# Array, ArrayList

#### Q1. Arrays

Which of the following commands will be used to declare and initialise an integer-type array, ‘random’, of size 5?

Ans: int random[ ] = new int[5];

**✓ Correct**

**Feedback:**

The format for declaring and initialising an array is ‘data\_type<array\_name>[] = new <data\_type>[data\_size]’. Here, the data type is int and the array’s name is ‘random’. So, option D is the correct statement. This is equivalent to creating a new array object random[ ], of the int class.

#### Q2. Array Declaration

Suppose you are initialising an array as:

int a[], b;

int[] c, d;

For such a variable declaration,

Ans: a, c, and d are arrays.

**✓ Correct**

**Feedback:**

If you mention [ ] with the data type only (for example, int[ ]), this means all the variables declared with it are arrays. Hence, both c and d are arrays. If you mention only the data type (for example, int), this means you can declare both the arrays and non-arrays together, by using the [ ] symbol with the arrays. Hence, a is also an array.

#### Q3. Accessing an Element in an Array

The following code segment stores a set of six numbers in an array named ‘arr’.

int arr[] = {2, 4, 5, 10, 42, 76};

Which of the following statements will help you access only the first element of this array?

Ans: arr[0]

**✓ Correct**

**Feedback:**

You can access any element in an array by using the arrayName[index] format. The index of the first element of an array is 0.

#### Q4. Multiple Data Types in an Array

Can you store both String and Integer in the same array?

Note: This is the class Integer, not the primitive data type int.

Ans: Yes, Integer and String can be stored in the same array.

**✓ Correct**

**Feedback:**

You can always create such an array of different data types by declaring the type of the array as ‘Object’. Then, you can store objects of different data types in this Object array, since all the objects in Java extend the ‘Object’ class. And through polymorphism, any object will also belong to the Object class, in addition to the class(es) that it belongs to. To know more about how to do this, click the link given below this question.

#### Q5. Arrays

Till now, you saw two different programs with the following code snippets:

I:

String studentList[] = {"Sujit", "Siddharth", "Karanpreet"};

II:

Student s1 = new Student("Sujit", 1);

Student s2 = new Student("Siddharth", 2);

Student s3 = new Student("Karanpreet", 3);

Student studentList[] = {s1, s2, s3};

Now, which of the following options are correct with respect to both the snippets given above?  
  
More than one options can be correct.

Ans: i)   
I implements the array of the String data type.

**✓Correct**You missed this!

**Feedback:**

This is an array with the data type of all the elements as String.

ii)   
II implements the array of the Student data type.

**✓Correct**You missed this!

**Feedback:**

This is an array with the data type of all the elements as Student.

#### Q6. Data Types

In both the arrays you saw so far, the data types of all the objects was either String or Student.

Which of the following options is true about both these data types?

Ans: Both the arrays store non-primitive (reference) data types.

**✓ Correct**

**Feedback:**

Primitive data types are basic data types that store primitive values. Reference data types are any instantiable class as well as arrays. So, arrays can be used to store reference (non-primitive) and primitive data type variables and objects.

#### Q7. Data Types of Arrays

In the code snippets given in the previous question, you created arrays of the **data type String** as well as of the **data type Student**. In the latter case, you declared the data type as **Student**, as shown in the code below.

I:   Arrays of the String data type

String studentList[] = {"Sujit", "Siddharth", "Karanpreet"};

II:  Arrays of the Student data type

Student s1 = new Student("Sujit", 1);

Student s2 = new Student("Siddharth", 2);

Student s3 = new Student("Karanpreet", 3);

Student studentList[] = {s1, s2, s3};

Now, what do you think would be stored in the second array of the **Student** data type?

Ans: An array of the Student data type would contain multiple objects of the Student class. For example, if you create three objects called s1, s2, and s3 of the Student class, they would be stored in three cells of that array. You cannot insert any object of any other data type (say, String or Professor) in this array of the Student class. Each cell of the array would contain values of the instance variables for any particular Student class object.

#### Q8. Structure of ArrayList

Which of the following options is correct about ArrayList, as implemented by Prof. Sujit in the code shown in the video?

ArrayListstudentList = new ArrayList();

studentList.add(new Student("Sujit", 1));

studentList.add(new Student("Siddharth", 2));

studentList.add(new Student("Karanpreet", 3));

Ans: ArrayList is a predefined class, structurally similar to an array, with three objects of the Student class stored in it.

**✓ Correct**

**Feedback:**

ArrayList is a predefined class that is imported from the java.util package. It is an independent class that can have the objects/elements of any data type (Student, in this case).

#### Q9. Printing the ArrayList Class

Explain the significance of line 1 in the code segment given below.

for(Object o : students) {

Student s = (Student)o; //line 1

System.out.println(s.getDetails());

}

Ans: Since the ArrayList class can contain the elements of any data type, you can refer to its type as ‘Object’. Whenever you create the ArrayList of the Object data type, you need to typecast its elements to access them or use them to call methods. Now, all the elements of the ‘students’ ArrayList are of the type Object. You cannot access the getDetails( ) method of the Student class from ‘o’ since it is of the type Object. So, the type of the object o has to be typecast to Student to access the getDetails( ) method. That’s why line 1 is written, to typecast o into s, which is of the Student type. Then, you can access the method using s.getDetails( ).

#### Q10. Type Safety

A type-safe program is one that doesn’t (or can’t) encounter a run-time error.

So, how can you make your program type-safe?

Ans: By converting a potential run-time error into a compile time error

**✓ Correct**

**Feedback:**

Compile time errors are always preferred over run-time errors because they are easier to identify and fix. So, you can make your program type-safe by converting potential run-time errors into compile time errors.

#### Q11.Type Safety

What do you think is the actual reason behind the run-time error that occurred in the above video?

Ans All the elements of ArrayList cannot be typecast into the Student type to call the getDetails( ) method.

**✓ Correct**

**Feedback:**

You had to typecast since you wanted to access all the elements of ArrayList to call the getDetails( ) method. Now, new elements of the String data type cannot be typecast into the Student type because the elements are of different classes and the String and Student classes do not have an ‘is-a’ relationship (i.e. polymorphism is not applicable here). So, there’s a run-time error.

#### Q12. ArrayList

So, you created an ArrayList with two elements, and you want to print both of them. Can you identify the issue in the code segment given below?

public static void main(String[] args) {

ArrayList mix = new ArrayList();

mix.add("Hari");

mix.add(2);

printMixList(mix);

}

public static void printMixList(ArrayList elements) {

for(Object o : elements) {

System.out.println(o);

}

}

Ans: There’s no issue in the code. It will run perfectly well.

**✓ Correct**

**Feedback:**

In this code, you’re creating ArrayList with two elements, a String type and an Integer type. Then, you’re directly printing these elements by assigning them to a variable ‘0’ of the ‘Object’ type. Typecasting is not required here, so the code should work perfectly well.

#### Q13. Making a Program Type-Safe

When you add elements of different data types to an ArrayList, you may face run-time errors because of typecasting issues.

How will you convert these potential run-time errors into compile-time errors and make the program type-safe?

Ans: You can make the program type-safe by changing the way you declare the ArrayList. You can declare it using the ArrayList format, by which the compiler will ensure that objects of any other data type cannot be added to this special ArrayList.

#### Q14. ArrayList Using Generics

Which of the following declarations would be used to create an ArrayList that accepts only integer values?

Ans: ArrayList<Integer>myList = new ArrayList<Integer>() ;

**✓ Correct**

**Feedback:**

ArrayList do not accept parameters of primitive data types such as int, float, or char. They only accept parameters of the reference type, such as Integer or String. Integer is a predefined class in Java and is different from the primitive data type int. But all the objects of the Integer class have the same behaviour as that of an int-type variable.

#### Q15. Adding an Element to ArrayList

What will the output of the following code segment be?

**Note**: get() returns the element at the specified index in the given list. Refer to [this](https://beginnersbook.com/2013/12/java-arraylist-get-method-example/) link to learn more about this method.

ArrayList<String> shape = new ArrayList<String>();

shape.add("Square");

shape.add("Triangle");

shape.add(2, "Circle");

shape.add(1, "Rhombus");

System.out.println(shape.get(2));

Ans: triangle

# Summary

Here’s a quick summary of the topics you learnt about in this session:

1. **Array**: An array is a collection of elements of the same data type, stored in a single variable (declared with the [ ] symbol). Arrays are simple to create and enable easy access to elements stored at different positions using index values. But they have fixed lengths and can store only fixed sets of elements.
2. **ArrayList**: A Java ArrayList is very similar to an array, with the facility to add or remove elements dynamically during run-time. That’s why **ArrayList is also referred to as a dynamic array**. It is used to store a group of elements, where storing duplicates is allowed. Elements are stored in contiguous positions in an ArrayList and you can easily access, add or remove elements from an ArrayList.

**LinkedList**

A LinkedList is another data structure in Java that works like ArrayList, with similar methods and functionalities.

However, there is a difference in the structure. The elements are not stored in contiguous memory locations. This affects programs from a performance perspective, i.e. even though the outputs of all the operations are the same, the speeds are different.

For the time being, let’s focus on just the operations of both ArrayList and LinkedList.

All the three methods—add, remove, and contains—work the same way in a LinkedList. Now, let’s look at another method to access an element from a list using its index.

# Lists and Polymorphism

In reality, **List** is an interface that is implemented by the **ArrayList** and **LinkedList** classes. This is the reason why you can instantiate both 'ArrayList' and 'LinkedList' by declaring the type of variable as List.

Refer to the diagram below to understand how all these classes and interfaces are linked to a larger interface named **Collection**.

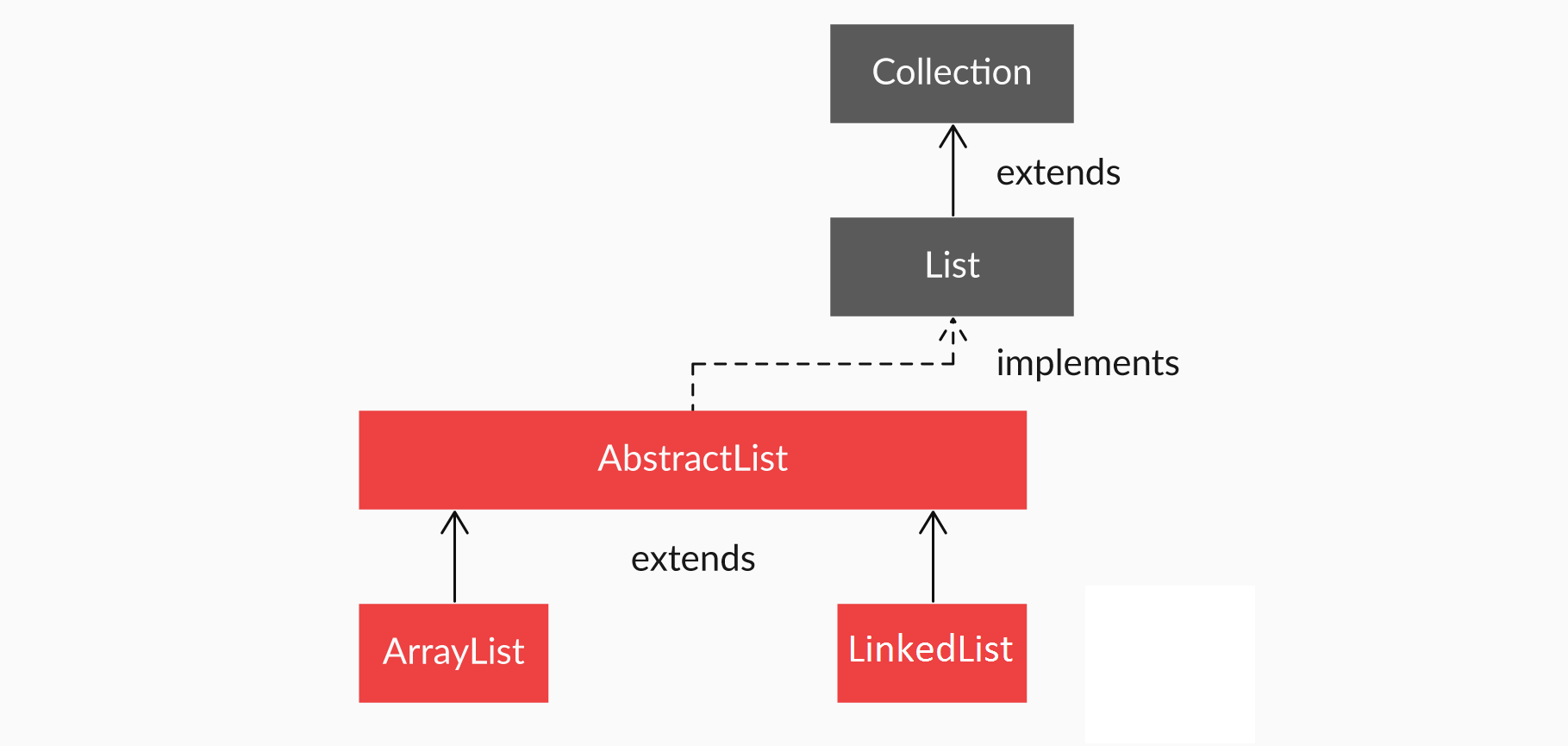


Figure 1 - Hierarchy of classes and interfaces under the Collection Interface

The diagram above denotes the following steps:

1. The List interface extends the Collection interface, or List is the child interface of Collection.
2. AbstractList implements the List interface, which is further extended by the ArrayList and LinkedList classes. Or, the ArrayList and LinkedList classes are implementations of the List interface.
3. The AbstractList class is extended by the ArrayList and LinkedList classes. Or in other words, ArrayList and LinkedList are the subclasses of the Abstract class.

#### Q14: Implementing ArrayList Using the List Interface

ArrayList needs to be implemented using the List interface. The code segment for this is given below. Can you find the mistake in the code?

public static void main(String[] args) {

List<Student> studentList = new List<Student>();

studentList.add(new Student("Sujit", 1));

studentList.add(new Student("Siddharth", 2));

studentList.add(new Student("Karanpreet", 3));

studentList.add(new Student("Hari", 5));

printStudentList(studentList);

}

public static void printStudentList(List<Student> students) {

for(Student s : students) {

System.out.println(s.getDetails());

}

}

Ans: List cannot be instantiated. So, write ‘new ArrayList’ as the constructor of List.

**✓ Correct**

**Feedback:**

Interfaces cannot be instantiated by themselves. They require classes for their implementations.

# ListIterator - I

ListIterator is a subinterface of the Iterator interface, which is used to iterate over a list. It has a lot more features than the Iterator interface. These features are given below:

1. It is used to traverse a list in any direction, i.e. forward or backward, very easily.
2. It does not point to any current element; its cursor position always lies between the previous and next elements (Figure 1 given below).
3. It has methods to find out if there exists a next or previous element and to find the value of this next or previous element, as well as its indices.

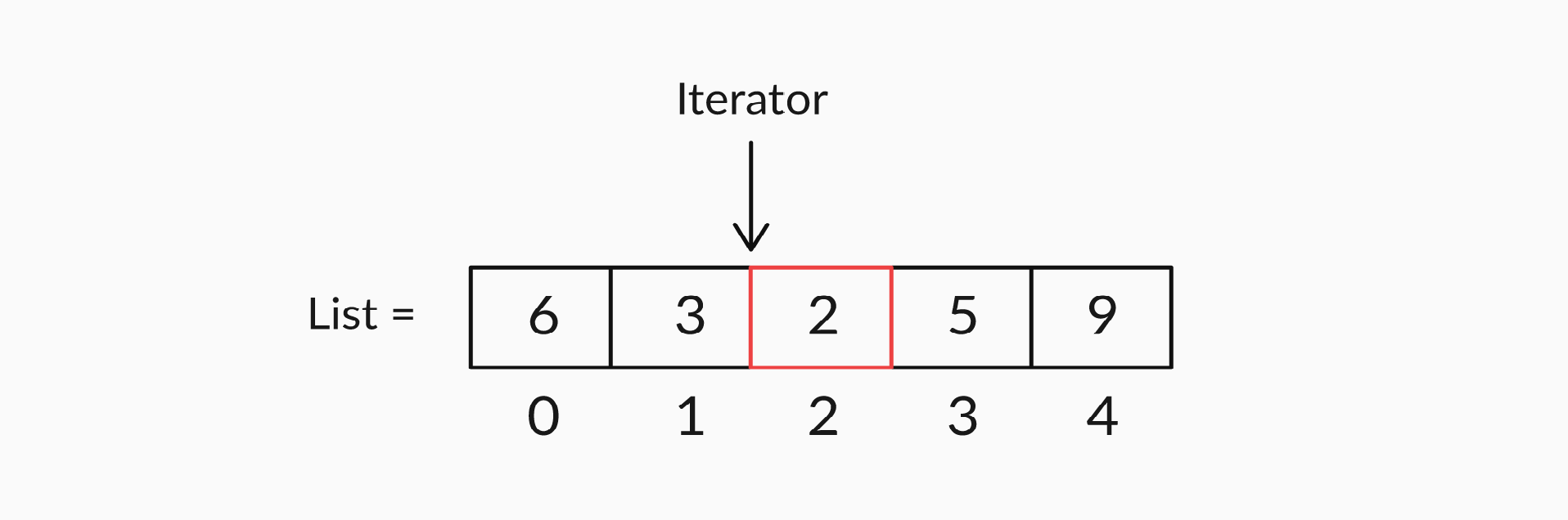


Figure 1 - Representation of a List Iterator

You now know how to traverse forward using an iterator. Now, answer the following questions about iterators to test your skills on forward iteration. Before we move to the next topic, let’s look at the real benefit of using iterators, i.e. the ability to move in any direction.

#### Q15: Forward Iteration Using ListIterator

Suppose you have a list of integers that looks like this: {1,2,3}. What will the output of the following code snippet be if this list is passed to the iterate method shown below?

public void iterate (List<Integer> numberList) {

ListIterator<Integer> it = numberList.listIterator();

System.out.println(it.next());

}

Ans: 1

**✓ Correct**

**Feedback:**

When you call the listIterator method, its cursor is positioned at the start of the list, i.e. before the first element. So, the ‘it.next( )’ method will return the value of the element at index 0.

#### Q16: Forward Iteration Using ListIterator

Again, suppose you have a list of integers that looks like this: {1,2,3}. What will the output of the following code snippet be if this list is passed to the iterate method shown below?

**public** **void** **iterate** (List<Integer> numberList) {

ListIterator<Integer> it = numberList.listIterator();

it.next();

it.next();

System.out.println(it.next());

}

Ans: 3

**✓ Correct**

**Feedback:**

When you call the listIterator method, its cursor is positioned at the start of the list, i.e. before the first element. So, the ‘it.next( )’ method will return the value of the element at index 0 and move the iterator to the start of the next element. And after writing ‘it.next( )’ twice, the iterator will point to the start of the third element. Then, running ‘it.next( )’ will print this element.

#### Q17: Forward Iteration Using ListIterator

Once more, you have a list of integers that looks like this: {1,2,3}. What will the output of the following code snippet be if this list is passed to the iterate method shown below?

public void iterate (List<Integer> numberList) {

ListIterator<Integer> it = numberList.listIterator();

it.next();

it.next();

it.next();

System.out.println(it.next());

}

Ans: NoSuchElementException

**✓ Correct**

**Feedback:**

The iterator stops when it reaches the end of the list, i.e. when the cursor is positioned after the last element. Writing ‘it.next( )’ three times in a list of three elements will take the cursor to the end of the list, and printing ‘it.next( )’ will return NoSuchElementException.

#### Q18: Forward Iteration Using ListIterator

Once again, you have a list of integers: {1,2,3}. What will the output of the following code snippet be if this list is passed to the iterate method shown below?

public void iterate (List<Integer> numberList) {

ListIterator<Integer> it = numberList.listIterator(2);

System.out.println(it.next());

}

Ans: 3

**✓ Correct**

**Feedback:**

When you call the listIterator method without any parameter, its cursor is positioned at the start of the list. But when you call the method with a parameter, say, ‘n’, with the syntax ‘listIterator(n)’, its cursor moves to the start of index ‘n’, i.e. ‘it.next( )’ returns the element at index ‘n’. So, calling ‘listIterator(2)’ will return the element at index 2.

#### Q19: Backward Iteration Using ListIterator

Where would the cursor point after calling the listIterator method in the following manner?

ListIterator<Student> it = students.listIterator(students.size());

Ans: After the last element of the list

**✓ Correct**

**Feedback:**

The ‘students.size()’ argument will return the number of elements in the list. When you call the method with a parameter ‘n’, with the syntax ‘listIterator(n)’, its cursor moves to the start of index ‘n’. So, after calling ‘listIterator(students.size())’, the cursor will move to the start of the index number denoted by ‘students.size()’. As the index of the last element of the list would be ‘students.size() - 1’, the cursor would be after the last element.

# ListIterator - II

**These are fixed output coding question, hence your test cases would not be evaluated and would not be shown as  Accepted or Rejected when you click Submit. They would be shown as Submitted.**

**Your program will only be verified based on one sample test case when you click Verify.**

# Summary

Here’s a quick summary of the topics you learnt about in this session:

1. **LinkedList**: A LinkedList is another data structure in Java that works like ArrayList, with similar methods and functionalities. However, there is a difference in the structure. The elements are not stored in contiguous memory locations.
2. **List**: A Listis an interface that is implemented by the ArrayList and LinkedList classes. The List interface extends the Collection interface, or List is the child interface of Collection. AbstractList implements the List interface, which is further extended by the ArrayList and LinkedList classes. Or, the ArrayList and LinkedList classes are implementations of the List interface.
3. **ListIterator:**Iterator is an interface in Java, which is used to iterate over a collection of objects. You can think of an iterator as a cursor to an element in a collection. You can also use the iterator (or the cursor) to move to the next element in the collection.

ListIterator is a subinterface of the Iterator interface, which is used to iterate over a list. It has a lot more features than the Iterator interface, like traversing the list in any direction etc.

#### Q 20: Operations on the List Interface

Suppose you have a list of integers that looks like this: {1,2,3,4}. What will the output be if this list is passed to the process method below?

**public** **void** **process**(List<Integer> numberList){

numberList.add(**2**, **5**);

System.out.println(numberList.get(**3**));

}

Ans: 3

**✓ Correct**

**Feedback:**

With ‘numberList.add(2, 5)’, 5 will be added at index 2 of this list. So, the modified list will be {1, 2, 5, 3, 4}. Then, ‘numberList.get(3)’ will return the element at index 3, i.e. 3.

Q21 : Again, suppose you have a list of integers that looks like this: {1,2,3,4}. What will the output be if this list is passed to the process method below?

**public** **void** **process**(List<Integer> numberList){

numberList.add(**2**, **5**);

numberList.remove(**1**);

System.out.println(numberList.get(**3**));

}

Ans: 4

**✓ Correct**

**Feedback:**

With ‘numberList.add(2, 5)’, 5 will be added at index 2 of this list. Then, the element at index 1 will be removed from the list.

#### Q 22: Using Iterators on the List Interface

Once more, suppose you have a list of integers that looks like this: {1,2,3,4}. What will the output be if this list is passed to the iterate method below?

**public** **void** **iterate** (List<Integer> numberList) {

ListIterator<Integer> it = numberList.listIterator(**1**);

it.next();

it.next();

it.previous();

System.out.println(it.next());

}

Ans: 3

**✓ Correct**

**Feedback:**

The ‘listIterator(1)’ code moves the cursor to the start of index 1. The ‘it.next( )’ method will return the value of the next element and move the iterator to the start of this element. So, after writing ‘it.next’ twice, the iterator will point to the start of the third element, i.e. 4. However, the ‘it.previous( )’ method will return the value of the element previous to the position of the iterator and move the iterator behind by one position, i.e. before 3. Now, printing ‘it.next( )’ will output the next element, i.e. 3.

Q 23: Now, suppose you have a list of integers that looks like this: {1,2,3,4,5,6}. What will the output be if this list is passed to the iterate method below?

**public** **static** **void** **iterate**(List<Integer> num) {

ListIterator<Integer> it = num.listIterator(num.size());

**while**(it.hasPrevious()) {

**if**(it.hasPrevious()) {

System.out.println(it.previous());

}

it.previous();

}

}

Ans: 6, 4, 2

**✓ Correct**

**Feedback:**

‘num.listIterator(num.size())’ will move the iterator to the end of the list. For every element previous to the pointer in the loop, the code checks if a previous element exists, and then, it prints that element and moves the pointer to the left side. Then, ‘it.previous()’ moves the pointer to the left by one more position (without printing it). This continues for all the iterations of the loop, until the pointer reaches the start of the list. Therefore, alternative elements get printed backwards, starting from the last element.

#### Q24: ArrayList with Iterators

What would be the output of the following code snippet?

List<Integer> arrayList = **new** ArrayList<Integer>();

arrayList.add(**1**);

arrayList.add(**2**);

ListIterator<Integer> listIterator = arrayList.listIterator();

**while**(listIterator.hasNext()) {

System.out.print(listIterator.next());

listIterator.previous();

}

Ans: Will print 1 infinite times

**✓ Correct**

**Feedback:**

Printing listIterator.next( ) will print the next element and move the iterator position to right by one element. So this print ‘1’ and move iterator before ‘2’. listIterator will move the iterator position to left by one position, i.e. before ‘1’. So in each iteration of the loop, 1 will be printed and iterator remains at same position. So this will become an infinite loop since you’ll never reach the last element of the ArrayList. Therefore, “while(listIterator.hasNext())” will always return true, and hence you’ll be stuck in an infinite loop.