

```
Done: email
Done: laundry
Done: homework
All tasks finished
```

Adding is just as much of a problem as removing. Change one thing: the loop body adds an element instead of removing one.

WORKED EXAMPLE **Adding during an enhanced for loop**

CartAddCrash.java

```
 1 import java.util.ArrayList;
 2
 3 public class CartAddCrash
 4 {
 5     public static void main(String[] args)
 6     {
 7         ArrayList<String> cart = new ArrayList<String>();
 8         cart.add("milk");
 9         cart.add("eggs");
10         cart.add("rice");
11
12         for (String item : cart)
13         {
14             System.out.println(item);
15             if (item.equals("milk"))
16             {
17                 cart.add("cereal");
18             }
19         }
20     }
21 }
```

The first pass prints `milk`, and because the item equals `"milk"`, line 17 appends `"cereal"`. When the loop fetches the next element, the list has grown, so Java throws a ConcurrentModificationException after printing only `milk`.

An indexed loop has no hidden position to fall out of step. Its test calls `cart.size()` on every pass, so it simply sees the longer list:

CartAddFixed.java

```
import java.util.ArrayList;

public class CartAddFixed
{
    public static void main(String[] args)
    {
        ArrayList<String> cart = new ArrayList<String>();
        cart.add("milk");
        cart.add("eggs");
        cart.add("rice");

        for (int i = 0; i < cart.size(); i++)
        {
            System.out.println(cart.get(i));
            if (cart.get(i).equals("milk"))
            {
                cart.add("cereal");
            }
        }
    }
}
```

```
milk
eggs
rice
cereal
```

The new item is added at the end, after the current index, so the loop reaches it on its last pass.

Java does not promise to catch every size change in an enhanced for loop; occasionally such a loop just ends early with no exception. You cannot count on either outcome, so the rule is simple.

KEY IDEA

Never add or remove elements while traversing an ArrayList with an enhanced for loop. Use an indexed for loop or a while loop, where you control the index, whenever the size will change.

Traversal	Can it change the size safely?	What to watch for
Indexed `for`, forward	Yes, with care	Removing skips the next element
Indexed `for`, backward	Yes	Start at `size() - 1`
`while`, advance on keep	Yes	Do not `i++` after a removal
Enhanced `for`	No	ConcurrentModificationException

WATCH OUT

Keeping your own counter in an enhanced for loop. Since the enhanced loop has no index, some students add one by hand and call `remove(index)`. That still changes the size inside an enhanced for loop, so it still causes the exception. (Question 14 shows this.) If you find yourself needing an index, that is the sign to switch to an indexed loop.

PAUSE AND TRY

Question 9 Multiple choice

What happens when this program runs?

GuestAdd.java

```java
import java.util.ArrayList;

public class GuestAdd
{
    public static void main(String[] args)
    {
        ArrayList<String> guests = new ArrayList<String>();
        guests.add("Ana");
        guests.add("Kim");
        guests.add("Lu");

        for (String guest : guests)
        {
            System.out.println(guest);
            if (guest.equals("Kim"))
            {
                guests.add("Kim's friend");
            }
        }
    }
}
```

A. It prints `Ana`, `Kim`, `Lu`, and `Kim's friend`.
B. It prints `Ana`, `Kim`, and `Lu`, and ends normally.
C. It prints `Ana` and `Kim`, and then stops with a ConcurrentModificationException.
D. It does not compile, because `add` cannot be called inside an enhanced for loop.

Question 10 Short answer

In 4.9.4, the forward indexed loop could be fixed by not moving `i` forward after a removal. Explain why that kind of fix is impossible in an enhanced for loop.

Putting it together

A weather station records one temperature reading per hour in an `ArrayList<Integer>`. When the sensor fails, it records `-999` instead of a real reading. This program prints the raw readings with their positions, removes every failed reading, and prints what is left. It uses three different traversals, each chosen for its job. Predict the output before reading the trace.

WORKED EXAMPLE **Cleaning up sensor readings**

SensorCleanup.java

```
 1 import java.util.ArrayList;
 2
 3 public class SensorCleanup
 4 {
 5     public static void main(String[] args)
 6     {
 7         ArrayList<Integer> readings = new ArrayList<Integer>();
 8         readings.add(72);
 9         readings.add(-999);
10         readings.add(-999);
11         readings.add(68);
12         readings.add(75);
13         readings.add(-999);
14
15         System.out.println("Raw readings:");
16         for (int i = 0; i < readings.size(); i++)
17         {
18             System.out.println(i + ": " + readings.get(i));
19         }
20
21         for (int i = readings.size() - 1; i >= 0; i--)
22         {
23             if (readings.get(i) == -999)
24             {
25                 readings.remove(i);
26             }
27         }
28
29         System.out.println("Clean readings:");
30         for (int reading : readings)
31         {
32             System.out.println(reading);
33         }
34         System.out.println("Readings kept: " + readings.size());
35     }
36 }
```

Lines 16–19: an indexed for loop, because the output needs each position as well as the value. It runs `i` from 0 to 5 and prints `0: 72` through `5: -999`.

Lines 21–27: a backward indexed loop, because it removes elements and must not skip any. Two failed readings sit side by side at indices 1 and 2.

`i`	List before this pass	`readings.get(i)`	Action
5	`[72, -999, -999, 68, 75, -999]`	-999	remove index 5
4	`[72, -999, -999, 68, 75]`	75	keep
3	`[72, -999, -999, 68, 75]`	68	keep
2	`[72, -999, -999, 68, 75]`	-999	remove index 2
1	`[72, -999, 68, 75]`	-999	remove index 1

0	`[72, 68, 75]`	72	keep

Lines 30–33: an enhanced for loop, because it only reads each value, in order, and does not change the size. Each `Integer` is unboxed into the `int` variable `reading`.

```
Raw readings:
0: 72
1: -999
2: -999
3: 68
4: 75
5: -999
Clean readings:
72
68
75
Readings kept: 3
```

If lines 21–27 had used a forward loop with `i++`, the second `-999` would have slid into index 1 right after the first was removed and been skipped, leaving `[72, -999, 68, 75]`. And if the removal had been done in an enhanced for loop, the program would have stopped with a ConcurrentModificationException on the first removal.

Vocabulary

Term	Definition	Example
traversing (a **traversal**)	Using a loop to visit all the elements of a list, or an ordered run of them	`for (int i = 0; i < list.size(); i++)`
IndexOutOfBoundsException	The runtime error thrown when `get`, `set`, or `remove` gets an index below 0 or above `size() - 1`	`list.get(list.size())`
enhanced for loop	A loop that hands its variable each element in order, without using indices	`for (String s : names)`
enhanced for loop variable	The variable in an enhanced for loop header; it holds a copy of the current element	`s` in `for (String s : names)`
skipped (element)	An element a forward loop never checks because a removal slid it into an index already passed	`58` in `[85, 62, 58, 90]`
ConcurrentModificationException	The runtime error an enhanced for loop can throw when the list's size changes during the traversal	`names.add("x")` inside `for (String s : names)`

Chapter summary

- Traversing an ArrayList by index works like traversing an array, with `size()` in place of `.length` and `get(i)` in place of `[i]`. Changing the start, test, and update lets a for or while loop go backward or visit only some elements.
- Valid indices run from 0 to `size() - 1`. Using `size()` itself, or any other index outside that range, throws an IndexOutOfBoundsException at run time.
- An enhanced for loop visits every element front to back with no index. Its variable holds a copy of each element: assigning to it never changes the list, but calling a mutator through it changes the shared object.
- Removing at index `i` slides later elements left, so a forward loop that always does `i++` skips elements. Traverse backward, or advance only when nothing was removed.
- Adding or removing elements inside an enhanced for loop can throw a ConcurrentModificationException. When the size will change, use an indexed for loop or a while loop.

Review questions

Question 11 Multiple choice

A bus app stores its stops in order in `stops`, an `ArrayList<String>` holding `Main`, `Oak`, and `Pine`. Which loop prints all three stops in reverse order, `Pine`, `Oak`, `Main`, and then ends normally? Each loop has the body `System.out.println(stops.get(i));`.

A. `for (int i = stops.size() - 1; i >= 0; i--)`
B. `for (int i = stops.size(); i >= 0; i--)`
C. `for (int i = stops.size() - 1; i > 0; i--)`
D. `for (int i = stops.size() - 1; i >= 0; i++)`

Question 12 Tracing

A seating chart marks empty seats with `"x"`. This loop is meant to remove every `"x"`. What does it print, and why is one `"x"` left?

SeatsSkip.java

```java
import java.util.ArrayList;

public class SeatsSkip
{
    public static void main(String[] args)
    {
        ArrayList<String> seats = new ArrayList<String>();
        seats.add("x");
        seats.add("x");
        seats.add("Ana");
        seats.add("x");
        seats.add("Bo");

        for (int i = 0; i < seats.size(); i++)
        {
            if (seats.get(i).equals("x"))
            {
                seats.remove(i);
            }
        }
        System.out.println(seats);
    }
}
```

Question 13 Multiple choice

This program uses the `Player` class from 4.9.3. What does it print?

TeamReset.java

```java
import java.util.ArrayList;

class Player
{
    private String name;
    private int points;

    public Player(String playerName, int startPoints)
    {
        name = playerName;
        points = startPoints;
    }

    public String getName()
    {
```

```
16          return name;
17      }
18
19      public int getPoints()
20      {
21          return points;
22      }
23
24      public void addPoints(int amount)
25      {
26          points = points + amount;
27      }
28  }
29
30  public class TeamReset
31  {
32      public static void main(String[] args)
33      {
34          ArrayList<Player> team = new ArrayList<Player>();
35          team.add(new Player("Ada", 10));
36          team.add(new Player("Ben", 7));
37
38          for (Player p : team)
39          {
40              p = new Player(p.getName(), 0);
41          }
42
43          for (Player p : team)
44          {
45              p.addPoints(2);
46          }
47
48          for (Player p : team)
49          {
50              System.out.println(p.getName() + " " + p.getPoints());
51          }
52      }
53  }
```

A. `Ada 2` and `Ben 2`
B. `Ada 12` and `Ben 9`
C. `Ada 10` and `Ben 7`
D. Nothing; assigning to `p` inside an enhanced for loop throws a ConcurrentModificationException.

Question 14 Debugging

A student wants to remove every score below 70. Since an enhanced for loop has no index, they keep their own counter. The program stops with a ConcurrentModificationException and prints nothing. Explain the error and correct the code.

ScoresEnhancedRemove.java

```java
import java.util.ArrayList;

public class ScoresEnhancedRemove
{
    public static void main(String[] args)
    {
        ArrayList<Integer> scores = new ArrayList<Integer>();
        scores.add(85);
        scores.add(62);
        scores.add(90);
        scores.add(77);

        int index = 0;
        for (int score : scores)
        {
            if (score < 70)
            {
                scores.remove(index);
            }
            index++;
        }
        System.out.println(scores);
    }
}
```

Question 15 Short answer

A student tests the forward removal loop from 4.9.4 (`for (int i = 0; i < scores.size(); i++)` with `scores.remove(i)` when a score is below 70) on the list `[62, 85, 58, 90]`. It prints `[85, 90]`, so the student says the loop is correct. Explain why this test did not reveal the bug, and describe a list that would.

Reflection

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

- [ ] I can traverse an ArrayList with an indexed for loop or a while loop, forward, backward, or by steps. Reread: 4.9.1 Visiting Every Element by Index

- [ ] I can explain why an index of size() or more causes an IndexOutOfBoundsException, and fix the loop. Reread: 4.9.2 Staying Inside the Index Range
- [ ] I can traverse an ArrayList with an enhanced for loop and say when it can and cannot change the list. Reread: 4.9.3 Traversing with an Enhanced for Loop
- [ ] I can explain why a forward loop skips elements when removing, and remove without skipping. Reread: 4.9.4 Removing Elements Without Skipping
- [ ] I can explain why adding or removing in an enhanced for loop can cause a ConcurrentModificationException. Reread: 4.9.5 Changing the Size in an Enhanced for Loop