

Open in app ↗

Medium

 Search

★ Get unlimited access to the best of Medium for less than \$1/week. [Become a member](#)



JAVA Collection Cheat Sheet

satish kathiriya · [Follow](#)

9 min read · Aug 22, 2021



Listen



Share

... More

Important methods:

`ArrayList<String> list = new ArrayList<>();``ArrayList(class)` — implements `→List(Interface)` — extends `→ Collection``ArrayList` is **not Synchronized**, Its equivalent synchronized class is **Vector**.

Creation

```
List list = new ArrayList(); // Empty Constructor
List list = new ArrayList(50); // Initial capacity
List list = new ArrayList(oldList); // add collection
```

An `ArrayList` can also be created using an existing `Collection`. The newly created `ArrayList` will contain all the elements in the same order in the original collection.

Methods

e = element (Object), i = index , c = Collection

- `list.size();`
- `list.contains(e);` //returns bool
- `list.indexOf(e);`
- `list.lastIndexOf(e);`

- **list.get(i);**
- **list.set(i,e);**
- **list.add(e);** // Add element
- **list.add(i,e);**
- **list.addAll(c)** // Add Collections
- **list.addAll(i, c)**
- **list.remove(i);**//returns bool : return at index

eg : `arr.remove(3)` -> this will remove element at index 3

- **list.remove(e)** or **list.remove(object);**

//returns bool : This is tricky in case of integer list,pass Integer object

NOTE: It will only remove first occurrence of the object

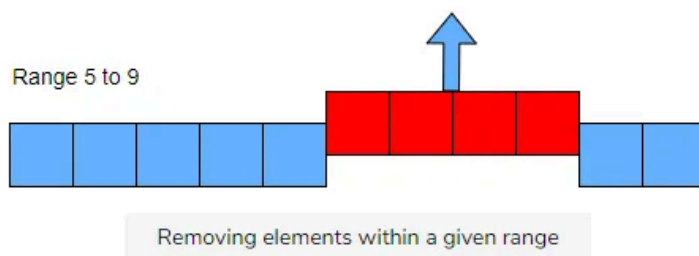
eg : `arr.remove(new Integer(3))` -> this will remove first occurrence of object with value 3

We saw that `remove(int index)` removes a method at the given index and `remove(Object o)` removes the given object from the **ArrayList**. Suppose we have an **ArrayList** that contains five elements i.e [13, 21, 43, 2, 9]. Now, if we do `list.remove(2)`, then which overloaded method will be called. Will `remove(int index)` be called or `remove(Object o)` be called? `remove(int index)` will be called because we are passing a primitive to remove method. If we want to delete element 2, we should call `remove(new Integer(2))` because elements are stored in an **ArrayList** as objects and not primitives.

- **list.removeRange(int fromIndex, int toIndex)**

Removing all the elements within a range

Let's suppose we need to remove all the elements from index 5 to 9. This can be done using the `removeRange(int fromIndex, int toIndex)` method. This method will remove, from this list, all of the elements whose index is between *fromIndex*, inclusive, and *toIndex*, exclusive. Please note that this method is not defined in the List class. So, it can be used only when the reference type is also **ArrayList** and not List.



Remove Range

- `list.removeAll(Collection<?> c)`

We can use the `removeAll(Collection<?> c)` method to remove, from the given list, all of the elements that are contained in the specified collection.

- `list.toArray();` // convert list to array
- `Collections.sort(list);` // sorts in Ascending Order
- `Collections.sort(list, Collections.reverseOrder());` // Descending order

The `Collections` class contains a `sort(List<T> list)` method, which is used to sort an `ArrayList`. This method takes an `ArrayList` as input and sorts it in ascending order.

In the `sort(List<T> list)` method, *T* represents the type of object that is stored in the `ArrayList`. The `Collections.sort(List<T> t)` method takes an `ArrayList` of type *T* objects as the input. It is a must that *T* should implement the Comparable interface; otherwise, the code will not compile.

- `List<Integer> sortedList = list.stream().sorted().collect(Collectors.toList());` // USING STREAM
- `List<Integer> sortedList = list.stream().sorted(Comparator.reverseOrder())`
`.collect(Collectors.toList());`

The ArrayList can be sorted in reverse order using streams bypassing `Comparator.reverseOrder()` to the `sorted()` method.

- `list.clear()` //remove all element from the list
- `list.replaceAll(UnaryOperator<E> operator)`

```
list.replaceAll((element) -> element.toUpperCase());
```

QUICK FACTS:

- List indexes start from '0', just like array index.
- List allows "null"
- List supports Generics and we should use it whenever possible. Using Generics with List will avoid `ClassCastException` at runtime.

copyOnWriteArrayList — thread-safe version of list

```
List<String> list = new CopyOnWriteArrayList<>();
```

HashSet<Integer> set= new HashSet<>();

- `set.add(e);`

*If the element is not already present, then this method puts the element and returns `true` .
If the element is already present, then it returns `false` .*

- `set.contains(e);`
- `set.remove(e);`
- `set.clear();` // remove all elements in set

- `set.isEmpty();`

SORTING : NOT POSSIBLE IN HASHSET

*Since a **HashSet** stores the elements in random order, it is not possible to store the elements in a **HashSet** in sorted order. If we want to sort the elements of a **HashSet**, then we should convert it into some other Collection such as a **List**, **TreeSet**, or **LinkedHashSet**.*

```
/ Creating an ArrayList from existing set.  
List<Integer> list = new ArrayList<>(set);  
  
// Sorting the list.  
Collections.sort(list);
```

***HashSet** does not allow duplicate elements.*

***HashSet** allows only one null element.*

*The elements are inserted in random order in a **HashSet**.*

*A **HashSet** is internally backed by a **HashMap**.*

initial capacity of 16 and a load factor of 0.75

TreeSet<Integer> set= new TreeSet<>();

```
TreeSet<Integer> set = new TreeSet<>();  
TreeSet<Integer> reverseSet =  
    new TreeSet<(Comparator.reverseOrder());  
// with comparator as arguments  
TreeSet<Integer> set = new TreeSet<>(list); // pass collection
```

Since all the elements are stored in sorted order in a **TreeSet**, storing elements should either implement the **Comparable** interface or a custom **Comparator** while creating the **TreeSet**.

- `set.add(e)`
- `set.addAll(Collection c);`
- `set.first()` // Fetching the first element in TreeSet
- `set.last();` // Fetching the last element in TreeSet
- `set.headSet(40)` // Fetching all the elements less than 40 | 40 is not inclusive.
- `set.tailSet(40)` // Fetching all the elements greater than 40 | 40 is not inclusive.
- `set.remove(e);`
- `set.isEmpty();`
- `set.size();`
- `set.contains(e);`

TreeSet does not allow duplicate elements.

TreeSet class doesn't allow null elements.

Since elements are stored in a tree, the access and retrieval times are quite fast in a TreeSet.

The elements are stored in ascending order in a TreeSet.

Difference between a HashSet and TreeSet

1. The HashSet allows one null element, whereas a TreeSet does not allow a null element.
2. The elements are stored in random order in a HashSet, whereas it is stored in sorted order in TreeSet.
3. HashSet is faster than TreeSet for the operations like add, remove, contains, size, etc.

`HashMap<String, Integer> map= new HashMap<>();`

k = key, v = value

- `map.put(k,v);`
- `map.putIfAbsent(k,v);`
- `map.putAll(anotherMap);`
- `map.get(k)` //returns v mapped to k
- `map.getDefault(key , defaultValue);`

`map.put(key, map.getDefault(key,0)+1);` // very common to maintain frequency of the elements. . .

- `map.remove(k)` //returns v and removes it.
- `map.size();`
- `map.containsKey(k);`
- `map.containsValue(v);`

```
Map<String, Integer> stockPrice = new HashMap<>();
stockPrice.put("Oracle", 56);
stockPrice.put("Fiserv", 117);
stockPrice.put("BMW", 73);
stockPrice.put("Microsoft", 213);
```

```
System.out.println(stockPrice.containsKey("Oracle"));
```

```
System.out.println(stockPrice.containsValue(73));
```

Looping through a HashMap:

- `map.entrySet()`

```
for(Map.Entry<String, Integer> element : map.entrySet()){
    element.getKey(); // use getKey() for key
    element.getValue(); //use getValue() for values.
}
```

- **map.keySet()** // This method returns a **Set** containing all the keys present in the Map.
- **map.values()** // This method returns a **Collection** containing all the values present in the Map.

```
Set<String> keys = stockPrice.keySet();
for(String key : keys) {
    System.out.println(key);
}

Collection<Integer> values = stockPrice.values();
for(Integer value : values) {
    System.out.println(value);
}
```

Replace :

- **map.replace(K key, V oldValue, V newValue);**

// replace value of the key k if value of the key is same as old value

- **map.replace(K key,V newValue);** // replace value of the key with new value
- **map.replaceAll(bifunction);**

```
stockPrice.replaceAll((k,v) -> v + 10);
```

Using the `replace(K key, V oldValue, V newValue)` method#

The `replace(K key, V oldValue, V newValue)` method takes three parameters: the key, the old value, and a new value. It checks if the current value of the key is equal to the `oldValue` provided in the parameter. If yes then it replaces the value with `newValue` and returns `true`; otherwise, it returns `false`.

Using the `replace(K key, V value)` method#

This method takes only two parameters: a key and a value. It replaces the value of the key with the new value provided as a parameter and returns the old value. If the key is not present, then it returns `null`.

Using the `replaceAll(BiFunction<? super K, ? super V, ? extends V> function)` method#

This method takes a `BiFunction` as input and replaces the values of all the keys with the result of the given function. Suppose we need to add ten to the stock price of each company. Instead of updating the value for each stock one by one, we can use this method. The lambda expression to do this task will look like this:

```
(key, value) -> value + 10
```

The keys should be unique.

HashMap allows only one `null` key.

The values can be null or duplicate.

The keys are stored in random order.

an initial capacity of 16 and load factor of 0.75

TreeMap: `TreeMap` is a class in the `java.util` package that stores the keys in sorted order. Some of the features of `TreeMap` are:

`TreeMap<String, Integer> treeMap = new TreeMap<>();`

**`TreeMap<String, Integer> reverseMap = new TreeMap<>(
(Comparator.reverseOrder()));`**

```
// Creating a TreeMap using existing HashMap. This will store the  
elements in ascending order.
```

```
TreeMap<String, Integer> treeMap1 = new TreeMap<>(hashMap);
```

```
// Creating a TreeMap using existing TreeMap. This will store the  
elements in the same order as it was in the passed Map.
```

```
TreeMap<String, Integer> treeMap2 = new TreeMap<>(reverseMap);
```

- `put(k,v);`

- `putAll(map);`
- `get(k);`
- `remove(k)`
- `firstKey()` // gives smallest key (since **TreeMap** stores elements in sorted order)
- `firstEntry()` // give smallest Entry
- `lastKey()` // give largest key
- `lastEntry()` // give largest entry
- `replace(key,value);`
- `replace(key,oldValue,newValue);`

*The entries in **TreeMap** are sorted in the natural ordering of its keys.*

It does not allow null keys, however there can be null values.

*The **TreeMap** is not thread-safe, although it can be made thread-safe using the `synchronizedMap()` method of the **Collections** class.*

*Since a **TreeMap** stores the keys in sorted order, the objects that we are storing in the **TreeMap** should either implement the **Comparable** interface or we should pass a **Comparator** while creating the **TreeMap** object.*

PriorityQueue<Integer> pq = new PriorityQueue<>()

PriorityQueue<Integer> minHeap = new PriorityQueue<Integer>();

PriorityQueue<Integer> maxHeap = new PriorityQueue<Integer>((a,b) -> b-a);
//(max elements on top/descending order)

Let's say you want your own comparator, example, the pq has keys of a hashmap as it's elements, and they need to be arranged according to their values in a hashmap.

pq = new PriorityQueue<>((a,b) -> map.get(a)-map.get(b));

You can define your own comparator separately(named MyComparator for example):

```
pq = new PriorityQueue<>(new MyComparator());
```

String s = "hello";

- s.toCharArray();
- s.indexOf(substring);
- s.charAt(i);
- s.toUpperCase();
- s1.equals(s2); //Note to self: DON'T DO "==" AGAIN AND WONDER WHY THE ANSWER IS CONSTANTLY INCORRECT
- s1.equalsIgnoreCase(s2);
- s1.compareTo(s2); //returns s1-s2, in dictionary order so a-b returns -1
- s1.contains(s2);
- s.length();
- s.startsWith("hell"); //s.endsWith("ello"), returns boolean
- s.substring(incl,excl); // incl: inclusive index, included, excl: exclusive index, excluded
- str_array = s.split(" "); //return string array separated by spaces. "Hello world" returns ["Hello","world"]

Iterator itr = array_name.iterator(); //or list.iterator()

- itr.hasNext(); //returns bool
- itr.next();

- `itr.remove();` //removes curr element

Random r = new Random();

- `r.nextInt(n);` //return a random int from 0 to n

OR

- `Math.random();` //returns a random double between 0.0 and 1.0

Stack<Integer> s = new Stack<>();

- `s.push();`
- `s.peek();`
- `s.pop();`
- `s.empty();` //returns bool
- `s.size();`

Queue<Integer> q = new ArrayList<>();

- `q.add(e);`
- `q.remove();` //throws exception if empty
- `q.poll();` //same as remove but returns null if empty
- `q.peek();`
- `q.size();`

LINKED LIST :

```
//CREATTION

List<Integer> list = new LinkedList<Integer>();
List<Integer> list = new LinkedList<Integer>(oldList);

//ADD ELEMENTS
addLast(E e)
addFirst(E e)
add(int index, E element)

addAll(Collection c)
addAll(int index, Collection c)

//FETCHING
getFirst()
getLast()
get(int index)

//REMOVING
removeFirst()
removeLast()
remove(int index)
remove(E element) // first occurrence is removed.
removeLastOccurrence(E element)

//SORT
Collections.sort(linkedList);
```

Other Useful Methods:

- `Character.isDigit(c);` //returns bool
- `Character.isAlphabetic(c);`
- `Integer.parseInt(str);` //if str = "1234", it returns integer 1234.
- `Float.parseFloat(str);` //same as above for floats
- `Integer.toBinaryString(num);` //returns binary representation of num, input = 10 returns "1010"
- `Arrays.asList(arr);` //converts array to list
- `Integer.MIN_VALUE` //returns least possible int in Java
- `object.hashCode();` //returns hashcode value for object

- `IntegerList.get(i).intValue();` // to convert Integer to int;

Comparator:

- `Arrays.sort(arr, (a,b)->b-a);` //sorts in descending order
- Defining MySort to use in other sorting methods (example sort arrays based on their first element)

```
public class MySort implements Comparator<int[]>{
    public int compare(int[] a, int[] b){
        return a[0]-b[0];
    }
}
```

And then using it like:

- `Arrays.sort(arr, MySort)`
- For ArrayLists, `Collections.sort(arraylist, MyArrListSort());`

Comparable VS Comparator

the `Collections.sort()` method sorts the given List in ascending order. But the question is, how does the `sort()` method decide which element is smaller and which one is larger?

*Each wrapper class(Integer, Double, or Long), String class, and Date class implements an interface called **Comparable**. This interface contains a `compareTo(T o)` method which is used by sorting methods to sort the Collection. This method returns a negative integer, zero, or a positive integer if **this** object is less than, equal to, or greater than the object passed as an argument.*

```
class Employee implements Comparable<Employee> {
    String name;
    int age;

    public Employee(String name, int age) {
        super();
        this.name = name;
        this.age = age;
    }

    @Override
    public int compareTo(Employee emp) {
```

```

        return (this.age - emp.age);
    }
    //We will sort the employee based on age in ascending order returns
    a negative integer, zero, or a positive integer as this employee age
    is less than, equal to, or greater than the specified object.
}

public class Vehicle implements Comparable<Vehicle> {
    String brand;
    Integer makeYear;
    public Vehicle(String brand, Integer makeYear) {
        super();
        this.brand = brand;
        this.makeYear = makeYear;
    }

    @Override
    public int compareTo(Vehicle o) {
        //Using the compareTo() method of String class.
        return this.brand.compareTo(o.brand);
    }
}

```

If we use the `Collections.sort(List<T> list)` method to sort an **ArrayList**, then the class whose objects are stored in the **ArrayList** must implement the **Comparable** interface. If the **ArrayList** stores an **Integer**, a **Long**, or a **String**, then we don't need to worry as these classes already implement the **Comparable** interface. But if the **ArrayList** stores a custom class object, then that class must implement the **Comparable** interface.

If we need some flexibility in sorting, we should use the **Comparator** interface instead of the **Comparable** interface. The **Comparator** interface has a method, `compare(T o1, T o2)`, which takes two objects, `o1` and `o2` as parameters. It returns `-1` if `o1 < o2`, `1` if `o1 > o2` and `0` if `o1` is equal to `o2`.

If we need to use the **Comparator** interface, then we can't use the `Collections.sort(List<T> t)` method as `T` should implement the **Comparable** interface. There is another overloaded method, `sort(List<T> list, Comparator<? super T> c)`, that takes the list as well as a **Comparator** object as input. It then sorts the List on the basis of logic, which is provided in the **Comparator** implementation.

```
import java.util.Comparator;

public class BrandComparator implements Comparator<Vehicle> {

    @Override
    public int compare(Vehicle o1, Vehicle o2) {
        return o1.brand.compareTo(o2.brand);
    }
}
```

Problems to practice:

- [Best Team with no conflicts\(Leetcode\)](#).
- [Largest Number\(Leetcode\)](#).
- [Queue Reconstruction By Height\(Leetcode\)](#).

Java

Cheatsheet

Hashmap

Arrays

Methods

[Follow](#)

Written by satish kathiriya

174 Followers · 435 Following

SDE — Amazon Robotics

Responses (5)



What are your thoughts?

Respond



Nishant Bhandari

Jul 14, 2024



thanks i have intetview tommorrow and i revised

qucikly



[Reply](#)



Syed Atamish Ali

Feb 10, 2024



Beautiful article.



[Reply](#)



ABHIJEET K BEHERA

Mar 3, 2022



Great summary to refer on the need basis. Thanks for sharing




[Reply](#)

See all responses

More from satish kathiriya



 satish kathiriya

Setting up Python workspace: Data science


setup python using python distribution. we are going to use Anaconda.

Mar 6, 2020  50



Binary Search

#SEARCHING #ALGORITHM

 satish kathiriya

Search in Rotated Sorted Array: Binary Search

Finding elements in the array is a very easy problem, we can use linear search to find elements in $O(N)$ time complexity.

Mar 11, 2021 🖱 123



satish kathiriya

Subarrays/SubString vs Subsequence vs Subsets

SUBARRAY :

Oct 4, 2022 🖱 239



- ✅ A stream **is** a pipeline of functions that can be evaluated.
- ✅ Streams **can** transform data.
- ❌ A stream **is not** a data structure.
- ❌ Streams **cannot** mutate data.

Intermediate operations

- Always return streams.
- Lazily executed.

Common examples include:

Function	Preserves count	Preserves type	Preserves order
<i>map</i>	✅	❌	✅
<i>filter</i>	❌	✅	✅
<i>distinct</i>	❌	✅	✅
<i>sorted</i>	✅	✅	❌

Get the unique surnames in uppercase of the first 15 book authors that are 50 years old or over.

```
library.stream()
    .map(book -> book.getAuthor())
    .filter(author -> author.getAge() >= 50)
    .distinct()
    .limit(15)
    .map(Author::getSurname)
    .map(String::toUpperCase)
    .collect(toList());
```

Compute the sum of ages of all female authors younger than 25.

```
library.stream()
    .map(Book::getAuthor)
    .filter(a -> a.getGender() == Gender.FEMALE)
    .map(Author::getAge)
    .filter(age -> age < 25)
    .reduce(0, Integer::sum);
```

Terminal operations

- Return concrete types or produce a side effect.
- Eagerly executed.

Common examples include:

Function	Output	When to use
reduce	concrete type	to cumulate elements
collect	list, map or set	to group elements

Parallel streams use the common ForkJoinPool for threading.

```
library.parallelStream()...
```

or intermediate operation:

```
IntStream.range(1, 10).parallel()...
```

Useful operations

Grouping:

```
library.stream().collect(
    groupingBy(Book::getGenre));
```

Stream ranges:

```
IntStream.range(0, 20)...
```

Infinite streams:

```
IntStream.iterate(0, e -> e + 1)...
```

Max/Min:

```
IntStream.range(1, 10).max();
```

FlatMap:

```
twitterList.stream()
    .map(member -> member.getFollowers())
    .flatMap(followers -> followers.stream())
    .collect(toList());
```

Pitfalls

- ❌ Don't update shared mutable variables i.e.


```
List<Book> myList =
    new ArrayList<>();
    library.stream().forEach(
        e -> myList.add(e));
```



satish kathiriya

Java Stream API CheatSheet:

Java stream API has the following important methods:

Sep 7, 2021  449  1[See all from satish kathiriya](#)

Recommended from Medium



In Coding Odyssey by Shivam Srivastava

Barclays Java Developer Interview

Senior Java Developer Interview Experience

Feb 1  69  1

Map VS



Sanjay Singh

Understanding map() vs flatMap() in Java 8 Streams: A Simple Guide

Overview



Sep 20, 2024



204



Lists



General Coding Knowledge

20 stories · 1902 saves



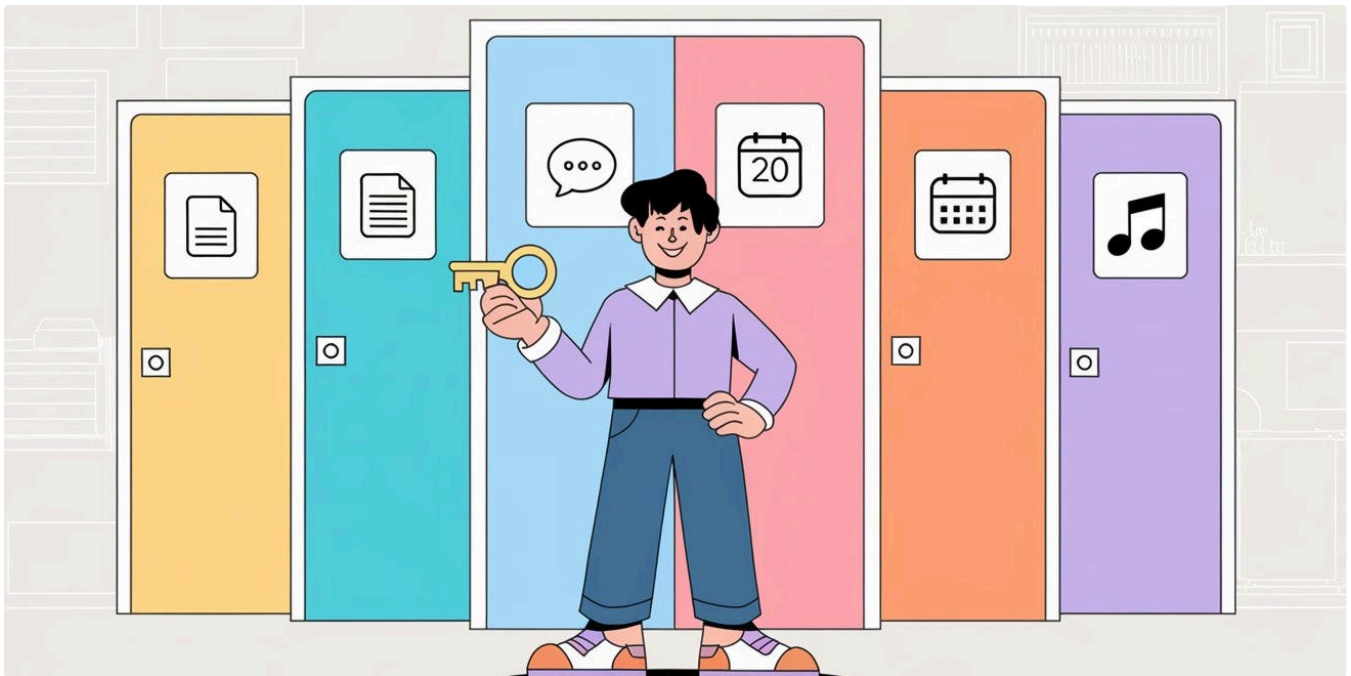
data science and AI

40 stories · 326 saves



Natural Language Processing

1924 stories · 1577 saves



 Rabinarayan Patra

Say Goodbye to Password Overload: Understanding Single Sign-On (SSO)

Tired of juggling multiple passwords? Discover how Single Sign-On (SSO) simplifies your login experience and boosts security in our...

✦ Oct 11, 2024 🖱 6

🔖
...



 Full Stack Developer

@RequestParam vs @QueryParam vs @PathParam vs @PathVariable in REST

Pretty confusing if you don't know this simple fact.

★ Sep 3, 2024 🖱 25




TOP 20 JAVA CODING PROGRAM FOR INTERVIEW

 Java Interview

Top 20 Java Coding Interview Programs

In this tutorial, we are going to discuss a few popular Coding Interview Questions. Practicing the problems provided in this article will...

★ Oct 20, 2024 🖱 7

 Praveen Tripathi

Java Collection Framework Questions You Can't Miss for Interviews: Part 1

“Preparing for a Java interview? Mastering the Java Collection Framework is crucial, as it forms the backbone of effective data...”

★ Nov 23, 2024 🖱 23



See more recommendations