

Unit 7: ArrayList

An ArrayList is a list that grows and shrinks while a program runs. This unit covers creating ArrayLists, the core methods for adding, reading, changing, and removing elements, how to traverse a list safely, the standard algorithms rewritten for ArrayLists, linear search, selection and insertion sort, and the ethics of collecting data about people.

AP Computer Science A

ArrayList

About 15 minutes to read

How to use this guide

Read it in order the first time because the ideas stack. You need the method table from 7.2 before the traversal traps in 7.3 make sense, and you need traversal before the algorithms in 7.4 through 7.6. Most exam points in this unit come from tracing short code segments by hand, so work the examples with a pencil and cover the answers.

After the first read, drill the two things the exam returns to every year. Trace the removal-during-traversal example until the skipping behavior is automatic, and trace one full pass of each sort until you can do it without the book. Finish with the practice questions, then complete the recall check on the last page out loud.

What this unit is worth. Unit 7 is about 10 to 12.5 percent of the AP CSA exam. The removal-during-traversal trap is one of the most reliable exam questions in the entire course. Selection and insertion sort tracing appear almost every year, and ArrayList algorithms are FRQ 3 material.

7.1 Introduction to ArrayList

An **ArrayList** is a list whose size can change while the program runs. An array has a fixed length chosen at creation, and that length never changes. An ArrayList starts empty and adjusts its size as elements are added or removed. It lives in the `java.util` package, so any file that uses one starts with an import statement.

```
import java.util.ArrayList;

ArrayList<String> names = new ArrayList<String>();
names.add("Maya");
names.add("Lee");
```

The type in the angle brackets is called the **generic type**. It must be a class, never a primitive. You cannot write `ArrayList<int>`. Instead you use the **wrapper class** that matches the primitive, such as `Integer` for `int`. Java converts between the two automatically. Converting an `int` to an `Integer` when it goes into the list is called **autoboxing**, and converting back to `int` when it comes out is called **unboxing**. You do not write any conversion code yourself.

PRIMITIVE TYPE	WRAPPER CLASS TO USE
<code>int</code>	<code>Integer</code>
<code>double</code>	<code>Double</code>
<code>boolean</code>	<code>Boolean</code>
<code>char</code>	<code>Character</code>

Trap. `ArrayList<int>` is a compile error. The compiler rejects primitive types inside the angle brackets, so every list of numbers on the exam is declared as `ArrayList<Integer>` or `ArrayList<Double>`. This is one of the fastest points on the test if you remember it, and one of the fastest losses if you do not.

7.2 ArrayList Functions

Everything you do with an ArrayList goes through its methods. There are six to know cold. Notice that the list adjusts its size for you. You never manage capacity yourself.

METHOD	WHAT IT DOES	RETURNS
<code>add(value)</code>	Appends value to the end of the list	<code>true</code>
<code>add(index, value)</code>	Inserts value at index, shifting later elements one slot right	nothing
<code>get(index)</code>	Reads the element at index	the element
<code>set(index, value)</code>	Replaces the element at index with value	the OLD element that was replaced
<code>remove(index)</code>	Removes the element at index, shifting later elements one slot left	the REMOVED element
<code>size()</code>	Counts the elements currently in the list	an <code>int</code>

The return values of `set` and `remove` are the most tested row in that table. Both hand you back the element that was there before the change. `add(index, value)` inserts and grows the list by one. `set` replaces and leaves the size alone. Mixing those two up is a standard wrong answer.

```
ArrayList<Integer> nums = new ArrayList<Integer>();
nums.add(4);           // list is [4]
nums.add(9);           // list is [4, 9]
nums.add(0, 7);        // list is [7, 4, 9]
int x = nums.set(1, 8); // x is 4, list is [7, 8, 9]
int y = nums.remove(0); // y is 7, list is [8, 9]
int n = nums.size();    // n is 2
```

Trace this line by line. The two-argument `add` puts 7 at index 0 and pushes 4 and 9 right. `set(1, 8)` swaps out the 4 at index 1 and hands the 4 back, so `x` is 4, not 8. `remove(0)` takes the 7 off the front and hands it back, so `y` is 7. The list ends at `[8, 9]` with size 2.

Trap. `int x = nums.set(1, 8)` stores the element that was replaced, which is 4. Students who read too fast assign 8, the new value. Whenever a question captures the return of `set` or `remove`, write down the old element before you move on.

7.3 Traversing ArrayLists

Traversing a list means visiting each element in turn. The standard indexed loop uses `size()` as the bound, and the valid indexes run from 0 to `size() - 1`.

```
for (int i = 0; i < names.size(); i++) {
    System.out.println(names.get(i));
}
```

The bound is `i < names.size()`, never `i <= names.size()`. Index `size()` does not exist, so `<=` throws an `IndexOutOfBoundsException`. The **enhanced for loop** is the shorter alternative when you only need to read each element.

```
for (String name : names) {
    System.out.println(name);
}
```

Read that as "for each name in names." It is convenient, but it hides the index from you, and you cannot add to or remove from the list inside it. Any traversal that changes the list needs the indexed loop.

7.3 The Removal Trap

Removing an element during a forward traversal is the single most tested idea in this unit, so slow down here. When `remove(i)` runs, every element after index `i` shifts one slot left. The loop then increments `i`, which moves it past the element that just shifted into the removed slot. That element is never examined.

```
ArrayList<Integer> vals = new ArrayList<Integer>();
vals.add(3);
vals.add(6);
vals.add(8);
vals.add(5);
for (int i = 0; i < vals.size(); i++) {
    if (vals.get(i) % 2 == 0) {
        vals.remove(i);
    }
}
System.out.println(vals);    // prints [3, 8, 5]
```

Trace it by hand. The list starts as `[3, 6, 8, 5]`. At `i = 0` the 3 is odd, so nothing happens. At `i = 1` the 6 is even, so it is removed and the list becomes `[3, 8, 5]`. Now the 8 sits at index 1, but index 1 has already been visited. At `i = 2` the 5 is odd. The loop ends, and the 8 survives even though it is even. The output is `[3, 8, 5]`, not `[3, 5]`.

The reliable fix is to traverse backward. Removing an element then shifts only elements you have already visited, so nothing gets skipped.

```
for (int i = vals.size() - 1; i >= 0; i--) {
    if (vals.get(i) % 2 == 0) {
        vals.remove(i);
    }
}
System.out.println(vals);    // prints [3, 5]
```

Trace this one too. Start `[3, 6, 8, 5]`. At `i = 3` the 5 is odd. At `i = 2` the 8 is even, removed, and the list becomes `[3, 6, 5]`. At `i = 1` the 6 is even, removed, giving `[3, 5]`. At `i = 0` the 3 is odd. Both evens are gone. Another correct fix is to decrement `i` after each removal in the forward loop, but the backward loop is easier to get right.

Trap. When a question removes elements inside a forward loop, do not trust the obvious answer. The list with every matching element removed is exactly what the trap wants you to pick. Trace the indexes, watch which element shifts into the removed slot, and check whether that slot gets visited.

7.4 Developing Algorithms Using ArrayLists

Every array algorithm you already know has an ArrayList version, and the logic does not change. Only the syntax changes. Replace `arr[i]` with `list.get(i)` for reading, replace `arr[i] = v` with `list.set(i, v)` for writing, and replace `arr.length` with `list.size()`. Finding a maximum looks like this.

```
int max = scores.get(0);
for (int i = 1; i < scores.size(); i++) {
    if (scores.get(i) > max) {
        max = scores.get(i);
    }
}
```

Start the running value at the first element, then compare each remaining element against it. Starting at index 1 avoids a wasted comparison of the first element with itself. The same shape works for minimums, sums, counts, and averages. Swapping two elements needs a temporary variable because `set` overwrites.

```
int temp = list.get(a);
list.set(a, list.get(b));
list.set(b, temp);
```

Read it as three moves. Save element `a` in `temp`, copy element `b` into position `a`, then write `temp` into position `b`. Without `temp`, the first `set` destroys the value you still need. Counting elements that meet a condition follows the same traversal shape.

```
int count = 0;
for (int i = 0; i < words.size(); i++) {
    if (words.get(i).length() > 5) {
        count++;
    }
}
```

ARRAY VERSION	ARRAYLIST VERSION
<code>arr.length</code>	<code>list.size()</code>
Read <code>arr[i]</code>	<code>list.get(i)</code>
Write <code>arr[i] = v</code>	<code>list.set(i, v)</code>
No equivalent	<code>list.add(v)</code> and <code>list.remove(i)</code> change the size

Trap. The only genuinely new logic in Unit 7 algorithms is `add` and `remove`. Everything else is array logic wearing `get` and `set`. If an algorithm question feels unfamiliar, translate it back into array syntax in your head and solve it there.

7.5 Searching

Linear search checks each element in order until it finds the target. It returns the index of the first match. When the target is not in the list, it returns -1. That -1 is a convention, not a real index, and the calling code is expected to check for it before using the result with `get`.

```
public static int linearSearch(ArrayList<String> list, String target) {
    for (int i = 0; i < list.size(); i++) {
        if (list.get(i).equals(target)) {
            return i;
        }
    }
    return -1;
}
```

Two details matter here. First, objects are compared with `.equals`, never `==`. The `==` operator asks whether two references point to the same object in memory, which is not the same as having the same text. Two distinct `String` objects can hold identical letters and still fail `==`. Second, the method returns from inside the loop the moment it finds a match, so a list with the target twice reports the first position.

Trap. Exam questions test what -1 means. It means "not found," and feeding it to `get` throws an exception. If a question asks what happens after a failed search, check whether the code guards the -1 before using it as an index.

7.6 Sorting

The exam tests two sorts, **selection sort** and **insertion sort**, and it tests them by asking you to trace. You do not need to invent them, but you need to follow them one pass at a time.

Selection sort

Selection sort builds the sorted list from left to right. Each pass scans the unsorted part, finds the smallest element, and swaps it into the next position. After pass 1, index 0 holds its final value. After pass 2, index 1 is final. The last element needs no pass, so the outer loop runs `size() - 1` times.

```
for (int i = 0; i < list.size() - 1; i++) {
    int minIndex = i;
    for (int j = i + 1; j < list.size(); j++) {
        if (list.get(j) < list.get(minIndex)) {
            minIndex = j;
        }
    }
    int temp = list.get(i);
    list.set(i, list.get(minIndex));
    list.set(minIndex, temp);
}
```

Trace pass 1 on [7, 3, 5, 2]. `minIndex` starts at 0. At `j = 1`, 3 is less than 7, so `minIndex` becomes 1. At `j = 2`, 5 is not less than 3. At `j = 3`, 2 is less than 3, so `minIndex` becomes 3. Swap index 0 with index 3,

and the list is [2, 3, 5, 7]. One question this unit loves: after the first pass only, the answer is [2, 3, 5, 7], not the fully sorted list.

Insertion sort

Insertion sort grows a sorted section from the left. Each pass takes the next element and shifts larger elements one slot right until the element reaches its spot. Unlike selection sort, which swaps once per pass, insertion sort shifts a run of elements and then places the value.

```
for (int i = 1; i < list.size(); i++) {  
    int value = list.get(i);  
    int j = i - 1;  
    while (j >= 0 && list.get(j) > value) {  
        list.set(j + 1, list.get(j));  
        j--;  
    }  
    list.set(j + 1, value);  
}
```

Trace all three passes on [4, 1, 3, 2]. Pass 1 (i = 1): value is 1. At j = 0, 4 is greater than 1, so shift it right to get [4, 4, 3, 2], then j drops to -1 and the loop ends. Place 1 at index 0: [1, 4, 3, 2]. Pass 2 (i = 2): value is 3. At j = 1, 4 is greater than 3, so shift to get [1, 4, 4, 2]. At j = 0, 1 is not greater than 3, so stop and place 3 at index 1: [1, 3, 4, 2]. Pass 3 (i = 3): value is 2. At j = 2, 4 shifts right: [1, 3, 4, 4]. At j = 1, 3 shifts right: [1, 3, 3, 4]. At j = 0, 1 is not greater than 2, so stop and place 2 at index 1: [1, 2, 3, 4].

WHAT TO WATCH	SELECTION SORT	INSERTION SORT
Work per pass	Scan for the minimum, then one swap	Shift larger elements right, then one placement
After one pass on [7, 3, 5, 2]	[2, 3, 5, 7]	[3, 7, 5, 2]
Outer loop bound	i < size() - 1	i < size(), starting at 1

Trap. "After one pass" questions punish students who finish the sort in their heads. Trace exactly the passes the question names and stop. For insertion sort, the other trap is treating the shifts as swaps. Shifting overwrites the slot to the right; the saved value is placed only once, at the end.

7.7 Ethical Issues Around Data Collection

This topic is light on the exam, usually one question. Programs collect data about people: names, locations, purchase histories, browsing behavior, and more. That collection raises ethical questions. Users often do not realize how much is gathered, the data can be used in ways they never agreed to, it can be shared with or sold to third parties, and it can be stolen in a security breach. A responsible programmer thinks about what data is collected, who can see it, how long it is kept, and whether the person it describes understands and consents. On the test, expect a scenario and a question asking you to identify the ethical concern or the responsible practice. You are not asked to recite definitions.

Confusions That Cost Points

PAIR	HOW TO KEEP THEM STRAIGHT
<code>size()</code> vs <code>length</code> vs <code>length()</code>	ArrayLists use <code>size()</code> . Arrays use <code>length</code> with no parentheses. Strings use <code>length()</code> . Giving one type another type's version is a compile error.
<code>add(index, value)</code> vs <code>set(index, value)</code>	<code>add</code> inserts and grows the list by one. <code>set</code> replaces an element and leaves the size alone. Swapping them shifts the list when you meant to replace, or destroys a value you meant to keep.
<code>set</code> and <code>remove</code> return values	Both return the OLD element. <code>int x = list.set(1, 8)</code> stores the element that was replaced, not 8.
Forward removal vs backward removal	Removing inside a forward loop skips the element that shifts into the removed slot. Loop backward so removals only shift elements you have already visited.
<code>ArrayList<Integer></code> vs <code>ArrayList<int></code>	Generics require wrapper classes. <code>ArrayList<int></code> does not compile.
<code>==</code> vs <code>.equals</code>	<code>==</code> compares references for objects. Use <code>.equals</code> to compare the contents of objects like strings.
Valid indexes	Indexes run 0 to <code>size() - 1</code> . Using <code>size()</code> itself, or a <code>-1</code> from a failed search, throws an <code>IndexOutOfBoundsException</code> .

Practice Questions

Original questions written for this guide in the style of the AP exam. Answers and explanations are on the next page, so complete the questions before checking them.

1. Consider the following code segment.

```
ArrayList<Integer> list = new ArrayList<Integer>();  
list.add(10);  
list.add(20);  
list.add(1, 15);  
list.set(0, 5);  
System.out.println(list);
```

What is printed as a result of executing the code segment?

1. [10, 15, 20]
2. [5, 15, 20]
3. [5, 20, 15]
4. [5, 10, 15, 20]

2. Consider the following code segment.

```
ArrayList<String> words = new ArrayList<String>();  
words.add("red");  
words.add("blue");  
words.add("green");  
String s = words.set(1, "yellow");  
words.remove(0);  
System.out.println(s + " " + words.size());
```

What is printed as a result of executing the code segment?

1. blue 2
2. yellow 3
3. yellow 2
4. red 2

3. Consider the following code segment.

```
ArrayList<Integer> nums = new ArrayList<Integer>();
nums.add(4);
nums.add(8);
nums.add(7);
nums.add(9);
for (int i = 0; i < nums.size(); i++) {
    if (nums.get(i) % 2 == 0) {
        nums.remove(i);
    }
}
System.out.println(nums);
```

What is printed as a result of executing the code segment?

1. [7, 9]
2. [8, 7, 9]
3. [4, 7, 9]
4. [7, 9, 4]

4. Consider the following method.

```
public static int search(ArrayList<String> list, String target) {
    for (int i = 0; i < list.size(); i++) {
        if (list.get(i).equals(target)) {
            return i;
        }
    }
    return -1;
}
```

If list contains ["cat", "dog", "bird"] and target is "fish", what value is returned?

1. 0
2. -1
3. 3
4. An exception is thrown

5. An `ArrayList<Integer>` `a` contains `[7, 3, 5, 2]`. After the first pass of the outer loop of selection sort (ascending), what are the contents of `a`?

1. `[2, 3, 5, 7]`
2. `[3, 7, 5, 2]`
3. `[7, 3, 5, 2]`
4. `[2, 3, 7, 5]`

6. An `ArrayList<Integer>` `b` contains `[4, 1, 3, 2]`. After the first two passes of the outer loop of insertion sort (ascending, `i = 1` and `i = 2`), what are the contents of `b`?

1. `[1, 3, 4, 2]`
2. `[1, 2, 4, 3]`
3. `[1, 4, 3, 2]`
4. `[1, 2, 3, 4]`

7. Consider the following code segment.

```
ArrayList<Integer> scores = new ArrayList<Integer>();
scores.add(70);
scores.add(85);
scores.add(90);
int total = 0;
for (int s : scores) {
    total += s;
}
System.out.println(total);
```

What is printed as a result of executing the code segment?

1. 245
2. 255
3. 70
4. 90

8. Which of the following declarations compiles without error?

1. `ArrayList<int> ages = new ArrayList<int>();`
2. `ArrayList<Integer> ages = new ArrayList<Integer>();`
3. `ArrayList<boolean> flags = new ArrayList<boolean>();`
4. `ArrayList<char> letters = new ArrayList<char>();`

Answer Key

- 1. B.** The two single-argument adds give `[10, 20]`. `add(1, 15)` inserts 15 at index 1, shifting 20 right, giving `[10, 15, 20]`. `set(0, 5)` replaces the element at index 0, giving `[5, 15, 20]`. A skips the final set call entirely. C ignores the index argument in `add` and appends 15 to the end instead of inserting it at index 1. D treats `set` as an insertion, but `set` replaces without changing the size; inserting at index 0 is `add`'s job.
- 2. A.** `set(1, "yellow")` replaces "blue" and returns the old element, so `s` is "blue". `remove(0)` takes "red" off the front, leaving `["yellow", "green"]` with size 2. The output is blue 2. B thinks `set` returns the new value and also forgets that `remove` shrank the list. C makes only the first of those two mistakes, capturing "yellow" as the return value. D confuses `set`'s return value ("blue") with `remove`'s return value ("red").
- 3. B.** The list starts `[4, 8, 7, 9]`. At `i = 0`, 4 is even and is removed, giving `[8, 7, 9]`. The 8 shifts into index 1, which has already been visited. At `i = 1`, 7 is odd. At `i = 2`, 9 is odd. The 8 is skipped, so the result is `[8, 7, 9]`. A is the answer you get if you ignore the skipping behavior and remove every even number. C keeps the 4 and removes the 8, which reverses the actual shift. D scrambles the order; removal never reorders the elements that remain.
- 4. B.** The loop checks "cat", "dog", and "bird" against "fish" and finds no match, so the method falls through to return -1, the standard signal for "not found." A confuses the first index with the not-found signal. C returns `list.size()`, which is not a valid index and is never what a search returns. D is wrong because nothing in the method can throw here; the loop guard `i < list.size()` keeps every access in bounds.
- 5. A.** On the first pass, the scan finds the minimum, 2, at index 3, and swaps it into index 0. The list becomes `[2, 3, 5, 7]`. B is what insertion sort produces after one pass (shifting 3 past 7), so it catches students who mix up the two sorts. C means the scan never updated `minIndex`, as if the inner loop did nothing. D swaps with the wrong element, as if the scan stopped before reaching the true minimum at the end.
- 6. A.** Pass 1 (`i = 1`): value 1 shifts past 4 and is placed at index 0, giving `[1, 4, 3, 2]`. Pass 2 (`i = 2`): value 3 shifts past 4 and is placed at index 1, giving `[1, 3, 4, 2]`. The question names exactly two passes, so stop there. B is what two passes of selection sort produce, so it catches students who trace the wrong algorithm. C stops after one pass. D is the fully sorted list, which needs the third pass.
- 7. A.** The enhanced for loop visits each element once, and autoboxing converts each `Integer` back to `int` for the addition. The sum is `70 + 85 + 90 = 245`. B is an arithmetic slip; the code adds exactly the three elements in the list. C would mean the loop ran only once or that `total` kept only the first element. D would mean `total` kept only the last element instead of accumulating.
- 8. B.** Generic type arguments must be reference types, and `Integer` is the wrapper class for `int`, so B compiles. A, C, and D all use primitive types (`int`, `boolean`, `char`) inside the angle brackets, which the compiler rejects. The matching wrappers would be `Integer`, `Boolean`, and `Character`.

When you check your answers, note which distinction each miss came from. Make a flashcard for that distinction and drill it spaced out over the next few days instead of rereading the whole section. If you missed one of these questions, the same distinction is worth practicing again in Rycal, where the ArrayList deck has flashcards for it and more practice questions use the same kinds of traps.

One-Page Recall Check

Say each answer out loud before you look back, and mark the ones you cannot finish. Anything you cannot say out loud yet belongs in your flashcard deck. In Rycal, add those items to the ArrayList deck and let spaced review bring them back over the next few days.

- Explain how an ArrayList differs from an array, and write the import statement.
- Declare an ArrayList<Integer> and explain why ArrayList<int> does not compile.
- Name the wrapper classes for int, double, boolean, and char.
- List the six core ArrayList methods and state what each one returns.
- Explain what set and remove hand back, and why it is the old element.
- Write a forward traversal using size() and get(), with the correct bound.
- Explain why removing elements during a forward traversal skips elements, and trace an example.
- Show the backward-loop fix for safe removal and explain why it works.
- Explain why the enhanced for loop cannot be used to remove elements.
- Rewrite an array max-finding algorithm for an ArrayList.
- Write the three-line swap using get and set, and explain the temporary variable.
- Write linear search, explain the -1 convention, and say why the comparison uses .equals.
- Trace one full pass of selection sort by hand on a four-element list.
- Trace one full pass of insertion sort by hand on a four-element list.
- Name one ethical concern about collecting personal data and one responsible practice.

Where to go next. Turn every missed item above into flashcards and drill them spaced out over several days rather than in one sitting. In Rycal, open the ArrayList deck under AP Computer Science A. The deck covers the terms in this guide, and its practice questions target the same traps named here. If you have a test date, add it in the Test Planner. You can also start your next review with a Brain Dump, then check what you missed against this guide.

Key terms for this unit

ArrayList, Dynamic size, Generic type, Wrapper class (Integer, Double, Boolean, Character), Autoboxing, Unboxing, add, get, set, remove, size, Traversal, Enhanced for loop, IndexOutOfBoundsException, Linear search, Selection sort, Insertion sort, Data collection ethics.

About this guide. Written for Rycal and aligned to the College Board AP Computer Science A course framework, Unit 7. All questions and explanations are original Rycal writing. Rycal is independent and is not affiliated with or endorsed by the College Board.