

4.8 ArrayList Methods

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

By the end you will be able to

- Import the ArrayList class and create an empty ArrayList that holds a chosen type of element.
- Use size, add, get, set, and remove to build and change a list, and predict the list after each call.
- Explain how inserting and removing shift the indexes of later elements, and why an index outside the valid range crashes the program.
- Store numbers in an ArrayList through the Integer and Double wrapper classes, and explain how the element type lets the compiler catch mistakes.
- Explain that an ArrayList holds references to objects, and trace code that changes an object through the list or through another variable.

What to remember from earlier

- An array's length is fixed when it is created. ([Topic 4.3: Array Creation and Access, 4.3.5 Array Length Is Fixed](#))
- An index outside the valid range throws an `ArrayIndexOutOfBoundsException`. ([Topic 4.3: Array Creation and Access, 4.3.6 Valid Indexes and Out-of-Bounds Errors](#))
- Java converts between `int` and `Integer`, and `double` and `Double`, automatically. ([Topic 4.7: Wrapper Classes](#))
- A variable of a class type holds a reference, so two variables can refer to the same object. ([Topic 3.6: Methods: Passing and Returning References of an Object, 3.6.1 Passing an Object Reference](#))

4.8.1 Creating an Empty ArrayList

Picture an app that keeps a grocery list. When you open it, the list is empty. During the week you add items, cross some off, and squeeze a forgotten item into the middle. Nobody knows ahead of time how many items there will be.

An array from Topic 4.3 is a poor fit for this job. Its length is set when it is created and can never change. If you make an array of 10 items and need an 11th, you are stuck: you would have to build a bigger array and copy everything over. And removing an item from the middle of an array leaves a gap that you would have to close by hand.

Java has a class built for lists that change: `ArrayList`. An `ArrayList` object is a list of elements, kept in order, that can **grow and shrink** while the program runs. You can add an element, and the list gets one longer. You can remove an element, and the list gets one shorter, with no gap left behind. Because its size can change, we say an `ArrayList` is **mutable in size**. (An array's elements can change, but its length cannot; an `ArrayList` can change both.)

Importing the class

`String`, `Math`, `Integer`, and `Double` live in a package named `java.lang`, which every Java program can use automatically (Topic 1.15). `ArrayList` lives in a different package, `java.util` (short for "utilities"), the same package as `Scanner`. To use a class from any package other than `java.lang`, you must tell Java where to find it with an **import statement** at the very top of the file, before the class. You wrote one in Topic 1.4, `import java.util.Scanner;`, and `ArrayList` needs the same kind of line:

```
1 import java.util.ArrayList;
```

Read it as "bring in the class `ArrayList` from the package `java.util`." Without this line, Java has never heard of `ArrayList`.

Declaring and constructing the list

When you create an `ArrayList`, you also say what type of element it will hold, written in angle brackets after the class name. Here is a list that holds `Strings`:

```
1 ArrayList<String> groceries = new ArrayList<String>();
```

Piece	What it means
<code>ArrayList<String></code>	The variable's type: "an <code>ArrayList</code> whose elements are <code>Strings</code> "
<code>groceries</code>	The variable name; like any object variable (Topic 1.13), it holds a reference to the list
<code>new ArrayList<String>()</code>	Calls the <code>ArrayList</code> constructor , which takes no arguments and builds an empty list

The type in the angle brackets is called the **type parameter**. Java's documentation writes the class as `ArrayList<E>`, where `E` stands for "the type of the elements" and gets replaced by a real type, such as `String`, when you write your code. A class that takes a type parameter like this is called a **generic** class. You will see in 4.8.7 why stating the type matters so much.

Notice what the constructor does **not** take: a size. With an array you wrote `new String[10]` and got 10 slots. An `ArrayList` starts with no elements at all and grows only as you add them. (Java also lets you shorten the right side to `new ArrayList<>()`, leaving the type out of the second pair of brackets because it is already on the left. Both forms work; this guide writes the type both times.)

KEY IDEA

Write `import java.util.ArrayList;` at the top of the file, then `ArrayList<String> groceries = new ArrayList<String>();` to create an empty list of `Strings`. The constructor `ArrayList()` always makes a list with no elements.

WORKED EXAMPLE A brand-new list is empty

This program creates a list and asks two questions about it right away. The method `size()` returns the number of elements in the list as an `int`; you will use it constantly in the next subsection.

EmptyList.java

```
1 import java.util.ArrayList;
2
3 public class EmptyList
4 {
5     public static void main(String[] args)
6     {
7         ArrayList<String> groceries = new ArrayList<String>();
8         System.out.println(groceries.size());
9         System.out.println(groceries);
10    }
11 }
```

```
0
[]
```

Line 1 imports the class. Line 7 creates the list; there are no elements yet, so line 8 prints 0. Line 9 prints the list itself. As you learned in Topic 1.15, printing an object makes Java call that object's `toString` method. `ArrayList`'s version of `toString` lists the elements inside square brackets, separated by commas, so an empty list prints as `[]`. That bracket format makes it easy to see what a list holds, and this guide uses it in almost every example.

Now vary the element type. The type parameter can be any class: `String`, `Double`, `Integer`, or a class you wrote yourself in Unit 3.

WORKED EXAMPLE Two empty lists of different types

EmptyTwoLists.java

```
1 import java.util.ArrayList;
2
3 public class EmptyTwoLists
4 {
5     public static void main(String[] args)
6     {
7         ArrayList<String> names = new ArrayList<String>();
8         ArrayList<Double> temperatures = new ArrayList<Double>();
9         System.out.println(names.size());
10        System.out.println(temperatures.size());
11        System.out.println(temperatures);
12    }
13 }
```

```
0
0
[]
```

Lines 7 and 8 build two separate lists. `names` will hold `Strings` and `temperatures` will hold `Double`s, but right now both are empty, so both sizes are 0. One import line on line 1 covers every `ArrayList` in the file, whatever its element type.

WATCH OUT

Forgetting the import. This file is the same as `EmptyList.java` without line 1:

`MissingImport.java`

```
1 public class MissingImport
2 {
3     public static void main(String[] args)
4     {
5         ArrayList<String> groceries = new ArrayList<String>();
6         System.out.println(groceries.size());
7     }
8 }
```

```
MissingImport.java:5: error: cannot find symbol
    ArrayList<String> groceries = new ArrayList<String>();
    ^
symbol:   class ArrayList
location: class MissingImport
```

Java reports the same error a second time for the `ArrayList` after `new`. The message says Java cannot find a **class** named `ArrayList`: since it is not in `java.lang`, Java does not look for it unless you import it. Students forget the line because most programs since Unit 1 have used only `java.lang` classes, which need no import. The fix is to add `import java.util.ArrayList;` as the first line. (Exact wording can vary between Java versions; look for "cannot find symbol" and "class ArrayList".)

PAUSE AND TRY**Question 1** Multiple choice

A student runs `ArrayList<Double> prices = new ArrayList<Double>();`. Which statement about `prices` right after this line is true?

- A. It holds 10 elements, each equal to `0.0`.
- B. It holds no elements, so `prices.size()` returns `0`.
- C. It holds one element, `null`, as a placeholder.
- D. It cannot be used until a length is written inside the parentheses.

Question 2 Short answer

A program uses `String`, `Math`, and `ArrayList`. Only `ArrayList` needs an import statement. Explain why.

4.8.2 Adding to the End and Counting

An empty list is a start, but a grocery list needs items. The simplest way to put an element in a list is to tack it onto the end, the way you write a new item at the bottom of a paper list.

The method `add(E obj)` does exactly that: it **appends** `obj` to the end of the list, and the list's size goes up by 1. The `E` in the header is the type parameter from 4.8.1, so for an `ArrayList<String>`, `add` takes a `String`. Its full header is `boolean add(E obj)`: it returns `true` every time. The return value exists for reasons outside this course, so you will almost always call `add` as a statement on its own and ignore what it returns.

The method `int size()` returns how many elements the list holds right now. Because elements come and go, `size()` can give a different answer each time you call it.

KEY IDEA

`list.add(obj)` puts `obj` at the end of the list and makes the list one longer. `list.size()` returns the current number of elements.

WORKED EXAMPLE Building a grocery list

AddToEnd.java

```
1 import java.util.ArrayList;
2
3 public class AddToEnd
4 {
5     public static void main(String[] args)
6     {
7         ArrayList<String> groceries = new ArrayList<String>();
8         groceries.add("milk");
9         System.out.println(groceries.size());
10        groceries.add("eggs");
11        groceries.add("bread");
12        System.out.println(groceries.size());
13        System.out.println(groceries);
14    }
15 }
```

Line	Call	groceries afterward	size()
7	constructor	<code>[]</code>	0
8	<code>groceries.add("milk")</code>	<code>[milk]</code>	1
10	<code>groceries.add("eggs")</code>	<code>[milk, eggs]</code>	2
11	<code>groceries.add("bread")</code>	<code>[milk, eggs, bread]</code>	3

```
1
3
[milk, eggs, bread]
```

Line 9 asks for the size after one `add`, so it prints 1. Lines 10 and 11 each add one more, so line 12 prints 3. Line 13 shows the order: elements stay in the order they were added, with the newest at the end.

The next example changes two things: it adds a value that is already in the list, and it prints what `add` returns.

WORKED EXAMPLE Duplicates are allowed, and add returns true

AddDuplicates.java

```

1  import java.util.ArrayList;
2
3  public class AddDuplicates
4  {
5      public static void main(String[] args)
6      {
7          ArrayList<String> orders = new ArrayList<String>();
8          orders.add("taco");
9          orders.add("salad");
10         System.out.println(orders.add("taco"));
11         System.out.println(orders);
12         System.out.println(orders.size());
13     }
14 }

```

```

true
[taco, salad, taco]
3

```

Line 10 calls `add` inside `println`, so the value `add` returns gets printed: `true`. The call still does its job, so "taco" is appended a second time. An `ArrayList` does not check for repeats; "taco" now appears at the front and at the end, and the size counts both, giving 3.

You now know three different ways Java asks "how many?", and they are easy to mix up:

Kind of object	How to ask its size	Example
Array	<code>length</code> (no parentheses; it is not a method)	<code>scores.length</code>
String	<code>length()</code> method	<code>word.length()</code>
ArrayList	<code>size()</code> method	<code>groceries.size()</code>

WATCH OUT

Writing `length` for a list. A student used to arrays and strings writes both of the wrong forms:

SizeNotLength.java

```

1  import java.util.ArrayList;
2
3  public class SizeNotLength
4  {
5      public static void main(String[] args)
6      {
7          ArrayList<String> groceries = new ArrayList<String>();
8          groceries.add("milk");
9          System.out.println(groceries.length);
10         System.out.println(groceries.length());
11     }
12 }

```

```
SizeNotLength.java:9: error: cannot find symbol
    System.out.println(groceries.length);
                           ^
SizeNotLength.java:10: error: cannot find symbol
    System.out.println(groceries.length());
                           ^
```

An ArrayList has no `length` variable and no `length()` method, so Java cannot find either one. The only way to ask a list for its number of elements is `groceries.size()`. (Error wording can vary between Java versions; the key words are "cannot find symbol".)

PAUSE AND TRY

Question 3 Tracing

Write the exact output of this program.

AddTrace.java

```
1 import java.util.ArrayList;
2
3 public class AddTrace
4 {
5     public static void main(String[] args)
6     {
7         ArrayList<String> pets = new ArrayList<String>();
8         System.out.println(pets.size());
9         pets.add("cat");
10        pets.add("dog");
11        System.out.println(pets.size());
12        pets.add("cat");
13        pets.add("fish");
14        System.out.println(pets.size());
15        System.out.println(pets);
16    }
17 }
```

4.8.3 Indexes and get

A list is only useful if you can look at what is in it. Just like an array (Topic 4.3), each element of an ArrayList has a position number called its **index**. The first element is at index 0, the next at index 1, and so on. The last element is at index `size() - 1`, one less than the number of elements, because counting started at 0 instead of 1.

Here is the grocery list from 4.8.2 with its indexes:

index:	0	1	2
value:	"milk"	"eggs"	"bread"

There are 3 elements, so the valid indexes are 0, 1, and 2. There is no index 3.

With an array you would write `groceries[1]`, but square brackets do not work on an `ArrayList`, because it is an object, not an array. Instead you call the method `E get(int index)`, which **returns** the element at that index. The return type is `E`, the list's element type, so on an `ArrayList<String>`, `get` hands back a `String`. That means you can store the result in a `String` variable, or call a `String` method on it right away.

`get` only looks. It does not remove or change anything, and the size stays the same.

KEY IDEA

The indexes of an `ArrayList` run from 0 to `size() - 1`. `list.get(i)` returns the element at index `i`, with the list's element type, and leaves the list unchanged.

WORKED EXAMPLE Reading elements by index

GetByIndex.java

```
1 import java.util.ArrayList;
2
3 public class GetByIndex
4 {
5     public static void main(String[] args)
6     {
7         ArrayList<String> groceries = new ArrayList<String>();
8         groceries.add("milk");
9         groceries.add("eggs");
10        groceries.add("bread");
11        System.out.println(groceries.get(0));
12        System.out.println(groceries.get(2));
13        String middle = groceries.get(1);
14        System.out.println(middle.toUpperCase());
15        System.out.println(groceries.size());
16    }
17 }
```

Line	Expression	Value	Printed
11	<code>groceries.get(0)</code>	"milk"	milk
12	<code>groceries.get(2)</code>	"bread"	bread
13	<code>middle = groceries.get(1)</code>	"eggs"	
14	<code>middle.toUpperCase()</code>	"EGGS"	EGGS
15	<code>groceries.size()</code>	3	3

```
milk
bread
EGGS
3
```

Line 13 stores the returned `String` in a `String` variable; this compiles because the list's element type is `String`. Line 15 shows that the three `get` calls did not use up or remove anything: the list still has 3 elements.

Because the last index is always `size() - 1`, you can reach the last element without knowing how long the list is. The next example uses that idea while the list grows.

WORKED EXAMPLE Always reaching the last element

GetLast.java

```
1 import java.util.ArrayList;
2
3 public class GetLast
4 {
5     public static void main(String[] args)
6     {
7         ArrayList<String> songs = new ArrayList<String>();
8         songs.add("Rain");
9         songs.add("Glow");
10        System.out.println(songs.get(songs.size() - 1));
11        songs.add("Tides");
12        songs.add("Ember");
13        System.out.println(songs.get(songs.size() - 1));
14        System.out.println(songs.get(1).length());
15    }
16 }
```

Line	songs	size() - 1	get(size() - 1)
10	[Rain, Glow]	1	"Glow"
13	[Rain, Glow, Tides, Ember]	3	"Ember"

```
Glow
Ember
4
```

The same expression on lines 10 and 13 finds a different element, because `size()` is recalculated at each call. Line 14 chains two calls: `songs.get(1)` returns `"Glow"`, and then `.length()` is called on that `String`, giving 4.

WATCH OUT

Using `size()` as an index. Since the list has 3 elements, it is tempting to think index 3 is the last one:

GetPastEnd.java

```
1 import java.util.ArrayList;
2
3 public class GetPastEnd
4 {
5     public static void main(String[] args)
6     {
7         ArrayList<String> groceries = new ArrayList<String>();
8         groceries.add("milk");
9         groceries.add("eggs");
10        groceries.add("bread");
11        System.out.println(groceries.get(2));
12        System.out.println(groceries.get(3));
13    }
14 }
```

```
bread
Exception in thread "main" java.lang.IndexOutOfBoundsException: Index 3 out of
bounds for length 3
    at GetPastEnd.main(GetPastEnd.java:12)
```

The program compiles, because Java cannot know the list's size until the program runs. Line 11 works and prints `bread`. On line 12 the list checks the index, finds that 3 is not between 0 and 2, and throws an `IndexOutOfBoundsException`, which stops the program. (The real message also lists a few lines from inside the `ArrayList` class, and the exact wording can vary between Java versions.) This is the list version of the `ArrayIndexOutOfBoundsException` you met with arrays. The fix is to use `groceries.get(2)`, or `groceries.get(groceries.size() - 1)` for "the last element."

PAUSE AND TRY

Question 4 Tracing

Write the exact output of this program.

GetTrace.java

```
1 import java.util.ArrayList;
2
3 public class GetTrace
4 {
5     public static void main(String[] args)
6     {
7         ArrayList<String> cities = new ArrayList<String>();
8         cities.add("Lima");
9         cities.add("Oslo");
10        cities.add("Cairo");
11        cities.add("Pune");
12        String pick = cities.get(cities.size() - 2);
13        System.out.println(pick);
14        System.out.println(cities.get(1).substring(0, 2));
15        System.out.println(cities.get(0).length() + cities.size());
16    }
17 }
```

Question 5 Multiple choice

days is an ArrayList<String> holding [Mon, Tue, Wed, Thu]. Which call throws an IndexOutOfBoundsException?

- A. days.get(0)
- B. days.get(3)
- C. days.get(days.size() - 1)
- D. days.get(4)

4.8.4 Replacing an Element with set

Suppose your chore list says wash, dry, fold, and you decide to iron instead of dry. You do not want a fourth chore; you want to cross out dry and write iron in its place. With an array, you would write chores[1] = "iron";. An ArrayList again uses a method instead of square brackets.

The method **E set(int index, E obj)** **replaces** the element at index with obj. The list keeps the same size, and every other element stays where it was. As a bonus, set **returns the element that used to be at that index**, so you can keep the old value if you need it. (If you do not need it, call set as a statement and ignore the result.)

Because set replaces an element that already exists, the index must be a valid index for get: from 0 to size() - 1.

KEY IDEA

list.set(i, obj) puts obj at index i in place of the element that was there, and returns that old element. The size does not change.

WORKED EXAMPLE Swapping one chore for another

SetReplace.java

```
1 import java.util.ArrayList;
2
3 public class SetReplace
4 {
5     public static void main(String[] args)
6     {
7         ArrayList<String> chores = new ArrayList<String>();
8         chores.add("wash");
9         chores.add("dry");
10        chores.add("fold");
11        String old = chores.set(1, "iron");
12        System.out.println(old);
13        System.out.println(chores);
14        System.out.println(chores.size());
15    }
16 }
```

Step	Line	What happens	chores
1	8–10	three appends	[wash, dry, fold]
2	11	set(1, "iron") puts "iron" at index 1	[wash, iron, fold]
3	11	set returns the old element, "dry", into old	[wash, iron, fold]

```
dry
[wash, iron, fold]
3
```

Line 12 prints `dry`, not `iron`: `set` returns what was **replaced**, not what was put in. Line 14 confirms the size is still 3.

Now use that return value for something. Swapping two elements is a classic job: with only `set` and `get`, you might worry that the first `set` destroys a value before you can move it. The returned old element solves that.

WORKED EXAMPLE Swapping the first and last runners

SetSwap.java

```
1 import java.util.ArrayList;
2
3 public class SetSwap
4 {
5     public static void main(String[] args)
6     {
7         ArrayList<String> lineup = new ArrayList<String>();
8         lineup.add("Ava");
9         lineup.add("Ben");
10        lineup.add("Cal");
11        String moved = lineup.set(0, lineup.get(2));
12        lineup.set(2, moved);
13        System.out.println(lineup);
14    }
15 }
```

Line 11 does two things. First Java evaluates the argument `lineup.get(2)`, which is "Cal". Then `set(0, "Cal")` puts "Cal" at index 0 and returns the element it replaced, "Ava", which is stored in `moved`.

Line	lineup	moved
8–10	[Ava, Ben, Cal]	
11	[Cal, Ben, Cal]	"Ava"
12	[Cal, Ben, Ava]	"Ava"

```
[Cal, Ben, Ava]
```

After line 11, "Cal" appears twice for a moment, and "Ava" is no longer in the list at all. It survives only because `set` returned it. Line 12 puts it back at index 2.

WATCH OUT

Using `set` to add a new element. A student wants to put "iron" after `fold` and writes `set` with the next index:

SetPastEnd.java

```
1 import java.util.ArrayList;
2
3 public class SetPastEnd
4 {
5     public static void main(String[] args)
6     {
7         ArrayList<String> chores = new ArrayList<String>();
8         chores.add("wash");
9         chores.add("dry");
10        chores.add("fold");
11        System.out.println(chores);
12        chores.set(3, "iron");
13        System.out.println(chores);
14    }
15 }
```

[wash, dry, fold]

Exception in thread "main" java.lang.IndexOutOfBoundsException: Index 3 out of bounds for length 3
at SetPastEnd.main(SetPastEnd.java:12)

set can only replace an element that exists. There is nothing at index 3 to replace, so line 12 throws an `IndexOutOfBoundsException` (wording can vary between Java versions). To put a new element at the end, use `chores.add("iron");`.

PAUSE AND TRY

Question 6 Tracing

Write the exact output of this program.

SetTrace.java

```
1 import java.util.ArrayList;
2
3 public class SetTrace
4 {
5     public static void main(String[] args)
6     {
7         ArrayList<String> seats = new ArrayList<String>();
8         seats.add("Ira");
9         seats.add("Jo");
10        seats.add("Kim");
11        String first = seats.set(2, "Lee");
12        String second = seats.set(0, first);
13        System.out.println(first);
14        System.out.println(second);
15        System.out.println(seats);
16    }
17 }
```

4.8.5 Inserting in the Middle

Back to the grocery app: you forgot an item and want it near the top, not at the bottom. Or a friend joins a line of people, and you let them in at second place. `add(obj)` always goes to the end, and `set` overwrites someone. You need a way to **squeeze** an element in.

There is a second `add` method with two parameters: `void add(int index, E obj)`. It **inserts** `obj` at position `index`. To make room, the element that was at `index`, and every element after it, **moves one place to the right**: each one's index goes up by 1. The size goes up by 1. Nothing is lost or overwritten. This version returns nothing (`void`).

Think of people standing in numbered spots. When someone cuts in at spot 1, the person at spot 1 steps to spot 2, the person at spot 2 steps to spot 3, and so on to the end of the line.

For this method, the valid indexes are **0 through `size()`**, one more than for `get` and `set`. Index `size()` is allowed because inserting there means "after the last element," which is the end of the list; nothing has to move. Any index larger than `size()` would leave a gap, so Java refuses it.

KEY IDEA

`list.add(i, obj)` inserts `obj` at index `i`, shifting the element at `i` and everything after it one place right (each index goes up by 1). The index may be anywhere from 0 to `size()`.

WORKED EXAMPLE Letting Dev into the line

InsertMiddle.java

```
1 import java.util.ArrayList;
2
3 public class InsertMiddle
4 {
5     public static void main(String[] args)
6     {
7         ArrayList<String> line = new ArrayList<String>();
8         line.add("Ana");
9         line.add("Ben");
10        line.add("Cy");
11        line.add(1, "Dev");
12        System.out.println(line);
13        System.out.println(line.get(2));
14        System.out.println(line.size());
15    }
16 }
```

Before line 11, the list is:

index:	0	1	2
value:	"Ana"	"Ben"	"Cy"

Line 11, `line.add(1, "Dev")`, puts "Dev" at index 1. "Ben" moves from index 1 to 2, and "Cy" moves from 2 to 3:

```
index:    0      1      2      3
value:   "Ana"  "Dev"  "Ben"  "Cy"
```

```
[Ana, Dev, Ben, Cy]
Ben
4
```

Line 13 shows the shift: `get(2)` was "Cy" before the insert, but now it is "Ben". After an insert, any index you remembered for a later element is out of date.

Now vary the index: insert at the very front, at the very end, and then one past the end.

WORKED EXAMPLE Inserting at the ends

InsertEnds.java

```
1 import java.util.ArrayList;
2
3 public class InsertEnds
4 {
5     public static void main(String[] args)
6     {
7         ArrayList<String> line = new ArrayList<String>();
8         line.add("Ana");
9         line.add("Ben");
10        line.add("Cy");
11        line.add(0, "Eve");
12        System.out.println(line);
13        line.add(line.size(), "Flo");
14        System.out.println(line);
15        line.add(line.size() + 1, "Gus");
16        System.out.println(line);
17    }
18 }
```

Line	Call	Size before	Result
11	<code>add(0, "Eve")</code>	3	[Eve, Ana, Ben, Cy]; all three shift right
13	<code>add(line.size(), "Flo")</code>	4	inserts at index 4, the end; nothing shifts
15	<code>add(line.size() + 1, "Gus")</code>	5	index 6 is past <code>size()</code> ; exception

```
[Eve, Ana, Ben, Cy]
[Eve, Ana, Ben, Cy, Flo]
Exception in thread "main" java.lang.IndexOutOfBoundsException: Index: 6, Size: 5
    at InsertEnds.main(InsertEnds.java:15)
```

Index 0 is always allowed, and so is index `size()`: `line.add(line.size(), "Flo")` does the same thing as `line.add("Flo")`. Index 6 on a list of size 5 would leave index 5 empty, so line 15 throws an `IndexOutOfBoundsException`, and line 16 never runs. (Notice this message is worded differently from the one `get` gives; wording varies by method and Java version, but the exception name is the same.)

The two methods that take an index and an element, `set` and the two-parameter `add`, look almost alike, but they do very different things:

Call on [Ana, Ben, Cy]	Result	Size	Valid indexes
<code>set(1, "Dev")</code>	[Ana, Dev, Cy]	3	0 to <code>size() - 1</code>
<code>add(1, "Dev")</code>	[Ana, Dev, Ben, Cy]	4	0 to <code>size()</code>

WATCH OUT

Using `set` when you mean insert, or the reverse. Both lines below take index 1 and "Dev", but only one keeps "Ben":

SetVsInsert.java

```

1  import java.util.ArrayList;
2
3  public class SetVsInsert
4  {
5      public static void main(String[] args)
6      {
7          ArrayList<String> replaced = new ArrayList<String>();
8          replaced.add("Ana");
9          replaced.add("Ben");
10         replaced.add("Cy");
11         replaced.set(1, "Dev");
12         System.out.println(replaced);
13
14         ArrayList<String> inserted = new ArrayList<String>();
15         inserted.add("Ana");
16         inserted.add("Ben");
17         inserted.add("Cy");
18         inserted.add(1, "Dev");
19         System.out.println(inserted);
20     }
21 }
```

[Ana, Dev, Cy]

[Ana, Dev, Ben, Cy]

Line 11 **replaces** "Ben", so "Ben" is gone and the size stays 3. Line 18 **inserts**, so "Ben" and "Cy" shift right and the size becomes 4. Before you write either one, ask: "Should the element at this index still be in the list afterward?" If yes, use `add(index, obj)`; if it should be replaced, use `set(index, obj)`.

PAUSE AND TRY

Question 7 Tracing

Write the exact output of this program. Sketch the list after each line.

InsertTrace.java

```
1 import java.util.ArrayList;
2
3 public class InsertTrace
4 {
5     public static void main(String[] args)
6     {
7         ArrayList<String> stops = new ArrayList<String>();
8         stops.add("Park");
9         stops.add("Mall");
10        stops.add(1, "Zoo");
11        stops.add(0, "Depot");
12        stops.add(stops.size(), "Bay");
13        System.out.println(stops);
14        System.out.println(stops.get(2));
15    }
16 }
```

4.8.6 Removing an Element

The last thing a grocery list needs is a way to cross items off. When an item is gone, you want the list to close up so there is no hole. With an array that meant shifting elements yourself; an `ArrayList` does it for you.

The method `E remove(int index)` **removes** the element at `index`. Every element after it **moves one place to the left**: each one's index goes down by 1. The size goes down by 1. Like `set`, it **returns the element that was removed**, so you can use it after it leaves the list.

Insert and remove are mirror images: `add(index, obj)` shifts later elements right to open a space, and `remove(index)` shifts them left to close one. Because `remove` takes away an element that exists, its valid indexes are 0 to `size() - 1`, the same as `get` and `set`. An index outside that range throws an `IndexOutOfBoundsException`.

KEY IDEA

`list.remove(i)` takes out the element at index `i`, shifts every later element one place left (each index goes down by 1), shrinks the size by 1, and returns the removed element.

WORKED EXAMPLE Dev leaves the line

RemoveShift.java

```
1 import java.util.ArrayList;
2
3 public class RemoveShift
4 {
5     public static void main(String[] args)
6     {
7         ArrayList<String> line = new ArrayList<String>();
8         line.add("Ana");
9         line.add("Dev");
10        line.add("Ben");
11        line.add("Cy");
12        String gone = line.remove(1);
13        System.out.println(gone);
14        System.out.println(line);
15        System.out.println(line.get(1));
16        System.out.println(line.size());
17    }
18 }
```

Before line 12:

index:	0	1	2	3
value:	"Ana"	"Dev"	"Ben"	"Cy"

Line 12 removes index 1. "Dev" comes out and is stored in `gone`. "Ben" moves from index 2 to 1, and "Cy" from 3 to 2:

index:	0	1	2
value:	"Ana"	"Ben"	"Cy"

```
Dev
[Ana, Ben, Cy]
Ben
3
```

Line 15 asks for index 1 again and now gets "Ben", because "Ben" slid into the space "Dev" left.

Next, vary what happens to the removed element. Since `remove` returns it, you can pass it straight to another method, such as `add` on a different list.

WORKED EXAMPLE Moving people from waiting to served

ServeLine.java

```

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

```

Line	Call	waiting	served
8–10	three appends to waiting	[Ana, Ben, Cy]	
11	served created	[Ana, Ben, Cy]	[]
12	remove(0) returns "Ana"; add it to served	[Ben, Cy]	[Ana]
13	remove(0) returns "Ben"; add it to served	[Cy]	[Ana, Ben]
16	remove(size() - 1) removes "Cy"	[]	[Ana, Ben]

```

[Cy]
[Ana, Ben]
[]
0

```

Lines 12 and 13 both say `remove(0)`, yet they remove different people: after "Ana" leaves, "Ben" shifts to index 0. Line 16 removes the last element; nothing comes after it, so nothing shifts. Its return value is ignored, which is fine. The list is now empty, size 0.

WATCH OUT

Removing two elements with indexes from before the first removal. A coach wants to drop Ben (index 1) and Dev (index 3) from [Ana, Ben, Cy, Dev, Eve], and writes:

RemoveTwoWrong.java

```
1 import java.util.ArrayList;
2
3 public class RemoveTwoWrong
4 {
5     public static void main(String[] args)
6     {
7         ArrayList<String> team = new ArrayList<String>();
8         team.add("Ana");
9         team.add("Ben");
10        team.add("Cy");
11        team.add("Dev");
12        team.add("Eve");
13        team.remove(1);
14        team.remove(3);
15        System.out.println(team);
16    }
17 }
```

[Ana, Cy, Dev]

Dev is still on the team, and Eve was dropped instead. The coach looked up both indexes in the **original** list [Ana, Ben, Cy, Dev, Eve]. But line 13 removed Ben, and every later element shifted left: Cy to 1, Dev to 2, Eve to 3. So line 14's `remove(3)` removed Eve.

One fix is to remove the **higher** index first. Removing index 3 shifts only the elements after it, so index 1 still means Ben:

RemoveTwoFixed.java

```
1 import java.util.ArrayList;
2
3 public class RemoveTwoFixed
4 {
5     public static void main(String[] args)
6     {
7         ArrayList<String> team = new ArrayList<String>();
8         team.add("Ana");
9         team.add("Ben");
10        team.add("Cy");
11        team.add("Dev");
12        team.add("Eve");
13        team.remove(3);
14        team.remove(1);
15        System.out.println(team);
16    }
17 }
```

[Ana, Cy, Eve]

The general rule: after any `add(index, obj)` or `remove(index)`, the indexes of later elements are different. Work out each index from the list as it is **at that moment**.

PAUSE AND TRY**Question 8** Tracing

Write the exact output of this program.

RemoveTrace.java

```
1 import java.util.ArrayList;
2
3 public class RemoveTrace
4 {
5     public static void main(String[] args)
6     {
7         ArrayList<String> books = new ArrayList<String>();
8         books.add("Holes");
9         books.add("Wonder");
10        books.add("Dune");
11        books.add("Matilda");
12        String a = books.remove(0);
13        String b = books.remove(books.size() - 1);
14        books.add(1, a);
15        System.out.println(b);
16        System.out.println(books);
17        System.out.println(books.get(1));
18    }
19 }
```

Question 9 Debugging

The list starts as [email, lunch, report, nap]. The programmer wants to remove lunch and report, leaving [email, nap], but the program prints [email, report]. Explain the bug and give a corrected version of lines 12 and 13.

CleanupBug.java

```
1 import java.util.ArrayList;
2
3 public class CleanupBug
4 {
5     public static void main(String[] args)
6     {
7         ArrayList<String> tasks = new ArrayList<String>();
8         tasks.add("email");
9         tasks.add("lunch");
10        tasks.add("report");
11        tasks.add("nap");
12        tasks.remove(1);
13        tasks.remove(2);
14        System.out.println(tasks);
15    }
16 }
```

4.8.7 The Element Type and Wrapper Classes

You have used the type parameter since 4.8.1, writing `<String>` without much thought. Now look at why it is there, and what happens when you want a list of numbers.

The compiler checks the element type

When you write `ArrayList<String>`, every method that takes or returns an element uses `String` in place of `E`. So for that list, `add` expects a `String`, `set` expects a `String` and returns a `String`, and `get` and `remove` return `Strings`. The compiler knows this, so it can catch a wrong type **before the program ever runs**.

WORKED EXAMPLE Two type mistakes the compiler catches

WrongType.java

```
1 import java.util.ArrayList;
2
3 public class WrongType
4 {
5     public static void main(String[] args)
6     {
7         ArrayList<String> names = new ArrayList<String>();
8         names.add("Rosa");
9         names.add(42);
10        int letters = names.get(0);
11        System.out.println(letters);
12    }
13 }
```

```
WrongType.java:9: error: incompatible types: int cannot be converted to String
    names.add(42);
             ^
```

```
WrongType.java:10: error: incompatible types: String cannot be converted to int
    int letters = names.get(0);
                      ^
```

Line 9 tries to put an `int` into a list of `Strings`. Line 10 tries to store what `get` returns, a `String`, in an `int` variable. Both are caught at compile time because the compiler knows the element type is `String`. (Error wording can vary between Java versions.)

Java does let you write `ArrayList` with no type parameter at all, as in

`ArrayList list = new ArrayList();`. Such a list accepts any kind of object, so the compiler cannot check what goes in or what comes out. A mistake like line 9 would compile, and the problem would only surface later, as an error while the program runs, which is harder to track down. That is why `ArrayList<E>` with a real type is always preferred: it lets the compiler find errors that would otherwise show up only at run time. Always write the type parameter.

Lists of numbers need wrapper classes

An `ArrayList` holds **object references**, not primitive values. (You will see what that means for objects in 4.8.8.) The type parameter must therefore be a class, never a primitive type such as `int` or `double`:

IntList.java

```
1 import java.util.ArrayList;
2
3 public class IntList
4 {
5     public static void main(String[] args)
6     {
7         ArrayList<int> scores = new ArrayList<int>();
8         scores.add(88);
9         System.out.println(scores);
10    }
11 }
```

```
IntList.java:7: error: unexpected type
        ArrayList<int> scores = new ArrayList<int>();
                    ^
required: reference
found:    int
```

The message says the compiler needed a reference type and found the primitive `int` (and it reports the same error for the second `<int>`). This is exactly the job the wrapper classes from Topic 4.7 were made for. Use `Integer` for whole numbers and `Double` for decimals:

```
1 ArrayList<Integer> scores = new ArrayList<Integer>();
```

Then autoboxing and unboxing from Topic 4.7 do the conversions for you. When you pass an `int` to `add`, which expects an `Integer`, Java boxes it into an `Integer` automatically. When you assign the `Integer` that `get` returns to an `int` variable, Java unboxes it.

KEY IDEA

The element type of an `ArrayList` must be a class. For numbers, use `ArrayList<Integer>` or `ArrayList<Double>`; autoboxing turns an `int` or `double` into an object when you add it, and unboxing turns it back when you store the result of `get`, `set`, or `remove` in a primitive variable.

WORKED EXAMPLE A list of test scores

ScoresList.java

```

1  import java.util.ArrayList;
2
3  public class ScoresList
4  {
5      public static void main(String[] args)
6      {
7          ArrayList<Integer> scores = new ArrayList<Integer>();
8          scores.add(88);
9          scores.add(92);
10         scores.add(75);
11         int first = scores.get(0);
12         int total = scores.get(1) + scores.get(2);
13         System.out.println(first);
14         System.out.println(total);
15         int oldScore = scores.set(2, 80);
16         System.out.println(oldScore);
17         System.out.println(scores);
18     }
19 }

```

Line	What Java does	scores
8–10	Each int (88, 92, 75) is autoboxed into an Integer and appended	[88, 92, 75]
11	get(0) returns the Integer 88; assigning it to an int unboxes it	[88, 92, 75]
12	get(1) and get(2) are unboxed for +, giving 92 + 75 = 167	[88, 92, 75]
15	80 is autoboxed; set returns the old Integer 75, unboxed into oldScore	[88, 92, 80]

```

88
167
75
[88, 92, 80]

```

Double works the same way. This example averages two temperatures and inserts a third at the front.

WORKED EXAMPLE A list of temperatures

TemperatureList.java

```
1 import java.util.ArrayList;
2
3 public class TemperatureList
4 {
5     public static void main(String[] args)
6     {
7         ArrayList<Double> temps = new ArrayList<Double>();
8         temps.add(71.5);
9         temps.add(68.0);
10        double average = (temps.get(0) + temps.get(1)) / temps.size();
11        System.out.println(average);
12        temps.add(0, 75.25);
13        System.out.println(temps);
14    }
15 }
```

Line 10 unboxes 71.5 and 68.0, adds them to get 139.5, and divides by `temps.size()`, which is 2. A double divided by an int is a double (Topic 1.3), so average is 69.75. Line 12 boxes 75.25 and inserts it at index 0, shifting the other two right.

```
69.75
[75.25, 71.5, 68.0]
```

Notice 68.0 prints with its .0: it is a Double, so it prints like a double.

WATCH OUT

Expecting remove to remove a value. In a list of Integers, the number you pass to `remove` still means an **index**, not "the element equal to this number":

RemoveIndexNotValue.java

```
1 import java.util.ArrayList;
2
3 public class RemoveIndexNotValue
4 {
5     public static void main(String[] args)
6     {
7         ArrayList<Integer> nums = new ArrayList<Integer>();
8         nums.add(40);
9         nums.add(2);
10        nums.add(7);
11        nums.add(1);
12        int removed = nums.remove(1);
13        System.out.println(removed);
14        System.out.println(nums);
15    }
16 }
```

```
2  
[40, 7, 1]
```

The programmer wanted to remove the value 1, but `nums.remove(1)` removed whatever was at **index** 1, which was 2. The 1 at the end of the list is untouched. Students make this mistake because the list holds numbers and the argument is a number, so it reads like "remove 1." The `remove` method in this course always takes an index. To remove the 1 here, use its index: `nums.remove(3)`.

PAUSE AND TRY

Question 10 Debugging

This program does not compile. Explain the error and write a corrected line 9.

PricesBug.java

```
1 import java.util.ArrayList;  
2  
3 public class PricesBug  
4 {  
5     public static void main(String[] args)  
6     {  
7         ArrayList<Double> prices = new ArrayList<Double>();  
8         prices.add(2.5);  
9         prices.add(3);  
10        System.out.println(prices);  
11    }  
12 }
```

```
PricesBug.java:9: error: incompatible types: int cannot be converted to Double  
    prices.add(3);  
               ^
```

Question 11 Tracing

Write the exact output of this program.

NumbersTrace.java

```
1 import java.util.ArrayList;
2
3 public class NumbersTrace
4 {
5     public static void main(String[] args)
6     {
7         ArrayList<Integer> laps = new ArrayList<Integer>();
8         laps.add(5);
9         laps.add(3);
10        laps.add(0, 9);
11        int gone = laps.remove(2);
12        int changed = laps.set(0, gone + laps.get(1));
13        System.out.println(gone);
14        System.out.println(changed);
15        System.out.println(laps);
16    }
17 }
```

4.8.8 A List Holds References to Objects

In Topic 3.6 you learned that a variable of a class type holds a **reference** to an object, like an address, and that copying the variable copies the address, not the object. An `ArrayList` follows the same rule. Each element of a list is a reference. When you add an object, the list stores a copy of the **reference**, so the list and your variable now point to the **same object**. No copy of the object is made.

This matters as soon as the objects can change. Here is a `Player` class from Unit 3 with a method that changes the player's points. (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.)

KEY IDEA

An `ArrayList` stores references to objects. Adding an object to a list does not copy it, so a change made through the list (`team.get(0).addPoints(3)`) and a change made through another variable that refers to the same object are changes to one object.

WORKED EXAMPLE One player, two ways to reach it

TeamRefs.java

```
1 import java.util.ArrayList;
2
3 class Player
4 {
5     private String name;
6     private int points;
7
8     public Player(String name, int points)
```

```

 9      {
10          this.name = name;
11          this.points = points;
12      }
13
14      public String getName()
15      {
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 TeamRefs
31  {
32      public static void main(String[] args)
33      {
34          ArrayList<Player> team = new ArrayList<Player>();
35          Player kai = new Player("Kai", 10);
36          team.add(kai);
37          team.add(new Player("Mo", 4));
38          kai.addPoints(5);
39          System.out.println(team.get(0).getPoints());
40          team.get(0).addPoints(3);
41          System.out.println(kai.getPoints());
42          System.out.println(team.get(1).getName());
43      }
44  }

```

After lines 34 through 37, memory looks like this:

```

kai  —————→ Player "Kai", 10
team → index 0: ref —→ (the same Player as kai)
      index 1: ref —→ Player "Mo", 4

```

Line	What happens	Kai's points
36	team.add(kai) stores a copy of the reference in team	10
37	A new Player is created and its reference is added; no variable names it	10
38	kai.addPoints(5) changes the shared object	15
39	team.get(0) returns the same reference, so getPoints() gives 15	15
40	team.get(0).addPoints(3) changes the shared object through the list	18
41	kai.getPoints() sees that change	18

```
15
18
Mo
```

Line 40 chains two calls: `team.get(0)` returns a reference to Kai's `Player`, and `.addPoints(3)` is called on that object. Line 42 reaches the second player, which has no variable of its own; the list is the only way to get to it.

Now vary two things: add the **same** reference twice, and use `set` to replace a player. `set` replaces the reference stored in the list; it does not destroy the old object, and it returns a reference to it.

WORKED EXAMPLE The same player twice, then a replacement

SetOldPlayer.java

```
1  import java.util.ArrayList;
2
3  class Player
4  {
5      private String name;
6      private int points;
7
8      public Player(String name, int points)
9      {
10         this.name = name;
11         this.points = points;
12     }
13
14     public String getName()
15     {
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 SetOldPlayer
31 {
32     public static void main(String[] args)
33     {
34         ArrayList<Player> team = new ArrayList<Player>();
35         Player star = new Player("Kai", 10);
36         team.add(star);
37         team.add(star);
38         team.get(1).addPoints(4);
39         System.out.println(team.get(0).getPoints());
```

```

40     Player benched = team.set(0, new Player("Lu", 0));
41     System.out.println(benched.getName());
42     System.out.println(benched.getPoints());
43     System.out.println(team.get(0).getName());
44     System.out.println(team.get(1).getName());
45 }
46 }

```

Line	What happens	Index 0	Index 1
36–37	The reference in <code>star</code> is added twice; both elements point to one <code>Player</code>	Kai, 10	Kai, 10
38	<code>team.get(1).addPoints(4)</code> changes that one <code>Player</code>	Kai, 14	Kai, 14
40	<code>set(0, new Player("Lu", 0))</code> puts Lu's reference at index 0 and returns Kai's	Lu, 0	Kai, 14

```

14
Kai
14
Lu
Kai

```

Line 39 prints 14 even though the points were added through index 1: both indexes held the same reference. On line 40, `benched` receives the reference that used to be at index 0, so lines 41 and 42 can still use Kai's object. Kai is still in the list at index 1, because only index 0 was replaced.

WATCH OUT

Thinking that reassigning a variable changes the list. After adding a `String`, a student assigns a new value to the variable and expects the list to follow:

StringReassign.java

```

1 import java.util.ArrayList;
2
3 public class StringReassign
4 {
5     public static void main(String[] args)
6     {
7         ArrayList<String> groceries = new ArrayList<String>();
8         String item = "milk";
9         groceries.add(item);
10        item = "oat milk";
11        System.out.println(item);
12        System.out.println(groceries);
13    }
14 }

```

```

oat milk
[milk]

```

Line 9 stores a copy of the reference in `item`, which points to `"milk"`. Line 10 makes `item` point to a **different** object, `"oat milk"`. That changes what the variable holds, not what the list holds; the list still points to `"milk"`. This is different from `kai.addPoints(5)` in `TeamRefs.java`, which changed the object itself. (A `String` object can never be changed at all, as you learned in Unit 1.) To change what the list holds, call a list method: `groceries.set(0, "oat milk");` makes the list print `[oat milk]`.

PAUSE AND TRY

Question 12 Tracing

Using the same `Player` class, write the exact output of this program.

`RefsTrace.java`

```
1 import java.util.ArrayList;
2
3 class Player
4 {
5     private String name;
6     private int points;
7
8     public Player(String name, int points)
9     {
10         this.name = name;
11         this.points = points;
12     }
13
14     public String getName()
15     {
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 RefsTrace
31 {
32     public static void main(String[] args)
33     {
34         ArrayList<Player> squad = new ArrayList<Player>();
35         Player a = new Player("Rin", 2);
36         Player b = new Player("Sol", 6);
37         squad.add(a);
38         squad.add(b);
```

```
39     squad.add(a);
40     a.addPoints(1);
41     squad.get(1).addPoints(10);
42     Player out = squad.remove(0);
43     out.addPoints(5);
44     System.out.println(squad.get(0).getPoints());
45     System.out.println(squad.get(1).getPoints());
46     System.out.println(squad.size());
47 }
48 }
```

Putting it together

A playlist is a natural `ArrayList`: songs are added, one can be bumped up to play next, and skipping a song sends it to the back. This `Playlist` class keeps its songs in a private `ArrayList<String>` instance variable and uses every method from the chapter. Before reading the trace, predict all five lines of output.

WORKED EXAMPLE A road-trip playlist

PlaylistDemo.java

```
1  import java.util.ArrayList;
2
3  class Playlist
4  {
5      private ArrayList<String> songs;
6
7      public Playlist()
8      {
9          songs = new ArrayList<String>();
10     }
11
12     public void addSong(String title)
13     {
14         songs.add(title);
15     }
16
17     public void playNext(String title)
18     {
19         songs.add(1, title);
20     }
21
22     public String skip()
23     {
24         String skipped = songs.remove(0);
25         songs.add(skipped);
26         return skipped;
27     }
28
29     public String nowPlaying()
```

```

30     {
31         return songs.get(0);
32     }
33
34     public int getCount()
35     {
36         return songs.size();
37     }
38
39     public void show()
40     {
41         System.out.println(songs);
42     }
43 }
44
45 public class PlaylistDemo
46 {
47     public static void main(String[] args)
48     {
49         Playlist trip = new Playlist();
50         trip.addSong("Rain");
51         trip.addSong("Glow");
52         trip.addSong("Tides");
53         trip.playNext("Ember");
54         trip.show();
55         System.out.println(trip.skip());
56         System.out.println(trip.nowPlaying());
57         trip.show();
58         System.out.println(trip.getCount());
59     }
60 }

```

Here is what each method does, in terms of the list:

Method	Line	List call	Effect
constructor	9	<code>new ArrayList<String>()</code>	starts with an empty list
<code>addSong</code>	14	<code>songs.add(title)</code>	appends to the end
<code>playNext</code>	19	<code>songs.add(1, title)</code>	inserts right after the current song
<code>skip</code>	24–26	<code>songs.remove(0)</code> , then <code>songs.add(skipped)</code>	moves the first song to the back and returns it
<code>nowPlaying</code>	31	<code>songs.get(0)</code>	returns the first song
<code>getCount</code>	36	<code>songs.size()</code>	returns the number of songs
<code>show</code>	41	<code>System.out.println(songs)</code>	prints the list in brackets

Now trace main:

Line	Call	songs afterward	Printed
49	<code>new Playlist()</code>	<code>[]</code>	
50–52	three <code>addSong</code> calls	<code>[Rain, Glow, Tides]</code>	

53	<code>playNext("Ember")</code>	<code>[Rain, Ember, Glow, Tides]</code>	
54	<code>show()</code>	<code>[Rain, Ember, Glow, Tides]</code>	<code>[Rain, Ember, Glow, Tides]</code>
55	<code>skip(): remove index 0 ("Rain"), append it, return it</code>	<code>[Ember, Glow, Tides, Rain]</code>	<code>Rain</code>
56	<code>nowPlaying()</code>	<code>[Ember, Glow, Tides, Rain]</code>	<code>Ember</code>
57	<code>show()</code>	<code>[Ember, Glow, Tides, Rain]</code>	<code>[Ember, Glow, Tides, Rain]</code>
58	<code>getCount()</code>	<code>[Ember, Glow, Tides, Rain]</code>	<code>4</code>

```
[Rain, Ember, Glow, Tides]
Rain
Ember
[Ember, Glow, Tides, Rain]
4
```

Three things to notice:

- On line 53, inserting at index 1 shifts "Glow" and "Tides" right; nothing is overwritten. If `playNext` had used `set(1, title)`, "Glow" would be lost.
- In `skip`, the returned value of `remove(0)` is used twice: it is appended to the end and also returned to `main`. Removing shifts everyone left, so "Ember" becomes the new song at index 0.
- `main` never touches the `ArrayList` directly. The list is a private instance variable, and the methods decide which list calls are allowed, just like the classes you designed in Unit 3.

Common mistakes

The `ArrayList` methods look alike, and the most common errors come from mixing them up or from using an index that is out of range. Use this table as a quick lookup:

Call	Valid indexes	Size afterward	Returns
<code>size()</code>	(none)	same	number of elements
<code>add(obj)</code>	(none)	+1	true
<code>add(i, obj)</code>	0 to <code>size()</code>	+1	nothing (void)
<code>get(i)</code>	0 to <code>size() - 1</code>	same	the element at <code>i</code>
<code>set(i, obj)</code>	0 to <code>size() - 1</code>	same	the old element
<code>remove(i)</code>	0 to <code>size() - 1</code>	-1	the removed element

An index outside the valid range throws an `IndexOutOfBoundsException` when the program runs.

Vocabulary

Term	Definition	Example
ArrayList	A class whose objects are ordered lists that can grow and shrink while a program runs	<code>ArrayList<String> groceries</code>
mutable in size	Able to change how many elements it holds after it is created	<code>add</code> and <code>remove</code> change <code>size()</code>
java.util	The package that contains <code>ArrayList</code>	<code>import java.util.ArrayList;</code>
import statement	A line at the top of a file that makes a class from another package available	<code>import java.util.ArrayList;</code>
ArrayList constructor	<code>ArrayList()</code> , which builds an empty list	<code>new ArrayList<String>()</code>
empty list	A list with no elements; its size is 0 and it prints as <code>[]</code>	a newly constructed list
type parameter	The type in angle brackets that says what kind of element a list holds; written <code>E</code> in the documentation	<code>String</code> in <code>ArrayList<String></code>
generic	Describes a class that takes a type parameter	<code>ArrayList<E></code>
append	To add an element at the end of a list	<code>groceries.add("milk")</code>
index	An element's position number, from 0 to <code>size() - 1</code>	<code>"eggs"</code> is at index 1
insert	To add an element at a given index, shifting later elements right	<code>line.add(1, "Dev")</code>
IndexOutOfBoundsException	The runtime error thrown when a list method gets an index outside its valid range	<code>get(3)</code> on a list of size 3

Chapter summary

- An `ArrayList` is an ordered list that can grow and shrink. It is in the `java.util` package, so a file that uses it needs `import java.util.ArrayList;`. The constructor `new ArrayList<String>()` makes an empty list.
- `ArrayList<E>` names the element type. The methods use that type for their parameters and return values, so the compiler can catch a wrong type before the program runs. Always write the type; numbers go in `ArrayList<Integer>` or `ArrayList<Double>`, with autoboxing and unboxing doing the conversions.
- `size()` counts the elements, `add(obj)` appends, `get(i)` returns an element, and `set(i, obj)` replaces one and returns the old element.
- `add(i, obj)` inserts and shifts later elements right; `remove(i)` removes, shifts later elements left, and returns the removed element. Either one changes the indexes of every later element.
- Indexes run from 0 to `size() - 1` (up to `size()` for inserting). Any other index throws an `IndexOutOfBoundsException`.
- A list holds references to objects, so changing an object through the list or through another variable changes the same object.

Review questions

Question 13 Multiple choice

names is an ArrayList<String> holding [Ali, Bo, Cam]. Which call throws an IndexOutOfBoundsException?

- A. names.add(3, "Dee")
- B. names.set(3, "Dee")
- C. names.get(2)
- D. names.add("Dee")

Question 14 Tracing

This program uses every list method. Write the exact output.

MixedOps.java

```
1 import java.util.ArrayList;
2
3 public class MixedOps
4 {
5     public static void main(String[] args)
6     {
7         ArrayList<String> colors = new ArrayList<String>();
8         colors.add("red");
9         colors.add("blue");
10        colors.add("green");
11        colors.add(1, "pink");
12        String x = colors.set(2, "gold");
13        String y = colors.remove(0);
14        colors.add(y.toUpperCase());
15        System.out.println(x + " " + y);
16        System.out.println(colors);
17        System.out.println(colors.get(colors.size() - 2));
18    }
19 }
```

Question 15 Debugging

The programmer wants to print the name of the person who signed up most recently. The program prints 3 and then crashes. Explain the bug and write a corrected line 12.

LastNameBug.java

```
1 import java.util.ArrayList;
2
3 public class LastNameBug
4 {
5     public static void main(String[] args)
6     {
7         ArrayList<String> signups = new ArrayList<String>();
8         signups.add("Ada");
9         signups.add("Grace");
10        signups.add("Alan");
11        System.out.println(signups.size());
12        String newest = signups.get(signups.size());
13        System.out.println(newest);
14    }
15 }
```

```
3
Exception in thread "main" java.lang.IndexOutOfBoundsException: Index 3 out of bounds
for length 3
    at LastNameBug.main(LastNameBug.java:12)
```

Question 16 Multiple choice

What does this program print?

RemoveThenGet.java

```
1 import java.util.ArrayList;
2
3 public class RemoveThenGet
4 {
5     public static void main(String[] args)
6     {
7         ArrayList<Integer> vals = new ArrayList<Integer>();
8         vals.add(10);
9         vals.add(20);
10        vals.add(30);
11        int result = vals.remove(1) + vals.get(1);
12        System.out.println(result);
13        System.out.println(vals);
14    }
15 }
```

- A. 50 then [10, 30]
- B. 40 then [10, 30]
- C. 50 then [10, 20, 30]
- D. Nothing; the result of `remove` cannot be added to the result of `get`

Question 17 Tracing

In Topic 3.6 you learned that passing an object to a method passes a copy of its reference. Use that idea to trace this program.

ClubRoster.java

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

Question 18 Short answer

A student asks two questions: "Why can't I write `ArrayList<int>`?" and "Why bother writing `<Integer>` at all, if Java lets me write plain `ArrayList`?" Answer both.

Question 19 Tracing

This program uses the `Playlist` class from *Putting it together* with a different `main`. Write the exact output.

PlaylistReview.java

```
1 import java.util.ArrayList;
2
3 class Playlist
4 {
5     private ArrayList<String> songs;
6
7     public Playlist()
8     {
9         songs = new ArrayList<String>();
10    }
11
12    public void addSong(String title)
13    {
14        songs.add(title);
15    }
16
17    public void playNext(String title)
18    {
19        songs.add(1, title);
20    }
21 }
```

```
22     public String skip()
23     {
24         String skipped = songs.remove(0);
25         songs.add(skipped);
26         return skipped;
27     }
28
29     public String nowPlaying()
30     {
31         return songs.get(0);
32     }
33
34     public int getCount()
35     {
36         return songs.size();
37     }
38
39     public void show()
40     {
41         System.out.println(songs);
42     }
43 }
44
45 public class PlaylistReview
46 {
47     public static void main(String[] args)
48     {
49         Playlist mix = new Playlist();
50         mix.addSong("Echo");
51         mix.addSong("Bloom");
52         mix.playNext("Drift");
53         System.out.println(mix.skip());
54         mix.playNext("Spark");
55         mix.show();
56         System.out.println(mix.nowPlaying());
57         System.out.println(mix.getCount());
58     }
59 }
```

Reflection

- ☐ I can import ArrayList and create an empty list of a chosen element type. Reread: 4.8.1 Creating an Empty ArrayList
- ☐ I can use add and size to build a list and say how many elements it has. Reread: 4.8.2 Adding to the End and Counting
- ☐ I can find an element with get, name the valid indexes, and explain an IndexOutOfBoundsException. Reread: 4.8.3 Indexes and get
- ☐ I can replace an element with set and say what set returns. Reread: 4.8.4 Replacing an Element with set
- ☐ I can insert an element at an index and show how later elements shift. Reread: 4.8.5 Inserting in the Middle

- ☐ I can remove an element, use what remove returns, and keep track of shifted indexes. Reread: 4.8.6 Removing an Element
- ☐ I can store numbers in an ArrayList with Integer or Double and explain why the element type matters. Reread: 4.8.7 The Element Type and Wrapper Classes
- ☐ I can trace code where a list and a variable refer to the same object. Reread: 4.8.8 A List Holds References to Objects

Check Your Reasoning

Answers and explanations for every question, numbered to match.

Question 1

B. The `ArrayList` constructor takes no size and always builds an empty list. Elements appear only when you add them.

- **A** is wrong: it mixes up an `ArrayList` with an array. `new double[10]` makes 10 slots filled with `0.0` (Topic 4.3), but an `ArrayList` starts with no slots at all.
- **C** is wrong: an empty list has nothing in it, not even `null`. Printing it shows `[]`.
- **D** is wrong: `new ArrayList<Double>()` with empty parentheses is the complete constructor call, and the list is ready to use.

Reread 4.8.1 *Creating an Empty ArrayList* if you chose differently.

Question 2

Model answer: `String` and `Math` belong to the `java.lang` package, which every program can use without an import. `ArrayList` belongs to the `java.util` package, so the program must say `import java.util.ArrayList;` at the top of the file. Without it, Java reports `cannot find symbol for ArrayList`.

A complete answer says:

- `ArrayList` is in the `java.util` package.
- Classes in `java.lang` (like `String` and `Math`) are available automatically; classes in other packages must be imported.

Reread 4.8.1 *Creating an Empty ArrayList* if your answer was missing one of these.

Question 3

```
0
2
4
[cat, dog, cat, fish]
```

Line	What happens	<code>pets</code>	<code>size()</code>
7	empty list created	<code>[]</code>	0
8	prints the size	<code>[]</code>	0
9–10	two appends	<code>[cat, dog]</code>	2
11	prints the size	<code>[cat, dog]</code>	2
12–13	two more appends	<code>[cat, dog, cat, fish]</code>	4
14	prints the size	<code>[cat, dog, cat, fish]</code>	4

The second `"cat"` is a separate element, so it counts toward the size and appears again in the printed list. If you got 3, you assumed a list skips duplicates; it does not.

Reread 4.8.2 *Adding to the End and Counting* if your output differed.

Question 4

```
Cairo
0s
8
```

The list is `[Lima, Oslo, Cairo, Pune]` with size 4.

Line	Work	Printed
12	<code>cities.size() - 2</code> is 2, and <code>get(2)</code> is "Cairo"	
13	<code>prints pick</code>	Cairo
14	<code>get(1)</code> is "Oslo"; <code>substring(0, 2)</code> is "0s"	0s
15	<code>get(0).length()</code> is 4 ("Lima"); plus <code>size()</code> 4 gives 8	8

If you printed `0slo` first, you counted from 1 instead of 0. On line 15 both operands are `ints`, so `+` adds them.

Reread 4.8.3 *Indexes and get* if your output differed.

Question 5

D. The list has 4 elements, so its valid indexes are 0 through 3. Index 4 is past the end.

- **A** is wrong: index 0 is the first element, "Mon".
- **B** is wrong: index 3 is the last element, "Thu". It is the largest valid index, not an invalid one.
- **C** is wrong: `days.size() - 1` is 3, the same as choice B, so it returns "Thu".

Reread 4.8.3 *Indexes and get* if you chose differently.

Question 6

```
Kim
Ira
[Kim, Jo, Lee]
```

Line	Call	Returned	seats
8–10	three appends		[Ira, Jo, Kim]
11	<code>set(2, "Lee")</code>	"Kim" into first	[Ira, Jo, Lee]
12	<code>set(0, first)</code>	"Ira" into second	[Kim, Jo, Lee]

Line 12 puts the value of `first` ("Kim") at index 0 and returns the element it replaced, "Ira". If you printed `Lee` first, you thought `set` returns the new element; it returns the old one.

Reread 4.8.4 *Replacing an Element with set* if your output differed.

Question 7

```
[Depot, Park, Zoo, Mall, Bay]
Zoo
```

Line	Call	stops afterward
8–9	two appends	[Park, Mall]

10	<code>add(1, "Zoo")</code>	<code>[Park, Zoo, Mall]</code>
11	<code>add(0, "Depot")</code>	<code>[Depot, Park, Zoo, Mall]</code>
12	<code>add(stops.size(), "Bay")</code>	<code>[Depot, Park, Zoo, Mall, Bay]</code>

On line 11, inserting at the front shifts every element, so "Zoo" moves from index 1 to index 2, which is why line 14 prints `Zoo`. On line 12, `stops.size()` is 4, and inserting at index 4 appends.

Reread 4.8.5 *Inserting in the Middle* if your output differed.

Question 8

```
Matilda
[Wonder, Holes, Dune]
Holes
```

Line	Call	Returned	books afterward
8–11	four appends		<code>[Holes, Wonder, Dune, Matilda]</code>
12	<code>remove(0)</code>	"Holes" into a	<code>[Wonder, Dune, Matilda]</code>
13	<code>remove(books.size() - 1)</code> , which is <code>remove(2)</code>	"Matilda" into b	<code>[Wonder, Dune]</code>
14	<code>add(1, a)</code>		<code>[Wonder, Holes, Dune]</code>

On line 13, `books.size()` is 3 (not 4), because line 12 already shrank the list. If you thought line 13 removed index 3, you used the size from before the first removal.

Reread 4.8.6 *Removing an Element* if your output differed.

Question 9

Line 12 removes `lunch` at index 1, and the later elements shift left: `report` moves to index 1 and `nap` to index 2. Line 13's `remove(2)` then removes `nap`, not `report`. The programmer used index 2 from the original list, but it was out of date after the first removal.

Two corrections work:

```
1      tasks.remove(2);
2      tasks.remove(1);
```

Removing the higher index first does not disturb index 1. Or, since `report` slides into index 1 after the first removal:

```
1      tasks.remove(1);
2      tasks.remove(1);
```

Either version prints `[email, nap]`. Java gave no error for the original, because index 2 was still valid; it just held a different element.

Reread 4.8.6 *Removing an Element* if you missed it.

Question 10

The list's element type is `Double`, so `add` expects a `Double`. The literal 3 is an `int`. Autoboxing only wraps a primitive in **its own** wrapper class: an `int` becomes an `Integer`, and only a `double`

becomes a `Double`. Java will not turn an `int` into a `Double`, so it reports incompatible types. (Line 8 compiles because `2.5` is a `double`.)

Corrected line 9:

```
1      prices.add(3.0);
```

`3.0` is a `double`, so it is autoboxed into a `Double`, and the program prints `[2.5, 3.0]`.

Reread 4.8.7 *The Element Type and Wrapper Classes* if you missed it.

Question 11

```
3
9
[8, 5]
```

Line	Call	Returned	Laps afterward
8–9	two appends		[5, 3]
10	add(0, 9)		[9, 5, 3]
11	remove(2)	3, into gone	[9, 5]
12	set(0, gone + laps.get(1)), which is set(0, 3 + 5)	9, into changed	[8, 5]

On line 12, Java evaluates the argument first: `laps.get(1)` is 5 (unboxed), so the new value is 8. `set` then returns the old value at index 0, which is 9. If you wrote 8 for `changed`, you thought `set` returns the new element.

Reread 4.8.7 *The Element Type and Wrapper Classes* and 4.8.4 *Replacing an Element with set* if your output differed.

Question 12

```
16
8
2
```

Two `Player` objects exist: `Rin` (a) and `Sol` (b). The list holds three references: `[Rin, Sol, Rin]`.

Line	What happens	Rin	Sol	squad
39	after three adds	2	6	[Rin, Sol, Rin]
40	a.addPoints(1)	3	6	[Rin, Sol, Rin]
41	squad.get(1) is Sol; add 10	3	16	[Rin, Sol, Rin]
42	remove(0) removes one reference to Rin and returns it into out	3	16	[Sol, Rin]
43	out.addPoints(5) changes Rin's object	8	16	[Sol, Rin]

Line 44 prints `Sol`'s points (16), line 45 prints index 1, which is still a reference to `Rin` (8), and line 46 prints the size, 2. Removing a reference from the list does not destroy the object, and the other reference to `Rin` still sees the change.

Reread 4.8.8 *A List Holds References to Objects* if your output differed.

Question 13

B. `set` replaces an existing element, so its index must be 0 to `size() - 1`, which is 0 to 2 here. There is no element at index 3 to replace.

- **A** is wrong: inserting allows indexes 0 to `size()`, and `size()` is 3, so `add(3, "Dee")` appends "Dee" at the end.
- **C** is wrong: index 2 is the last element, "Cam".
- **D** is wrong: `add(obj)` takes no index and always appends.

Reread 4.8.4 *Replacing an Element with set* and 4.8.5 *Inserting in the Middle* if you chose differently.

Question 14

```
blue red
[pink, gold, green, RED]
green
```

Line	Call	Returned	colors afterward
8–10	three appends		[red, blue, green]
11	<code>add(1, "pink")</code>		[red, pink, blue, green]
12	<code>set(2, "gold")</code>	"blue" into x	[red, pink, gold, green]
13	<code>remove(0)</code>	"red" into y	[pink, gold, green]
14	<code>add(y.toUpperCase())</code>		[pink, gold, green, RED]

Line 15 concatenates `x`, a space, and `y`. Line 17: `size()` is 4, so `get(2)` is "green". The key step is line 12: after the insert on line 11, index 2 holds "blue", not "green".

Reread 4.8.5 *Inserting in the Middle* and 4.8.6 *Removing an Element* if your output differed.

Question 15

The most recent sign-up is the last element, because `add` appends. With 3 elements, the indexes are 0, 1, and 2, so the last index is `size() - 1`, which is 2. `signups.size()` is 3, which is one past the end, so `get` throws an `IndexOutOfBoundsException`.

Corrected line 12:

```
1      String newest = signups.get(signups.size() - 1);
```

The program then prints 3 and Alan.

Reread 4.8.3 *Indexes and get* if you missed it.

Question 16

A. Java evaluates `vals.remove(1)` first. It removes the element at index 1, 20, returns it, and shifts 30 left to index 1. Then `vals.get(1)` returns 30. Both `Integer`s are unboxed for `+`, so result is `20 + 30 = 50`, and the list is `[10, 30]`.

- **B** is wrong: 40 comes from thinking `get(1)` still finds 20, as if nothing shifted. After the removal, index 1 holds 30.
- **C** is wrong: `remove` really takes the element out of the list; it does not just look at it.
- **D** is wrong: both calls return `Integer`s, which are unboxed into `int`s for `+`, so the line compiles.

Reread 4.8.6 *Removing an Element* and 4.8.7 *The Element Type and Wrapper Classes* if you chose differently.

Question 17

```
[Zoe, Ivy]
2
```

Line	What happens	The one list
12–14	<code>cClub</code> is created and two names are appended	[Ivy, Max]
15	<code>addHonor(cClub, "Zoe")</code> : <code>roster</code> gets a copy of the reference, so it is the same list	[Ivy, Max]
7	<code>roster.add(0, name)</code> inserts at the front of that list	[Zoe, Ivy, Max]
16	<code>same = cClub</code> copies the reference; still one list	[Zoe, Ivy, Max]
17	<code>cClub.size() - 1</code> is 2; <code>same.remove(2)</code> removes "Max"	[Zoe, Ivy]

There is only one `ArrayList` object in this program. `cClub`, `roster`, and `same` are three variables that refer to it, so a change through any of them is seen through all of them.

Reread 4.8.8 *A List Holds References to Objects* and 4.8.5 *Inserting in the Middle* if your trace differed.

Question 18

Model answer: An `ArrayList` holds object references, so its type parameter must be a class. `int` is a primitive type, so `ArrayList<int>` does not compile; you write `ArrayList<Integer>` instead, and autoboxing and unboxing convert between `int` and `Integer` automatically. Writing the type parameter tells the compiler what every element must be, so it can reject a wrong type (such as adding a `String` to a list of `Integers`) at compile time. A plain `ArrayList` accepts anything, so the same mistake would not be caught until the program runs.

A complete answer says:

- The element type must be a class (a reference type), so use the wrapper class `Integer`.
- With `ArrayList<Integer>`, the compiler checks the types going in and coming out.
- Without a type parameter, type mistakes show up only at run time, which makes them harder to find.

Reread 4.8.7 *The Element Type and Wrapper Classes* if your answer was missing one of these.

Question 19

```
Echo
[Drift, Spark, Bloom, Echo]
Drift
4
```

Line	Call	songs afterward	Printed
50–51	two <code>addSong</code> calls	[Echo, Bloom]	
52	<code>playNext("Drift")</code> : insert at index 1	[Echo, Drift, Bloom]	

53	<code>skip()</code> : remove "Echo" from index 0, append it, return it	[Drift, Bloom, Echo]	Echo
54	<code>playNext("Spark")</code> : insert at index 1	[Drift, Spark, Bloom, Echo]	
55	<code>show()</code>	[Drift, Spark, Bloom, Echo]	the list
56	<code>nowPlaying()</code> is <code>get(0)</code>	[Drift, Spark, Bloom, Echo]	Drift
57	<code>getCount()</code> is <code>size()</code>	[Drift, Spark, Bloom, Echo]	4

The step that trips people up is line 54: after the skip, the song at index 0 is "Drift", so "Spark" goes right after it, pushing "Bloom" and "Echo" right.

Reread *Putting it together*, *4.8.5 Inserting in the Middle*, and *4.8.6 Removing an Element* if your output differed.