**R6.11: What is wrong with each of the following code segments?**

1. ArrayList<int> values = new ArrayList<int>();

Primitive types can’t be used with array lists. Instead, you must use the Integer wrapper class

1. ArrayList<Integer> values = new ArrayList();

On the right side of the statement, there is no type parameter after ArrayList.  
  
c. ArrayList<Integer> values = new ArrayList<Integer>;

This statement is missing the parentheses after the type at the end of the statement.  
  
d. ArrayList<Integer> values = new ArrayList<Integer>();  
  for (int i = 1; i <= 10; i++)  
  {  
     values.set(i - 1, i \* i);  
  }

The problem with this code is that you can’t use the set method with an array list if there are no elements in the array list. Because no elements were added to the array list, the set method can’t change any elements (need to use the add method).  
  
e. ArrayList<Integer> values;  
  
  for (int i = 1; i <= 10; i++)  
  {  
     values.add(i \* i);  
  }

In this code, the array list is never actually initialized. As a result, the values can’t be added to the array. The first line should look something like the following: ArrayList<Integer> values = new ArrayList<Integer>();

**R6.19 Consider the following loop for collecting all elements that match a condition; in this case, that the element is larger than 100.**  
    ArrayList<Double> matches = new ArrayList<Double>();  
    for (double element : values)  
    {  
       if (element > 100)  
       {  
          matches.add(element);  
       }  
    }

**Trace the flow of the loop, where values contains the elements 110 90 100 120 80. Show two columns, for element and matches.**

Element Matches

110 (>100) [110.0]

90 (not >100) [110.0]

100 (not >100) [110.0]

120 (>100) [110.0, 120.0]

80 (not > 100) [110.0, 120.0]

Based on this trace of the loop, matches will end up consisting of [110.0, 120.0].

**R6.31**[**Section 6.8.8**](https://drive.google.com/file/d/1YPFFDPmMgP84gnBhJCl1nUSoJeA98FM7/view?usp=sharing)**shows that you must be careful about updating the index value when you remove elements from an array list. Show how you can avoid this problem by traversing the array list backwards.**

Going off the example given in 6.8.8., the following code would solve the problem of skipping the check of certain elements:

for (int i = words.size()-1; i >= 0; i--)

{

String word = words.get(i);

if (word.length() < 4)

{

//Remove the element at index i.

words.remove(i);

}

}

Starting the for loop at the last element of the array list and decrementing by 1 each time until i is less than 0 will solve this issue. This works because if you work from the end of the array to the start, when you remove an element, only the indices of elements after it will be affected. Even after removing an element, there will still be the same number of elements before that element, so those indices will remain the same. For example, if you use the words “welcome”, “to”, “the”, and “island”, i will start at 3 and won’t remove “island”. Next, i will decrease to 2 and will remove “the”. Because “welcome” and “to” still remain, their indices are still 0 and 1. i will once again decrease to 1 and remove “to”. Finally, i will go down to 0, and check “welcome”.

**R6.33 How do you perform the following tasks with array lists in Java?**

1. Test that two array lists contain the same elements in the same order.

To test if two array lists have the same elements in the same order you could use the equals method. For example, if you have array list values1 and array list values2, you can do: values1.equals(values2);

This will return true if the array lists are the same and false if they are not.

b. Copy one array list to another.

To copy one array list to another, you just have to pass the reference of the original array list to the constructor of the new array list. If you have an array list called values1 and you want to make a copy called values 2, you would do:

ArrayList<Integer> values2 = new ArrayList<>(values1);  
  
c. Fill an array list with zeroes, overwriting all elements in it.

If you want to take an array list with elements currently in it and overwrite all elements to 0, you could use a for loop. For example, if you have an array list called values1, you could do the following:

for(int i = 0; i < val1.size(); i++){

val1.set(i, 0);

}  
  
d .Remove all elements from an array list.

There are several ways you could remove all elements from an array list, including using a backwards for loop. However, you can also use the removeAll and clear methods for array lists. These would look like this:

values1.removeAll(values1);

or

values1.clear();

**R6.34 True or false?**

1. All elements of an array list are of the same type.

True – all elements in an array list are the same type, but different array lists can take in elements of different types.  
  
b. Array list index values must be integers.

True  
  
c. Array lists cannot contain strings as elements.

False – can have something like the following:

ArrayList<String> words = new ArrayList<>();  
  
d. Array lists can change their size, getting larger or smaller.

True – that’s the main advantage they have over regular arrays.  
  
e. A method cannot return an array list.

False – array lists can be used like other variables – they can be returned and used as method parameters.  
  
f. A method cannot change the size of an array list argument.

False – there are several methods that can easily change the size of an array list.

**R7.6 What is the difference between throwing an exception and catching an exception?**Catching an exception allows you to prevent errors in your program using a catch clause. This is where you give the exception you are expecting and can print a message for when it happens. On the other hand, when you throwing exceptions, you are throwing an object of an exception class. A throw clause usually involves a condition in which an error should occur, followed by a declaration of a new exception, often with a message. Basically, throw statements construct new exception objects within a program. Typically, throwing exceptions occurs early on whereas catching exception happens later in the program. This is because any thrown exception must be caught at some point in the program.

**R7.7 What is a checked exception? What is an unchecked exception? Give an example for each. Which exceptions do you need to declare with the throws reserved word?**A checked exception is an exception that occurs while the program is compiling. Additionally, checked exceptions must be caught or declared in the same method where they happen. An example of a checked exception would be a FileNotFoundException in a program where you try to open a file at a specific location. On the other hand, unchecked exceptions happen at the time of execution – they are known as runtime exceptions. An example of this would be an InputMismatchException you would get if you entered a word when a scanner was looking for an int. Checked exceptions need to be declared with the reserved throws word.

**R7.8 Why don’t you need to declare that your method might throw an IndexOutOfBoundsException?**

You don’t need to declare that your method might throw an IndexOutOfBoundsException because this exception is an unchecked exception (runtime exception), and you do not need to declare that a method might throw an unchecked exception.  **R7.9 When your program executes a throw statement, which statement is executed next?**When your program executes a throw statement, the first line in the first catch block that matches the exception type will be executed first.

**R7.10 What happens if an exception does not have a matching catch clause?**

If an exception doesn’t have a matching clause, then the current method will be terminated and an exception will be thrown to the next highest level. If the program doesn’t have a matching catch clause at any point, then an error will occur and the program will end. In other words, if a try block doesn’t have a catch block, you will get a syntax error. Additionally, the catch block must directly follow the try block or else you will get more errors. **R7.11 What can your program do with the exception object that a catch clause receives?**

The program can use the exception object received by a catch clause to print some type of message telling the user what went wrong and how they should fix it. For example, if the user enters a word when they should’ve entered a number, you can print a message saying “Please enter a number.” Another useful thing that can be done in the catch clause is the program can print information about the specific error that occurred. This can be done using the name of the exception object followed by .printStackTrace();. **R7.12 Is the type of the exception object always the same as the type declared in the catch clause that catches it? If not, why not?**

No, the type of exception object is not always the same as the type declared in the catch clause. This is because you could have a more specific exception in a program, such as a FileNotFoundException. However, the program might not have a catch clause of type FileNotFoundException but instead might just have a catch clause of type Exception. Because type Exception is more general than FileNotFoundException, it will catch the FileNotFoundException despite not being the exact same type of exception. **R7.15 Which exceptions can the next and nextInt methods of the Scanner class throw? Are they checked exceptions or unchecked exceptions?**

The next method of the Scanner class can throw a NoSuchElementException (no more tokens are available) and an IllegalStateException (the scanner is closed). The nextInt method can throw an InputMismatchException (the next token doesn’t match the Integer regular expression, or is out of range), a NoSuchElementException (no more tokens are available), and an IllegalStateException (the scanner is closed). All three of these types of exceptions are unchecked exceptions.