**1 Why List allow duplicate but not set**

List in java allow duplicate element because it is designed as. List in java is simply a dynamic array whose size grows with data. Array is a collection of data and in array you can have duplicate value , that's why list allow duplicate element.

Set stores only unique objects, it does’t allow duplicate objects to be stored. It stores objects based on hashcode. Every object in java has unique id called hashcode which is a 32-bit signed integer allocated to an object by JVM.

Set interface is designed in such a way that it doesn’t accept duplicate elements into it. Internally, it invokes the .equals() method of the Object class (This is different from the overridden method of the String class). This method checks if the new object’s hash code matches with that of any of the previous entries in the Set and not the values (Unlike the overridden method of the String class). If not, it’s all cool and goes ahead and adds it to the Set.

Set is used for specific purpose where you want to avoid data redundancy.

Because life sometimes calls for things to happen multiple times.

Say you are processing temperature measurements over time, across 300 sensors, which are then appended to a file.

The chance that at least two readings will be the same, at least once over the lifetime of the monitor, is effectively 100%. An ordered Set (basically a list without duplicates) would screw up your readings like this:

Use Case Of Set- suppose you have dataset of employee with duplicate values. But you require only unique of them. Then you just simply need to iterate through the data and add it to set. At last you will have unique data of employees.

what happens internally when you pass duplicate elements in the add() method of the Set object , It will return false and do not add to the HashSet , as the element is already present .

How does a set interface in Java allow duplicate objects to be added but not duplicate elements?

The question can be put in a better way - “How does the Set interface allow objects with the same data?”

every time an object is created, it is assigned a unique identifier - a hash code. This is evident by printing the object’s reference on the console. I create multiple objects with the same data initialized on all of them, and JVM assigns unique hash codes to all of them. Consider the following snippet.

Animal a = new Animal("Dog");

Animal b = new Animal("Dog");

Animal c = new Animal("Dog");

From above, the three objects carry the same data in them but are identified by the unique hash codes JVM provides them. So, at a lower level, it is a bit ambiguous to consider them duplicates of each other.

We can add a, b, and c to the set as they’re different objects and are not duplicates of each other. With String objects and other primitive data types, the values are checked and so, duplicate values aren’t allowed in.

int x = 5;

int y = 5;

x and y cannot find their way into a Set unless they carry different values.

Define “duplicate objects” vs “duplicate elements”. They mean the same thing to a Set.

If the objects contain the same values but don’t return true when their equals() method is called to compare them, they are not considered duplicate by the contract defined for the Set interface.

If two objects equal each other via equals(), you cannot add them twice to the Set.

List<String> dupnumbers = new ArrayList<>();

dupnumbers.add("one");

dupnumbers.add("two");

dupnumbers.add("three");

dupnumbers.add("one");

System.err.println(dupnumbers);// [one, two, three, one]

List<String> disnumbers = new ArrayList<>(new HashSet<>(dupnumbers)); // hash set to remove duplicate

System.err.println(disnumbers);//[one, two, three]

List<String> streamnumbers = dupnumbers.stream().distinct().collect(Collectors.toList()); // duplciate remove using stram

System.err.println(streamnumbers);//[one, two, three]

**2 Difference between LIST and SET**

List and Set both are interfaces. They both extends Collection interface. The important differences between set and list are:

|  |  |  |
| --- | --- | --- |
| category | List | Set |
| Duplicate Objects | Allow | Don’t allow |
| Order | it maintains the insertion order, display the elements in the same order in which they got inserted into the list | unordered collection, it doesn’t maintain any order, There are few implementations of Set which maintains the order such as LinkedHashSet (It maintains the elements in insertion order) |
| Null elements | Allow multiple | Single Null |
| Access | provides positional access | doesn’t provides positional access |
| Impl Class | ArrayList, LinkedList, Vector and Stack | HashSet and LinkedHashSet interfaces. |

3 Set adding duplicate object will give error or not?

No error or exception simple it will return false and not gonna add duplicate value

Set<String> set = new HashSet<>();

set.add("sachin"); true

set.add("sachin"); false

set.add("darshan"); true

Size:2 no duplicate

**4 Hasp map duplicate key will give error or not?**

It will not give any error.

Internally If the map previously contained a mapping for the key, the old value is replaced by the specified value. A map m is said to contain a mapping for a key k if and only if [m.containsKey(k)](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html" \l "containsKey-java.lang.Object-) would return true if not return false.

If key key is new put method returns null telling its new key associated to map ,it it returns 1 then key is already associated with map

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

System.err.println(map.put("first", 1));// null

System.err.println(map.put("first", 1));// 1

System.err.println(map.put("first", 3));// 1

**5 how to achieve 100% encapsulation or Fully encapsulated class?**

By making all the data members of the class private. Now can use setter and getter methods to set and get the data in it. The Java Bean class is the example of a fully encapsulated class.

public class Customer {

private int id;

private String name;

public int getId() {

return id;

}

public void setId(int id) {

this.id = id;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

}

**6 Scenario for arrayList , LinkedList ,HashSet , LinkedHashSet, hashmap and LinkedHashMap ?**

ArrayList provides constant time for search operation,use ArrayList if searching is more frequent operation than add and remove operation. The LinkedList provides constant time for add and remove operations.use LinkedList for manipulation.

LinkedList is fast for adding and deleting elements, but slow to access a specific element. ArrayList is fast for accessing a specific element but can be slow to add to either end, and especially slow to delete in the middle.

understand the scenario and data set. If no insertions and removals are then good to go with ArrayList. Insertion order maintained in both

ArrayList use case:

This is the best case to use the ArrayList to load the records. Every application must be having this kind of use case.

For Example, the Amazon website has many customer's records in its database. If they want to retrieve the customers from the database and show it on a web screen. In this case, ArrayList will be used to add customer records from ResultSet.

Let us take a scenario where application A is using a third party API to get the credit card transactions. That API will returns all transactions for the recent month for a given credit card number.

Hashset use case:

When there should be no duplicate in in the data should go for HashSet

create one HashSet of Student records where each Student record contains three fields – name, rollNo and department. In these, rollNo will be unique for all students.

LisnkedHashset use case:

A LinkedHashSet is an ordered version of [HashSet](http://quiz.geeksforgeeks.org/hashset-in-java/" \o "quiz.geeksforgeeks.org" \t "https://www.quora.com/_blank) that maintains a doubly-linked List across all elements. When the iteration order is needed to be maintained this class is used. When iterating through a [HashSet](http://quiz.geeksforgeeks.org/hashset-in-java/" \o "quiz.geeksforgeeks.org" \t "https://www.quora.com/_blank) the order is unpredictable, while a LinkedHashSet lets us iterate through the elements in the order in which they were inserted. When cycling through LinkedHashSet using an iterator, the elements will be returned in the order in which they were inserted.

LinkedHashSet<String> hs = new LinkedHashSet<String>();

Maps are used for when you want to associate a key with a value and Lists are an ordered collection. HashMap are efficient for locating a value based on a key and inserting and deleting values based on a key.

While the HashMap will be slower at first and take more memory, it will be faster for large values of n. The reason the ArrayList has O(n) performance is that every item must be checked for every insertion to make sure it is not already in the list.

HashMap is faster/ than HashSet because values are associated with a unique key. HashSet is slower than HashMap because the member object is used for calculating hashcode value, which can be same for two objects.

HashMap should be used over ArrayList when there is no use of index and when the data is need to be identified using key value pair.

LinkedHashMap maintain insertion order of keys, i.e the order in which keys are inserted into LinkedHashMap. On the other hand HashMap doesn't maintain any order or keys or values. In terms of Performance there is not much difference between HashMap and LinkedHashMap but yes LinkedHashMap has more memory foot print than HashMap to maintain doubly linked list which it uses to keep track of insertion order of keys.

A HashMap has a better performance than a LinkedHashMap because a LinkedHashMap needs the expense of maintaining the linked list. The LinkedHashMap implements a normal hashtable, but with the added benefit of the keys of the hashtable being stored as a doubly-linked list. Both of their methods are not synchronized.

Shopping Cart is a real life example, where we see cart number against Item we have chosen in order we selected the item. So map could be LinkedHashMap<Cart Number Vs Item Chosen>

In most cases when using a Map you don't care whether the order of insertion is maintained. Use a HashMap if you don't care, and a LinkedHashMap is you care.

**7 why class should be declared final for immutable class?**

If class is not declared final a subclass can be created of that parent class and its functionality can be overridden. Which can easily modify existing value with latest value which makes it mutable class. Can use get and set method in sub class to change it value

class Immutable {

private final int value;

public Immutable(int value) {

this.value = value;

}

public int getValue() {

return value;

}

}

public class Mutable extends Immutable {

private int realValue;

public Mutable(int value) {

super(value);

realValue = value;

}

@Override

public int getValue() {

return realValue;

}

public void setValue(int newValue) {

realValue = newValue;

}

public static void main(String[] arg) {

Mutable obj = new Mutable(4);

Immutable immObj = (Immutable) obj;

System.out.println(immObj.getValue());

obj.setValue(8);

System.out.println(immObj.getValue());

}

}

**8 why hashmap or map interface is not included in collection framework**

It's part of the collection framework but it doesn't implement the java.util.Collection interface.However, it is a structure for keeping collections of data.

The Collection interface is implemented by (is the root of) List-like Collections while Map is implemented by(is the root of) the KEY-VALUE-like collections.

Because Map is three collections: Keys, values and key-value pairs.

"The collection framework" and Collection interface are not the same thing.

Maps work with key/value pairs, while the other collections work with just values. Map maps keys to values. It allows its content to be viewed as a set of keys, a collection of values and a set of key-value mappings

**9 deep copy,Shallow and Lazy copy**

In oops copying is creating a copy of an existing object, the resulting object is called an object copy or simply copy of the original object.There are several ways to copy an object, most commonly by a [copy constructor](https://www.geeksforgeeks.org/copy-constructor-in-java/) or [cloning](https://www.geeksforgeeks.org/clone-method-in-java-2/). We can define Cloning as “create a copy of object” Shallow, deep and lazy copy is related to cloning process

**Shallow Copy**

1 Whenever we use default implementation of clone method we get shallow copy of object means it creates new instance and copies all the field of object to that new instance and returns it as object type, we need to explicitly cast it back to our original object. This is shallow copy of the object.

2 clone() method of the object class support shallow copy of the object. If the object contains primitive as well as nonprimitive or reference type variable in shallow copy, the cloned object also refers to the same object to which the original object refers as only the object references gets copied and not the referred objects themselves.

3 If only primitive type fields or Immutable objects are there then there is no difference between shallow and deep copy in Java.

**Deep Copy**

1 Whenever we need own copy not to use default implementation we call it as deep copy,

So for deep copy we need to ensure all the member class also implement the Cloneable interface and override the clone() method of the object class.

A deep copy means actually creating a new object and copying over the values.

**when to use what**  
There is no hard and fast rule defined for selecting between shallow copy and deep copy but normally we should keep in mind that if an object has only primitive fields, then obviously we should go for shallow copy, but if the object has references to other objects, then based on the requirement, shallow copy or deep copy should be done. If the references are not updated then there is no point to initiate a deep copy.

**Lazy Copy**  
A lazy copy can be defined as a combination of both shallow copy and deep copy. The mechanism follows a simple approach – at the initial state, shallow copy approach is used. A counter is also used to keep a track on how many objects share the data. When the program wants to modify the original object, it checks whether the object is shared or not. If the object is shared, then the deep copy mechanism is initiated.

**Summary**  
In shallow copy, only fields of primitive data type are copied while the objects references are not copied. Deep copy involves the copy of primitive data type as well as object references.

public class ShallowCopy {

private int[] data;

public ShallowCopy(int[] values) {

data = values;

}

public void showData() {

System.out.println(Arrays.toString(data));

}

}

public class DeepCopy {

private int[] data;

public DeepCopy(int[] values) {

data = new int[values.length];

for (int i = 0; i < data.length; i++) {

data[i] = values[i];

}

}

public void showData() {

System.out.println(Arrays.toString(data));

}

}

public class TestCopyClass {

public static void main(String[] args) {

int[] vals1 = { 3, 7, 9 };

int[] vals2 = { 3, 7, 9 };

ShallowCopy sh = new ShallowCopy(vals1);

sh.showData();

vals1[0] = 13; // ShallowCopy copying ref change orignale ref val same effectes in copy

sh.showData();

DeepCopy dp = new DeepCopy(vals2);

dp.showData();

vals2[0] = 13; // deepcopy it creates a new object for the copies the values if change in

// orignal value dnt effect copiied object

dp.showData();// recommend if its non primitive type and muttable object

}

}

**10 when 2 object are equals then their hash code must be equal ?**

When 2 objects are equal their hash code has to be same. If two hash codes are equal then it cant say those object are same . two different objects can have same hash code but but can’t be said its same.

**11 if super class throwing exception does sub class need to handle vice versa?**

1. If SuperClass does not declare an exception, then the SubClass can only declare unchecked exceptions, but not the checked exceptions(compile time error).
2. If SuperClass declares an exception, then the SubClass can only declare the child exceptions of the exception declared by the SuperClass, but not any other exception.
3. If SuperClass declares an exception, then the SubClass can declare without exception.

**Summary in Simple**

If Super Class method throw exception then sub class method will be having option to throw only the same exception or his child exception class or no exception at all. It sub class cnt throw any parent level exception if parent it self has its child class excpetion leads to compiler time exception.

//case 1 If SuperClass doesn’t declare any exception and subclass declare checked exception

public class ExceptionOveridenRelation1 {

void method() {

System.out.println("SuperClass");

}

}

//SuperClass inherited by the SubClass

class SubClass extends ExceptionOveridenRelation1 {

// method() declaring Checked Exception IOException

void method() throws IOException {

// IOException is of type Checked Exception

// so the compiler will give Error

System.out.println("SubClass");

}

}

//case 2 If SuperClass doesn’t declare any exception and SubClass declare Unchecked exception

public class ExceptionOveridenRelation2 {

void method() {

System.out.println("SuperClass");

}

}

//SuperClass inherited by the SubClass

class SubClass2 extends ExceptionOveridenRelation2 {

// method() declaring unChecked Exception ArithmeticException

void method() throws ArithmeticException {

// ArithmeticException is of type unChecked Exception no compiler Error

System.out.println("SubClass");

}

}

//case 3.1 If SuperClass declares an exception and SubClass declares exceptions other than the child exception of the SuperClass declared Exception

public class ExceptionOveridenRelation3 {

void method() throws RuntimeException {

System.out.println("SuperClass");

}

}

//SuperClass inherited by the SubClass

class SubClass3 extends ExceptionOveridenRelation3 {

// SubClass declaring an exception

// which are not a child exception of RuntimeException

void method() throws Exception {

// Exception is not a child exception

// of the RuntimeException

// So the compiler will give an error

System.out.println("SubClass");

}

}

//case 3.2 If SuperClass declares an exception and SubClass declares an child exception of the SuperClass declared Exception.

public class ExceptionOveridenRelation4 {

void method() throws RuntimeException {

System.out.println("SuperClass");

}

}

//SubClass declaring a child exception

// of RuntimeException

class SubClass4 extends ExceptionOveridenRelation4 {

// method() declaring unChecked Exception ArithmeticException

void method() throws ArithmeticException {

// ArithmeticException is a child exception

// of the RuntimeException

// So the compiler won't give an error

System.out.println("SubClass");

}

}

//case 3.3 If SuperClass declares an exception and SubClass declares without exception.

public class ExceptionOveridenRelation5 {

void method() {

System.out.println("SuperClass");

}

}

//SuperClass inherited by the SubClass

class SubClass5 extends ExceptionOveridenRelation5 {

// method() declaring unChecked Exception ArithmeticException

void method() throws ArithmeticException {

// ArithmeticException is of type unChecked Exception no compiler Error

System.out.println("SubClass");

}

}

**12 how to prevent a class form being sub classed**

By using the final keyword in the class's declaration. Similarly, you can prevent a method from being overridden by sub classes by declaring it as a final method. An abstract class can only be subclassed; it cannot be instantiated.

**13 default size of hashing based collection ?**

The default size of load factor is 0.75. The default capacity is computed as initial capacity \* load factor.

For example, 16 \* 0.75 = 12. So, 12 is the default capacity of Map

**14 possible ways to sort a collection?**

You can sort a Java List collections using the java.util.Collections.sort() method. It is applied to only List<I> same cant be used for Set or Queue or Map if we try compile time error will appear.

There are two overloaded Collections.sort() methods, which are:

1 sort(List list): Sorts the elements of the List in ascending order of their natural ordering.

2 sort(List list, Comparator c): Sorts the elements of the list according to the order induced by the [comparator](https://www.journaldev.com/780/comparable-and-comparator-in-java-example).

**Sorting Objects by their Natural Order**

To sort a List you do this:

List list = new ArrayList();

//add elements to the list

Collections.sort(list);

For objects to have a natural order they must implement the interface java.lang.Comparable.

public interface Comparable<T> {

int compareTo(T o);

}

The compareTo() method should compare this object to another object, return an int value. Here are the rules for that int value:

1 Return a negative value if this object is smaller than the other object

2 Return 0 (zero) if this object is equal to the other object.

3 Return a positive value if this object is larger than the other object.

**Sorting Objects Using a Comparator**

Sometimes you may want to sort a list according to another order than their natural order. Perhaps the objects you are sorting do not even have a natural order. In that case you can use a Comparator

List list = new ArrayList();

//add elements to the list

Comparator comparator = new SomeComparator();

Collections.sort(list, comparator);

Collections.sort() method now takes a java.util.Comparator as parameter in addition to the List. This Comparator compares the elements in the list two by two. Here is how the Comparator interface looks:

public interface Comparator<T> {

int compare(T object1, T object2);

}

The compare() method compares two objects to each other and should:

1 Return a negative value if object1 is smaller than object2

2 Return 0 (zero) if objec1 is equal to object2.

3 Return a positive value if object1 is larger than object2.

public class SortNaturalOrder {

public static void main(String[] args) {

List<String> arraylist = Arrays.asList("Geeks For Geeks", "Friends", "Dear", "Is", "Superb");

List<String> linkedlist = new LinkedList<>(arraylist);

Set<String> setList = new HashSet<>(arraylist);

System.err.println(arraylist);

Collections.sort(arraylist);

System.err.println(arraylist);

System.err.println(linkedlist);

Collections.sort(linkedlist);

System.err.println(linkedlist);

System.err.println(setList);

//Collections.sort(setList); Collections.sort only applied List insterface not to any other collections

System.err.println(setList);

Collections.sort(arraylist, Collections.reverseOrder());

System.err.println(arraylist);

}

}

class StudentSort {

int rollno;

String name, address;

// Constructor

public StudentSort(int rollno, String name, String address) {

this.rollno = rollno;

this.name = name;

this.address = address;

}

// Used to print student details in main()

public String toString() {

return this.rollno + " " + this.name + " " + this.address;

}

}

class Sortbyroll implements Comparator<StudentSort> {

// Used for sorting in ascending order of

// roll number

public int compare(StudentSort a, StudentSort b) {

return a.rollno - b.rollno;

}

}

public class StudentSortCustom {

public static void main(String[] args) {

ArrayList<StudentSort> ar = new ArrayList<StudentSort>();

ar.add(new StudentSort(111, "bbbb", "london"));

ar.add(new StudentSort(131, "aaaa", "nyc"));

ar.add(new StudentSort(121, "cccc", "jaipur"));

System.out.println("Unsorted");

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

System.out.println(ar.get(i));

Collections.sort(ar, new Sortbyroll());

System.out.println("\nSorted by rollno");

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

System.out.println(ar.get(i));

}

}

**[Arrays.sort()](https://www.geeksforgeeks.org/arrays-sort-in-java-with-examples/) vs Collections.sort()**  
Arrays.sort works for arrays which can be of primitive data type also. [Collections](https://www.geeksforgeeks.org/collections-in-java-2/).sort() works for objects Collections like [ArrayList](https://www.geeksforgeeks.org/arraylist-in-java/), [LinkedList](http://geeksquiz.com/linked-list-in-java/), etc.

**15 peek, pull method and remove method**

**Peek()**

This method returns the object at the top of the current queue, without removing it. If the queue is empty this method returns null.

The **poll()** method of the Queue interface returns the object at the top of the current queue and removes it. If the queue is empty this method returns null.

The **remove()** method of the Queue interface returns the object at the top of the current queue and removes it. If the queue is empty this method throws NoelementFoundExecption.

public class QueueSamplePeekPull {

public static void main(String[] args) {

Queue<String> queue = new LinkedList<>();

queue.add("sachin");

queue.add("darshan");

queue.add("sachin");

System.err.println(queue);

System.err.println(queue.peek());

System.err.println(queue);

System.err.println(queue.poll());

System.err.println(queue);

System.err.println(queue.remove());

System.err.println(queue);

System.err.println(queue.poll());

System.err.println(queue);

System.err.println(queue.peek()); // return null since there is no element to pick

System.err.println(queue.poll()); // return null since there is no element to pick and remove

System.err.println(queue.remove());// Exception in thread "main" java.util.NoSuchElementException no element to

// remove

System.err.println(queue);

}

}

**16 when to use blank final variable (immutable class variable)**

A final variable in Java can be assigned a value only once, we can assign a value either in declaration or later.

A blank final variable in Java is a [final](https://www.geeksforgeeks.org/g-fact-48/) variable that is not initialized during declaration.

If we have more than one constructors or overloaded constructor in class, then blank final variable must be initialized in all of them. However constructor chaining can be used to initialize the blank final variable.

Blank final variables are used to create immutable objects (objects whose members can’t be changed once initialized).

public class BlankFinal {

// We can initialize here, but if we

// initialize here, then all objects get

// the same value. So we use blank final

final int i;

BlankFinal(int x) {

// Since we have initialized above, we must initialize i in constructor.

// If we remove this line, we get compiler error.

i = x;

}

}

**17 immutable class list few?**

1 String,Wrapper classes such as Integer, Long, Double etc.

2 Immutable collection classes such as Collections.singletonMap() etc.

3 java.lang.StackTraceElement,Java enums (ideally they should be),java.util.Locale,java.util.UUID

**18 convert a class to immutable class or create and immutable class?**

Immutable objects are instances whose state doesn’t change after it has been initialized. For example, [String](https://www.journaldev.com/16928/java-string) is an immutable class and once instantiated its value never changes.

An immutable class is good for caching purposes because you don’t have to worry about the value changes.

Another benefit of immutable class is that it is inherently [thread-safe](https://www.journaldev.com/1061/thread-safety-in-java), so you don’t have to worry about thread safety in case of multi-threaded environment.

To create an immutable class in Java, you have to do the following steps.

1. Declare the class as final so it can’t be extended.
2. Make all fields private so that direct access is not allowed.
3. Don’t provide setter methods for variables.
4. Make all **mutable fields final** so that its value can be assigned only once.
5. Initialize all the fields via a [constructor](https://www.journaldev.com/18899/constructor-in-java) performing deep copy.
6. Perform [cloning](https://www.journaldev.com/60/java-clone-object-cloning-java) of objects in the getter methods to return a copy rather than returning the actual object reference.

public final class SimpleImmutable {

private final int id;

private final String name;

private final Date date;

public SimpleImmutable(int id, String name, Date date) {

super();

this.id = id;

this.name = name;

this.date = date;

}

public int getId() {

return id;

}

public String getName() {

return name;

}

public Date getDate() {

return date;

}

@Override

public String toString() {

return "SimpleImmutable [id=" + id + ", name=" + name + ", date=" + date + "]";

}

}

public final class SimpleImmutable2 {

private final int id;

private final String name;

private final Date date;

public SimpleImmutable2(int id, String name, Date date) {

super();

this.id = id;

this.name = name;

this.date = date;

}

public int getId() {

return id;

}

public String getName() {

return name;

}

public Date getDate() {

return new Date(date.getTime());

}

@Override

public String toString() {

return "SimpleImmutable [id=" + id + ", name=" + name + ", date=" + date + "]";

}

}

public class Testmutable {

public static void main(String[] args) {

SimpleImmutable sm = new SimpleImmutable(1, "sachin", new Date());

System.err.println(sm);

sm.getDate().~~setDate~~(1);

System.err.println(sm);

SimpleImmutable2 sm2 = new SimpleImmutable2(1, "sachin", new Date());

System.err.println(sm2);

sm2.getDate().~~setDate~~(1);

System.err.println(sm2);

}

}

**19 what is benefit of immutable class in real time**

1. are simple to construct, test, and use
2. are automatically thread-safe and have no synchronization issues
3. do not need a copy constructor
4. do not need an implementation of clone
5. allow [hashCode()](https://howtodoinjava.com/java/related-concepts/working-with-hashcode-and-equals-methods-in-java/" \o "Working with hashCode and equals methods in java) to use lazy initialization, and to cache its return value
6. do not need to be copied defensively when used as a field
7. make good [Map keys and Set elements](https://howtodoinjava.com/java/collections/hashmap/how-hashmap-works-in-java/" \o "How hashmap works in java) (these objects must not change state while in the collection)
8. have their class invariant established once upon construction, and it never needs to be checked again
9. always have “**failure atomicity**” (a term used by Joshua Bloch) : if an immutable object throws an exception, it’s never left in an undesirable or indeterminate state

**20 Array list adding 3 null what is the size?**

It will give size also 3 since list allows duplicate and null values everything will be counted

**21 why return type is not considered in overloading**

The constructor doesn't return a value is because it's not called directly by your code, it's called by the memory allocation and object initialization code in the runtime. The whole purpose of constructor is to initialize the current state of the object by setting the initial values.

Constructor is internally a nonstatic method with name <init> and void return type. It does not return anything. Object is not allocated with constructor itself. In other words the syntax new Object() not only calls the constructor but also creates new object and after calling the constructor returns it.

**22 loose coupling and tight coupling**

Tight coupling means classes and objects are dependent on one another. In general, tight coupling is usually not good because it reduces the flexibility and re-usability of the code while Loose coupling means reducing the dependencies of a class that uses the different class directly.

Loose coupling : In simple words, loose coupling means they are mostly independent. If the only knowledge that class A has about class B, is what class B has exposed through its interface, then class A and class B are said to be loosely coupled. In order to over come from the problems of tight coupling between objects, spring framework uses dependency injection mechanism with the help of POJO/POJI model and through dependency injection its possible to achieve loose coupling.

**Difference between tight coupling and loose coupling**

* Tight coupling is not good at the test-ability. But loose coupling improves the test ability.
* Tight coupling does not provide the concept of interface. But loose coupling helps us follow the GOF principle of program to interfaces, not implementations.
* In Tight coupling, it is not easy to swap the codes between two classes. But it’s much easier to swap other pieces of code/modules/objects/components in loose coupling.
* Tight coupling does not have the changing capability. But loose coupling is highly changeable.

public class TightCouple {

TopicT t = new TopicT();

public void startReading() {

t.understand();

}

}

class TopicT {

public void understand() {

System.out.println("Tight coupling concept");

}

}

public class LosseCoupling {

public static void main(String[] args) {

Topic t = new Topic1();

t.understand();

}

}

interface Topic {

void understand();

}

class Topic1 implements Topic {

public void understand() {

System.out.println("Got it");

}

}

class Topic2 implements Topic {

public void understand() {

System.out.println("understand");

}

}

**23 BufferReader and Scanner**

[java.util.Scanner](https://www.geeksforgeeks.org/scanner-class-in-java/) class is a simple text scanner which can parse primitive types and strings. It internally uses regular expressions to read different types.

Java.io.BufferedReader class reads text from a character-input stream, buffering characters so as to provide for the efficient reading of sequence of characters

Scanner and BufferReader both classes are used to read input from external system. Scanner is normally used when we know input is of type string or of primitive types and BufferReader is used to read text from character streams while buffering the characters for efficient reading of characters.

BufferedReader has significantly larger buffer memory than Scanner. BufferedReader is a bit faster as compared to scanner because scanner does parsing of input data and BufferedReader simply reads sequence of characters.

Use Buffered Reader if you want to get long strings from a stream, and use Scanner if you want to parse specific type of token from a stream. Scanner can use tokenize using custom delimiter and parse the stream into primitive types of data, while Buffered Reader can only read and store String.

|  |  |  |
| --- | --- | --- |
| KEY | Scanner class | Buffer Reader class |
| Synchronous | Scanner is not synchronus in nature should be for single thread case. | Synchronus in nature, suited for multi thread environment |
| Buffer memory | 1 KB | 8KB |
| Processing speed | Slower since parsing involved | Faster since it reads only char stream |
| method | nextInt(),nextLong() | parseInt(),parseShort() |
| Read line | nextLine() | readLine() |

In Scanner class if we call nextLine() method after any one of the seven nextXXX() method then the nextLine() does not read values from console and cursor will not come into console it will skip that step. The nextXXX() methods are nextInt(), nextFloat(), nextByte(), nextShort(), nextDouble(), nextLong(), next().

In BufferReader class there is no such type of problem. This problem occurs only for Scanner class, due to nextXXX() methods ignore newline character and nextLine() only reads till first newline character. If we use one more call of nextLine() method between nextXXX() and nextLine(), then this problem will not occur because nextLine() will consume the newline character.

For BufferReader in class

1. Instantiate an InputStreamReader class bypassing your InputStream object as a parameter.
2. Then, create a **BufferedReader**, bypassing the above obtained InputStreamReader object as a parameter.
3. Now, read data from the current reader as String **using** the readLine() or read() method.

public class ScannerSample {

public static void main(String[] args) {

Scanner scn = new Scanner(System.in);

System.out.println("Enter an integer");

int a = scn.nextInt();

System.out.println("Enter a String");

// scn.nextLine(); // uncomment this to fix skiping of next char in nxt line

String b = scn.nextLine();

System.out.printf("You have entered:- " + a + " " + "and name as " + b);

}

}

public class BufferReaderSample {

public static void main(String args[]) throws IOException {

// BufferReader requires InputStreamReader to read the object

BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

System.out.println("Enter an integer");

int a = Integer.parseInt(br.readLine());

System.out.println("Enter a String");

String b = br.readLine();// read lines has stream of character

System.out.printf("You have entered:- " + a + " and name as " + b);

}

}

**24 ClassNot foundException and NoClassDefFoundError**

|  |  |
| --- | --- |
| ClassNotFoundException | NoClassDefFoundError |
| It is an exception(runtime exception). It is of type java.lang.Exception. | It is an error. It is of type java.lang.Error. |
| It occurs when an application tries to load a class at run time which is not updated in the classpath. | It occurs when java runtime system doesn’t find a class definition, which is present at compile time, but missing at run time. |
| It is thrown by the application itself. It is thrown by the methods like Class.forName(), loadClass() and findSystemClass(). | It is thrown by the Java Runtime System. |
| It occurs when classpath is not updated with required JAR files. Jdbs loading class | It occurs when required class definition is missing at runtime. |

For example, you may have come across this exception when you try to connect to MySQL or Oracle databases and you have not updated the classpath with required JAR files. Most of the time, this exception occurs when you try to run an application without updating the classpath with required JAR files.NoClassDefFoundError is an error that occurs when a particular class is present at compile time, but was missing at run time

When you compile the above program, two .class files will be generated. One is A.class and another one is B.class. If you remove the A.class file and run the B.class file, Java Runtime System will throw NoClassDefFoundError .

**25 can we override static method**

NO

Static methods cannot be overridden because they are class level not object level . Static methods can be overloaded.

We can declare static methods with the same signature in the subclass, but it is not considered overriding as there won't be any run-time polymorphism it is called method hiding. method overriding is based on dynamic binding at runtime and the static methods are bonded using static binding at compile time.

we can not override main method in java.As main method is static and we know very well that we can not override static methods in Java,hence main method could not be overridden..

Static methods with the same signature from the parent class are hidden when called from an instance of the subclass. However, you can't override/hide final methods. You would think the error message would use the word hidden instead of overridden

**Overriding**: Overriding in Java simply means that the particular method would be called based on the run time type of the object and not on the compile time type of it.

**Hiding**: Parent class methods that are static are not part of a child class (although they are accessible), so there is no question of overriding it. Even if you add another static method in a subclass, identical to the one in its parent class, this subclass static method is unique and distinct from the static method in its parent class.

class StaticOverLoad {

public static void testMethod() {

System.err.println("parent class");

}

}

class StaticoverLoadChild extends StaticOverLoad {

public static void testMethod() {

System.err.println("child class");

}

}

public class StaticOverLoadTest {

public static void main(String[] args) {

StaticoverLoadChild sc = new StaticoverLoadChild();

sc.testMethod(); // direct child object child

StaticOverLoad su = sc;

StaticOverLoad.testMethod(); // static child method class parent

su.testMethod();// method hiding parent

StaticoverLoadChild.testMethod(); //child

StaticOverLoad sp = new StaticoverLoadChild();

sp.testMethod();// parent

}

}

**26 where all we can use static**

Static keyword is mainly for memory management.Can be used with variables, methods, blocks and nested classes. It is a keyword which is used to share the same variable or method of a given class. Basically, static is used for a constant variable or a method that is same for every instance of a class.Class level not object level.

The "this" keyword is used as a reference to an instance. Since the static methods doesn't have (belong to) any instance you cannot use the "this" reference within a static method.

You can invoke static methods without creating an object.Where, the "super" keyword in Java is used as a reference to the object of the super class. This implies that to use "super" the method should be invoked by an object, which static methods are not. Therefore, you cannot use "super" keyword from a static method.

Java main() method is always static, so that compiler can call it without the creation of an object or before the creation of an object of the class. Static method of a class can be called by using the class name only without creating an object of a class.

In order to create a static member (block, variable, method, nested class), you need to precede its declaration with the keyword static. When a member of the class is declared as static, it can be accessed before the objects of its class are created, and without any object reference.

In Java programming language, static keyword is a non-access modifier and can be used for the following:

### **Static Block**

If you need to do the computation in order to initialize your static variables, you can declare a static block that gets executed exactly once, when the class is first loaded.

### **Static Variable**

When you declare a variable as static, then a single copy of the variable is created and divided among all [objects](https://www.edureka.co/blog/java-tutorial/" \l "obj" \t "https://www.edureka.co/blog/static-keyword-in-java/_blank) at the [class level](https://www.edureka.co/blog/java-objects-and-classes/" \t "https://www.edureka.co/blog/static-keyword-in-java/_blank). Static variables are, essentially, global variables. Basically, all the instances of the class share the same static variable. Static variables can be created at class-level only.

### **Static Method**

When a method is declared with the static keyword, it is known as a static method. The most common example of a static method is the main( ) method.  Methods declared as static can have the following restrictions:

They can directly call other static methods only.

They can access static data directly.

restrictions are imposed on the static methods and also how you are allowed to use super keyword in the static context. That was all about Static Methods.

### **Static Class**

A class can be made static only if it is a nested class. Nested static class doesn’t need a reference of Outer class. In this case, a static class cannot access non-static members of the Outer class.

public class StaticClassDemo {

// static variable

static int j = 10;

static int n;

// static variable

static int x = n();

// static block

static {

System.out.println("Static block initialized.");

n = j \* 8;

}

// static method

static int n() {

System.out.println("from n ");

new StaticClassDemo().a2();// works in ststic blocak since object created

// a2(); compileerror since its non ststci object level cant be called directly

// like static

return 20;

}

// instance method

void a2() {

System.out.println("Inside a2");

}

private static String str = "Edureka";

// Static class

static class MyNestedClass {

// non-static method

public void disp() {

System.out.println(str);

}

}

public static void main(String[] args) {

System.out.println("Inside main method");

System.out.println("Value of j : " + j);

System.out.println("Value of n : " + n);

System.out.println("Value of n : " + x);

StaticClassDemo.MyNestedClass obj = new StaticClassDemo.MyNestedClass();

obj.disp();

}

}

**27 serialized and externalized interface in java**

To serialize an object means to convert its state to a byte stream so that the byte stream can be reverted back into a copy of the object. A Java object is serializable if its class or any of its superclasses implements either the java. io. Serializable interface or its subinterface, java. io.

serialization allows us to convert the state of an object into a byte stream, which then can be saved into a file on the local disk or sent over the network to any other machine. And deserialization allows us to reverse the process, which means reconverting the serialized byte stream to an object again.

Serialization and externalization both are the processes of converting an object to stream byte and storing byte stream in database or memory. The class which implements java.io.Serializable interface can be serialized. On the other hand, externalization used for custom serialization based on the requirement in the application. Externalization extends java.io.Serializable.

Externalization serves the purpose of custom Serialization, where we can decide what to store in stream.  
 It consist of two methods which we have to override to write/read object into/from stream which are-

// to read object from stream

void readExternal(ObjectInput in)

// to write object into stream

void writeExternal(ObjectOutput out)

| Key | Serialization | Externalization |
| --- | --- | --- |
| Interface | Serialization is a marker interface | Externalization contains two methods readExternal and writeExternal. |
| Implementation logic | class implementing this interface gives responsibility to JVM for serializing or persist java object.  JVM use readObject and writeObject for serialization | Externalization provides implementation logic control to the application by overriding readExternal and writeExternal methods. |
| Way to ignore variables | In serialization, JVM ignores transient variable during serialization and deserialization of java object | Programmer can write their own logic to ignore some of the variables during externalization of java object |
| Performance | In serializable interface uses reflection causes relatively slow performance. | Externalizable gives full control over the implementation approach. |
| Object serialization with inheritance | 1. If the superclass is not serializable then the subclass still can be serialized. 2. If a subclass is not serialized but superclass is automatically serializable | We can apply this to externalizable as well |

public class ExternalizableExample implements Externalizable {

Integer id;

@Override

public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException {

this.id = in.readInt();

}

@Override

public void writeExternal(ObjectOutput out) throws IOException {

out.writeInt(id);

}

}

class SerializableExample implements Serializable {

private static final long serialVersionUID = -2731167082517723325L;

String name;

}

**28 wild card generics**

Java Generic's wildcards is a mechanism in Java Generics aimed at making it possible to cast a collection of a certain class, e.g A, to a collection of a subclass or superclass of A.

The question mark (?) is known as the wildcard in generic programming . It represents an unknown type. The wildcard can be used in a variety of situations such as the type of a parameter, field, or local variable; sometimes as a return type. Unlike arrays, different instantiations of a generic type are not compatible with each other, not even explicitly. This incompatibility may be softened by the wildcard if ? is used as an actual type parameter.

The need for making assignments of the type arises when creating reusable methods that operate on collections of a specific type.

List<?> listUknown = new ArrayList<A>();

List<? extends A> listUknown = new ArrayList<A>();

List<? super A> listUknown = new ArrayList<A>();

**Types of wildcards in Java:**

**Upper Bounded Wildcards:** These wildcards can be used when you want to relax the restrictions on a variable. For example, say you want to write a method that works on List < integer >, List < double >, and List < number > , you can do this  using an upper bounded wildcard.  
To declare an upper-bounded wildcard, use the wildcard character (‘?’), followed by the extends keyword, followed by its upper bound.

public static void add(List<? extends Number> list)

**Lower Bounded Wildcards:** It is expressed using the wildcard character (‘?’), followed by the super keyword, followed by its lower bound: <? super A>.

Syntax: Collectiontype <? super A>

**Unbounded Wildcard:** This wildcard type is specified using the wildcard character (?), for example, List. This is called a list of unknown type. These are useful in the following cases

When writing a method which can be employed using functionality provided in Object class.

When the code is using methods in the generic class that don’t depend on the type parameter

Use extend wildcard when you want to get values out of a structure and super wildcard when you put values in a structure. Don’t use wildcard when you get and put values in a structure.

Note: You can specify an upper bound for a wildcard, or you can specify a lower bound, but you cannot specify both.

public class WildCardGenericeDemo {

public static void main(String[] args) {

// Upper Bounded Integer List

List<Integer> list1 = Arrays.asList(4, 5, 6, 7);

// printing the sum of elements in list

System.out.println("Total sum is:" + sum(list1));

// Double list

List<Double> list2 = Arrays.asList(4.1, 5.1, 6.1);

// printing the sum of elements in list

System.out.print("Total sum is:" + sum(list2));

// Lower Bounded Integer List

List<Integer> list3 = Arrays.asList(4, 5, 6, 7);

// Integer list object is being passed

printOnlyIntegerClassorSuperClass(list3);

// Number list

List<Number> list4 = Arrays.asList(4, 5, 6, 7);

// Integer list object is being passed

printOnlyIntegerClassorSuperClass(list4);

// printOnlyIntegerClassorSuperClass(list2); it is of Interger type so only

// Interger and its parent not Double compile time error

// unbound generics

printlist(list1);

printlist(list2);

}

// Number is parent of Integer and double

private static double sum(List<? extends Number> list) {

double sum = 0.0;

for (Number i : list) {

sum += i.doubleValue();

}

return sum;

}

//

public static void printOnlyIntegerClassorSuperClass(List<? super Integer> list) {

System.out.println(list);

}

private static void printlist(List<?> list) {

System.out.println(list);

}

}

**29 Advantage from generics**

Generics enable the use of stronger type-checking at compile time, A Java compiler applies strong type checking to generic code and issues errors if the code violates type safety.the elimination of casts, and the ability to develop generic algorithms. Without generics, many of the features that we use in Java today would not be possible.

**30 Arraylist Declaration to understand concept of generics**

The following code snippet without generics requires casting:

List list = new ArrayList();

list.add("hello");

String s = (String) list.get(0);

When re-written to use generics, the code does not require casting:

List<String> list = new ArrayList<String>();

list.add("hello");

String s = list.get(0); // no cast

**31 when overriding equals()method y equals() and comareTo() must be kept consistent**

CompareTo method comapre 2 objects for String in lexical order and numeric order for integer. String and Integer implements CompareTo()

1) CompareTo method must return negative number if current object is less than other object, positive number if current object is greater than other object and zero if both objects are equal to each other.

2 )CompareTo must be in consistent with [equals method](http://javarevisited.blogspot.com/2011/02/how-to-write-equals-method-in-java.html) e.g. if two objects are equal via equals() , there compareTo() must return zero otherwise if those objects are stored in SortedSet or SortedMap they will not behave properly.

Since SortedSet or SortedMap use compareTo() to check the object if two unequal object are returned equal by compareTo those will not be added into Set or Map if they are not using external Comparator.

One example where compareTo is not consistent with equals in JDK is BigDecimal class. two BigDecimal number for which compareTo returns zero, equals returns false as clear from following BigDecimal comparison example:

if you store these two BigDecimal in HashSet you will end up with duplicates (violation of Set Contract) i.e. two elements while if you store them in TreeSet you will end up with just 1 element because HashSet uses equals to check duplicates while TreeSet uses compareTo to check duplicates. That's why its suggested to keep compareTo consistent with equals method in java.

3) CompareTo() must throw NullPointerException if current object get compared to null object as opposed to equals() which return false on such scenario.

4) Another important point to note is don't use subtraction for comparing integral values because result of subtraction can overflow as every int operation in Java is modulo 2^32. use either Integer.compareTo()  or logical operators for comparison.

5. Use relational operator to compare integral numeric value i.e. < or > but use Float.compareTo() or Double.compareTo() to compare [floating point number](http://javarevisited.blogspot.com/2011/10/convert-double-to-string-example.html) as relational operator doesn't obey contract of compareTo for floating point numbers.

6. CompareTo() method is for comparison so order in which you compare two object matters. If you have more than one significant field to compare than always start comparing from most significant field to least significant field. 

7. Another important point while comparing String using compareTo is to consider case. just like equals() doesn't consider case, compareTo also do not consider case, if you want to compare regardless of case than use String.compareToIgnoreCase()

public class EqaulsComparTo {

public static void main(String[] args) {

BigDecimal bd1 = new BigDecimal("2.0");

BigDecimal bd2 = new BigDecimal("2.00");

System.out.println("comparing BigDecimal using equals: " + bd1.equals(bd2));// fasle //not equal

System.out.println("comparing BigDecimal using compareTo: " + bd1.compareTo(bd2));// 0 equals

}

}

**32 final and effective final**

effectively final variables. A variable which is not declared as final but whose value is never changed after initialization is effectively final.This means you can now use the local variable without the final keyword inside an anonymous class or lambda expression, provided they must be effectively final.

Local Variables in Capturing Lambdas. start is a local variable, and we are trying to modify it inside of a lambda expression. The basic reason this won't compile is that the lambda is capturing the value of start, meaning making a copy of it. If we try to change compiler time error will appear

public class ClosureExample {

public static void main(String[] args) {

int a = 10;

int b = 20; // it will be efffective final cnt chnge if we cahnge compiler error

doProcess(a, new Process() {

@Override

public void process(int i) {

// b = i + b; cnat be changed it effective final variable

// System.err.println(i + b);

}

});

doProcess(b, (i) -> System.err.println(i + b));

}

public static void doProcess(int b, Process pro) {

pro.process(b);

}

}

**33 reverse linked list**

1. Initialize three pointers prev as NULL, curr as head and next as NULL.
2. Iterate through the linked list. In loop, do following.   
   // Before changing next of current,   
   // store next node   
   next = curr->next  
   // Now change next of current   
   // This is where actual reversing happens   
   curr->next = prev   
   // Move prev and curr one step forward   
   prev = curr   
   curr = next

**34 how to prevent overloading instead of overriding**

You can **prevent** a method from being overwritten by making it final, but you cannot **prevent** a method from being **overloaded**.

**35 wrapper class?**

The wrapper class in Java provides the mechanism to convert primitive into object and object into primitive. A Wrapper class is a class whose object wraps or contains primitive data types. When we create an object to a wrapper class, it contains a field and in this field, we can store primitive data types. In other words, we can wrap a primitive value into a wrapper class object.

Java Wrapper Classes. Wrapper classes provide a way to use primitive data types ( int , boolean , etc..) as objects.

Wrapper Class will convert primitive data types into objects. The objects are necessary if we wish to modify the arguments passed into the method (because primitive types are passed by value). The classes in java. util package handles only objects and hence wrapper classes help in this case also.

The primary advantage of Wrapper Classes is that we need Wrapper objects to function with collections which is only possible with the help of Wrapper classes. As the wrapper classes have objects we can store null as a value. We could not store null in variables of primitive datatype.

All primitive wrapper classes (Integer, Byte, Long, Float, Double, Character, Boolean and Short) are immutable in Java, so operations like addition and subtraction create a new object and not modify the old. i = i + 1; ... Assign the resulting Integer to i (thus changing what object i references)

In computer science, a wrapper is any entity that encapsulates (wraps around) another item. Wrappers are used for two primary purposes: to convert data to a compatible format or to hide the complexity of the underlying entity using abstraction. Examples include object wrappers, function wrappers, and driver wrappers.

**36 have u applied object cloning in the project**

Yes as per the requirement deep copey cloning in case of its an object.

Ledger record same will copied before any calculation.

We are using Object clone() method, so we have implemented the Cloneable [interface](https://www.journaldev.com/1601/interface-in-java). We are calling the superclass clone() method i.e. Object clone() method.

when the cloning process is done by implementing the Cloneable interface it is called Deep Cloning. In this type of cloning, an exact copy of all the fields of the original object will be created

If you want to use Java Object clone() method, you have to implement the java.lang.Cloneable marker interface. Otherwise, it will throw CloneNotSupportedException at runtime.

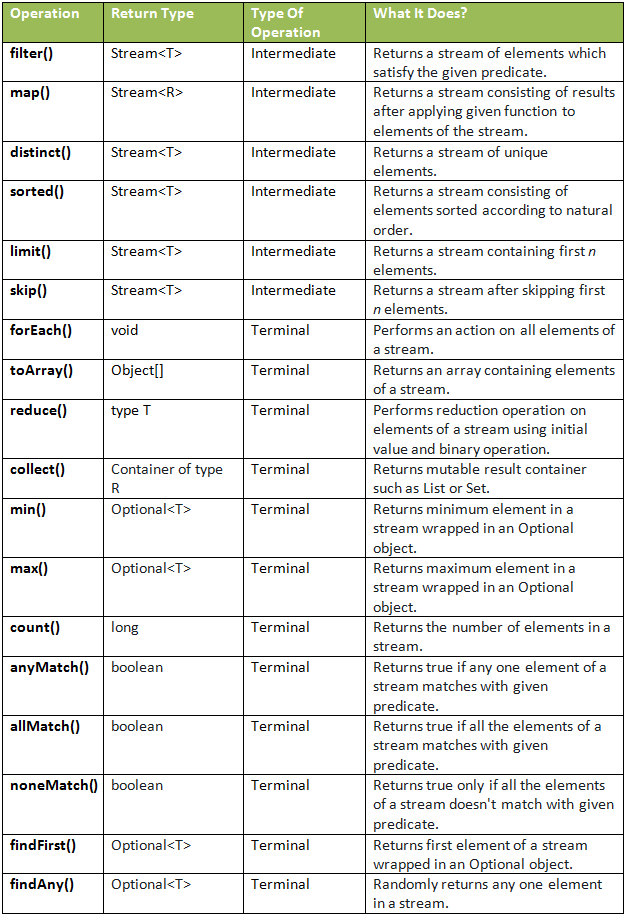
**37 terminal and intermediate operation in java 8**

When you invoke an intermediate operation on a stream, the operation is not executed immediately. It is executed only when a terminal operation is invoked on that stream. ... Terminal operations produces a non-stream (cannot be chained) result such as primitive value, a collection or no value at all.

Intermediate operations are lazily loaded. When you call intermediate operations, they are actually not executed. They are just stored in the memory and executed when the terminal operation is called on the stream.

As the names suggest, intermediate operations doesn’t give end result. They just transform one stream to another stream. On the other hand, terminal operations give end result.

|  |  |
| --- | --- |
| Intermediate Operations | Terminal Operations |
| intermediate operations return another stream as a result They return stream. | They return non-stream values.primitive or object or collection or may not return anything |
| They can be chained together to form a pipeline of operations. | They can’t be chained together. |
| Pipeline of operations may contain any number of intermediate operations. | Pipeline of operations can have maximum one terminal operation, that too at the end. |
| Intermediate operations are lazily loaded. | Terminal operations are eagerly loaded. |
| They don’t produce end result. | They produce end result. |
| Examples : filter(), map(), distinct(), sorted(), limit(), skip() | Examples : forEach(), toArray(), reduce(), collect(), min(), max(), count(), anyMatch(), allMatch(), noneMatch(), findFirst(), findAny() |



**38 purpose of defaulter and static method in interface**

All the methods of interfaces are public & abstract by default. Java 8 allows the interfaces to have default and static methods. The reason we have default methods in interfaces is to allow the developers to add new methods to the interfaces without affecting the classes that implements these interfaces

default and static methods in interfaces. This feature enables us to add new functionality in the interfaces without breaking the existing contract of the implementing classes.

Default methods enable you to add new functionality to existing interfaces and ensure binary compatibility with code written for older versions of those interfaces. In particular, default methods enable you to add methods that accept lambda expressions as parameters to existing interfaces.

Stream API and Lambda supporting API or example of default method

Default methods cant be have same method name as Object class method, any method override object class method it will give error

Static method can be accessed with even creating object of that interface

default void newMethod(){

System.out.println("Newly added default method");

}

static void anotherNewMethod(){

System.out.println("Newly added static method");

}

This is a static method. Static method in interface is similar to default method except that we cannot override them in the implementation classes. Similar to default methods, we need to implement these methods in implementation classes so we can safely add them to the existing interfaces.

This is a default method so we need not to implement this method in the implementation classes

**39 diff between map and filter**

The map operation allows us to apply a function, that takes in a parameter of one type, and returns something else.

Filter is used for filtering the data, it always returns the boolean value. If it returns true, the item is added to list else it is filtered out (ignored)

these two methods are intermediate operations and return a Stream as a result.

map() and filter() methods can be invoked on a single stream. There is no restriction to use the only operation at a time.

The map() will then return a Stream of Integer that contains both even and odd numbers. To select just even numbers, we can use the filter() method. It takes a predicate object which is technically a function to convert an object to boolean. We pass an object and it will return true or false.

the map() is used to transform one object into other by applying a function.

Stream.map(Function mapper) takes a function as an argument. For example, by using the map() function, you can convert a list of String into a List of Integer by applying the [Integer.valueOf()](https://javarevisited.blogspot.sg/2011/08/convert-string-to-integer-to-string.html" \t "https://dzone.com/articles/_blank) method to each String on the input list.

All you need is a mapping function to convert one object to the other. Then, the [map()](https://javarevisited.blogspot.sg/2016/03/difference-between-map-and-flatmap-in-java8.html) function will do the transformation for you. It is also an intermediate Stream operation, which means you can call other Stream methods, like a filter, or collect on this to create a chain of transformations.

The [filter](http://www.java67.com/2018/03/java-8-stream-find-first-and-filter-example.html" \t "https://dzone.com/articles/_blank) method, as its name suggests, filters elements based upon a condition you gave it. For example, if your list contains numbers and you only want numbers, you can use the filter method to only select a number that is fully divisible by two.

The filter method essentially selects elements based on a condition you provide. That's why the filter (Predicate condition) accepts a Predicate object, which provides a function that is applied to a condition. If the condition evaluates true, the object is selected. Otherwise, it will be ignored

**40 this and super keyword and Method**

super keyword is used to access methods of the parent class while this is used to access methods of the current class. this is a reserved keyword in java i.e, we can't use it as an identifier. this is used to refer current-class's instance as well as static members.

The "this" keyword is used as a reference to an instance. Since the static methods doesn't have (belong to) any instance you cannot use the "this" reference within a static method.

In static block or a static method, there is no instance to refer to, and therefore the "this" keyword is not permitted.

The super keyword in Java is a reference variable that is used to refer parent class objects. The super() in Java is a reference variable that is used to refer parent class constructors. super can be used to call parent class' variables and methods. super() can be used to call parent class' constructors only.

this() and super(), both are the constructors that's why must be the first statement. But we can use both in a program. this(): It is used to call, same class Default or Parametrized Constructor. super(): It is used to call, immediate super/parent class Default or Parametrized Constructor.

**41 fail safe and Fail fast**

The major **difference** is **fail**-**safe** iterator doesn't throw any Exception, contrary to **fail**-**fast** Iterator. This is because they work on a clone of Collection instead of the original collection and that's why they are called as the **fail**-**safe** iterator.

**42 final finally and finalize**

**Final** class can't be inherited, **final** method can't be overridden and **final** variable value can't be changed. **Finally** is used to place important code, it will be executed whether exception is handled or not. **Finalize** is used to perform clean up processing just before object is garbage collected. ... **Finalize** is a method.

**43 how to serialize an object**

object serialization where an object can be represented as a sequence of bytes that includes the object's data as well as information about the object's type and the types of data stored in the object.

After a serialized object has been written into a file, it can be read from the file and deserialized that is, the type information and bytes that represent the object and its data can be used to recreate the object in memory.

Most impressive is that the entire process is JVM independent, meaning an object can be serialized on one platform and deserialized on an entirely different platform.

Classes ObjectInputStream and ObjectOutputStream are high-level streams that contain the methods for serializing and deserializing an object.

The ObjectOutputStream class contains many write methods for writing various data types, but one method in particular stands out −

public final void writeObject(Object x) throws IOException

The above method serializes an Object and sends it to the output stream. Similarly, the ObjectInputStream class contains the following method for deserializing an object −

public final Object readObject() throws IOException, ClassNotFoundException

Notice that for a class to be serialized successfully, two conditions must be met −

The class must implement the java.io.Serializable interface

All of the fields in the class must be serializable. If a field is not serializable, it must be marked transient.

public class SerializationDemo {

public static void main(String[] args) throws IOException, ClassNotFoundException {

EmpolyeeClass e = new EmpolyeeClass();

e.name = "Reyan Ali";

e.address = "Phokka Kuan, Ambehta Peer";

e.SSN = 11122333;

e.number = 101;

FileOutputStream fout = new FileOutputStream("employee.ser");

ObjectOutputStream out = new ObjectOutputStream(fout);

out.writeObject(e);

out.close();

fout.close();

System.out.printf("Serialized data is saved in employee.ser");

EmpolyeeClass e1 = null;

FileInputStream Filein = new FileInputStream("employee.ser");

ObjectInputStream in = new ObjectInputStream(Filein);

e1 = (EmpolyeeClass) in.readObject();

System.out.println("Deserialized Employee...");

System.out.println("Name: " + e1.name);

System.out.println("Address: " + e1.address);

System.out.println("SSN: " + e1.SSN);

System.out.println("Number: " + e.number);

}

}

class EmpolyeeClass implements Serializable {

public String name;

public String address;

public transient int SSN;

public int number;

public void mailCheck() {

System.out.println("Mailing a check to " + name + " " + address);

}

}

**SerialVersionUID**  
The Serialization runtime associates a version number with each Serializable class called a SerialVersionUID, which is used during Deserialization to verify that sender and reciever of a serialized object have loaded classes for that object which are compatible with respect to serialization. If the reciever has loaded a class for the object that has different UID than that of corresponding sender’s class, the Deserialization will result in an **InvalidClassException**. A Serializable class can declare its own UID explicitly by declaring a field name.  
It must be static, final and of type long.  
i.e- ANY-ACCESS-MODIFIER static final long serialVersionUID=42L;

If a serializable class doesn’t explicitly declare a serialVersionUID, then the serialization runtime will calculate a default one for that class based on various aspects of class, as described in Java Object Serialization Specification.

**Points to remember**  
1. If a parent class has implemented Serializable interface then child class doesn’t need to implement it but vice-versa is not true.  
2. Only non-static data members are saved via Serialization process.  
3. Static data members and transient data members are not saved via Serialization process.So, if you don’t want to save value of a non-static data member then make it transient.  
4. Constructor of object is never called when an object is deserialized.  
5. Associated objects must be implementing Serializable interface.  
Example :

class A implements Serializable{

// B also implements Serializable

// interface.

B ob=new B();

}

**44 To not to make serailzie**

Use transient keyword to make it non serializable

In case of **transient variables:-** A variable defined with transient keyword is not serialized during serialization process.This variable will be initialized with default value during deserialization. (e.g: for objects it is null, for int it is 0).

In case of **static Variables:-** A variable defined with static keyword is not serialized during serialization process.This variable will be loaded with current value defined in the class during deserialization

**45 synchronized block and method**

**46 static method and block**

**47 y wait notify notify are kept in object class**

If **wait**() and **notify**() were on the Thread instead then each thread would have to know the status of every other thread and there is no way to know thread1 that thread2 was **waiting** for any resource to access. Hence, **notify**, **wait**, **notifyAll** methods are defined in **object class** in Java.

1. Object has monitors.
2. Multiple threads can access one Object. Only one thread can hold object monitor at a time for synchronized methods/blocks.
3. wait(), notify() and notifyAll() method being in Object class allows all the threads created on that object to communicate with other
4. Locking ( using synchronized or Lock API) and Communication (wait() and notify()) are two different concepts.

If Thread class contains wait(), notify() and notifyAll() methods, then it will create below problems:

1. Thread communication problem
2. Synchronization on object won’t be possible. If each thread will have monitor, we won’t have any way of achieving synchronization
3. Inconsistency in state of object

48

For better understanding why wait() and notify() method belongs to Object class, I'll give you a real life example: Suppose a gas station has a single toilet, the key for which is kept at the service desk. The toilet is a shared resource for passing motorists. To use this shared resource the prospective user must acquire a key to the lock on the toilet. The user goes to the service desk and acquires the key, opens the door, locks it from the inside and uses the facilities.

Meanwhile, if a second prospective user arrives at the gas station he finds the toilet locked and therefore unavailable to him. He goes to the service desk but the key is not there because it is in the hands of the current user. When the current user finishes, he unlocks the door and returns the key to the service desk. He does not bother about waiting customers. The service desk gives the key to the waiting customer. If more than one prospective user turns up while the toilet is locked, they must form a queue waiting for the key to the lock. Each thread has no idea who is in the toilet.

Obviously in applying this analogy to Java, a Java thread is a user and the toilet is a block of code which the thread wishes to execute. Java provides a way to lock the code for a thread which is currently executing it using the synchronized keyword, and making other threads that wish to use it wait until the first thread is finished. These other threads are placed in the waiting state. Java is NOT AS FAIR as the service station because there is no queue for waiting threads. Any one of the waiting threads may get the monitor next, regardless of the order they asked for it. The only guarantee is that all threads will get to use the monitored code sooner or later.

Finally the answer to your question: the lock could be the key object or the service desk. None of which is a Thread.

However, these are the objects that currently decide whether the toilet is locked or open. These are the objects that are in a position to notify that the bathroom is open (“notify”) or ask people to wait when it is locked wait.

As every object in java has only one lock(monitor) andwait(),notify(),notifyAll() are used for monitor sharing thats why they are part of Object class rather than Threadclass.

**48 copy constructor?**

A copy constructor in a Java class is a construct or that creates an object using another object of the same Java class. That's helpful when we want to copy a complex object that has several fields, or when we want to make a deep copy of an existing object.

Steps

That's helpful when we want to copy a complex object that has several fields, or when we want to make a [deep copy](https://www.baeldung.com/java-deep-copy) of an existing object.

## 2. How to Create a Copy Constructor

To create a copy constructor, we can first declare a constructor that takes an object of the same type as a parameter:

Then, we copy each field of the input object into the new instance:

What we have here is a shallow copy, which is fine since all of our fields – an int and a String in this case – are either [primitive types](https://www.baeldung.com/java-primitives) or [immutable types](https://www.baeldung.com/java-immutable-object).

If the Java class has mutable fields, then we can instead make a [deep copy](https://www.baeldung.com/java-deep-copy) inside its copy constructor. With a deep copy, the newly created object is independent of the original one because we create a distinct copy of each mutable object:

## Copy Constructor vs. Clone

In Java, we can also use the [clone](https://www.baeldung.com/java-deep-copy) method to create an object from an existing object. However, the copy constructor has some advantages over the clone method:

1. The copy constructor is much easier to implement. We do not need to implement the Cloneable interface and handle CloneNotSupportedException.
2. The clone method returns a general Object reference. Therefore, we need to typecast it to the appropriate type.
3. We can not assign a value to a final field in the clone method. However, we can do so in the copy constructor.

## 4. Inheritance Issues

Copy constructors in Java are not inheritable by subclasses. Therefore, if we try to initialize a child object from a parent class reference, we will face a casting issue when cloning it with the copy constructor.

**49 access spefiers default ,protected , private**

**50 Solid principles**

**It increased maintainability and felxibility**

**Single Responsibility**

**- A class should have one purpose and one reason to change the code focus on one purpose**

**- ex creating ENdpoint with same code**

**Open-Closed Principle**

**-avoding long if else blocks making separte instacne blocks so mutiple developer can work**

**Liskov Substitiution**

**- think when every time sub class created**

**Interface Segregation**

**Dependency inversion**

|  |  |  |
| --- | --- | --- |
| SRP | [The Single Responsibility Principle](https://docs.google.com/open?id=0ByOwmqah_nuGNHEtcU5OekdDMkk" \t "https://www.jrebel.com/blog/_blank) | A class should have one, and only one, reason to change. |
| OCP | [The Open Closed Principle](http://docs.google.com/a/cleancoder.com/viewer?a=v&pid=explorer&chrome=true&srcid=0BwhCYaYDn8EgN2M5MTkwM2EtNWFkZC00ZTI3LWFjZTUtNTFhZGZiYmUzODc1&hl=en" \t "https://www.jrebel.com/blog/_blank) | You should be able to extend a classes behavior, without modifying it. |
| LSP | [The Liskov Substitution Principle](http://docs.google.com/a/cleancoder.com/viewer?a=v&pid=explorer&chrome=true&srcid=0BwhCYaYDn8EgNzAzZjA5ZmItNjU3NS00MzQ5LTkwYjMtMDJhNDU5ZTM0MTlh&hl=en" \t "https://www.jrebel.com/blog/_blank) | Derived classes must be substitutable for their base classes. |
| ISP | [The Interface Segregation Principle](http://docs.google.com/a/cleancoder.com/viewer?a=v&pid=explorer&chrome=true&srcid=0BwhCYaYDn8EgOTViYjJhYzMtMzYxMC00MzFjLWJjMzYtOGJiMDc5N2JkYmJi&hl=en" \t "https://www.jrebel.com/blog/_blank) | Make fine grained interfaces that are client specific. |
| DIP | [The Dependency Inversion Principle](http://docs.google.com/a/cleancoder.com/viewer?a=v&pid=explorer&chrome=true&srcid=0BwhCYaYDn8EgMjdlMWIzNGUtZTQ0NC00ZjQ5LTkwYzQtZjRhMDRlNTQ3ZGMz&hl=en" \t "https://www.jrebel.com/blog/_blank) | Depend on abstractions, not on concretions. |

**51 What is argument type for filters of streams**

Predicate Functional Interface which take variable and produces boolean output.

**52 Functional interface**

All functional interfaces are recommended to have an informative @FunctionalInterface annotation. This not only clearly communicates the purpose of this interface, but also allows a compiler to generate an error if the annotated interface does not satisfy the conditions.

Any interface with a SAM(Single Abstract Method) is a functional interface, and its implementation may be treated as lambda expressions.

The most simple and general case of a lambda is a functional interface with a method that receives one value and returns another. This function of a single argument is represented by the Function interface which is parameterized by the types of its argument and a return value:

public interface Function<T, R> { … }

**53 Built in Functional Interface**

### 1) Function Interface

The Function interface has only one single method apply(). It can accept an object of any data type and returns a result of any datatype.

public class FunctionInterfaceDemo {

public static void main(String[] args) {

String[] countries = { "India", "Australia", "England", "South Africa", "Srilanka", "Newzealand", "West Indies",

"Scotland" };

Function<String[], String> converter = (all) -> { // lambda expression

String names = "";

for (String n : all) {

String result = n.substring(0, n.indexOf(""));

result = n.substring(n.indexOf("")) + " " + result;

names += result + "\n";

}

return names;

};

converter.apply(countries);

}

}

### 2) Supplier Interface

A Supplier interface has only one single method called get(). It does not accept any arguments and returns an object of any data type.

public class SupplierInterfaceDemo {

public static void main(String[] args) {

String[] countries = { "India", "Australia", "England", "South Africa", "Srilanka", "Newzealand",

"West Indies" };

List<String> countryList = Arrays.asList(countries);

listBeginWith(countryList, (s) -> s.startsWith("I"));

listBeginWith(countryList, (s) -> s.contains("I"));

listBeginWith(countryList, (s) -> s.endsWith("ia"));

}

private static void listBeginWith(List<String> list, Predicate<String> valid) {

printNames(() -> "\nList of countries:");

list.forEach(country -> { // lambda expression

if (valid.test(country)) {

printNames(() -> country);

}

});

}

private static void printNames(Supplier<String> arg) {

System.out.println(arg.get());

}

}

### 3) Consumer Interface

The Consumer interface has only one single method called accept(). It accepts a single argument of any data type and does not return any result.

public class ConusmerInterfaceDemo {

public static void main(String[] args) {

String[] countries = { "India", "Australia", "England", "South Africa", "Srilanka", "Newzealand",

"West Indies" };

System.out.print("The list of countries:\n");

Arrays.asList(countries).forEach((country) -> System.out.println(country)); // lambda expression

}

}

### 4) Predicate Interface

The Predicate interface has only one single method test(). It may be true or false depending on the values of its variables.

public class PredicateInterfaceDemo {

private static List getBeginWith(List<String> list, Predicate<String> valid) {

List<String> selected = new ArrayList<>();

list.forEach(country -> { // lambda expression

if (valid.test(country)) {

selected.add(country);

}

});

return selected;

}

public static void main(String[] args) {

String[] countries = { "India", "Australia", "England", "South Africa", "Srilanka", "Newzealand",

"West Indies" };

List<String> countryList = Arrays.asList(countries);

System.out.println(getBeginWith(countryList, (s) -> s.startsWith("A")));

System.out.println(getBeginWith(countryList, (s) -> s.contains("W")));

System.out.println(getBeginWith(countryList, (s) -> s.endsWith("nd")));

}

}

Not Only this there are many Functional Interface are there

**54 remove @Fucntional Interface wt will happen**

It is to make sure that only one abstract method will be available in interface. It will restrict add one more abstract method to add in interface. If it is removed then multiple abstract methods can be added and it no longer support function interface and lamda expression cant be done through so it is imp for restrciting to interface only have one method

**55 Ways to create a object**

**56 how to fix concurrent modifaction exception ()**

**57 abstract class and interface as per java 8 whrn to use abstract and when use interface**

**58 method overlaoding and over riding mehtod hiding**

**59 encapsulation and Abstraction diff**

**60 HASESET and Hashmap**

**61 when to Transient variable**

The **Java transient** keyword is used on class attributes/**variables** to indicate that serialization process of such class should ignore such **variables** while creating a persistent byte stream for any instance of that class. A **transient variable** is a **variable** that can not be serialized.

class Employee implements Serializable

{

   private String           firstName;

   private String           lastName;

   private transient String confidentialInfo;

   //Setters and Getters

}

try

{

   ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("empInfo.ser"));

   Employee emp = new Employee();

   emp.setFirstName("Lokesh");

   emp.setLastName("Gupta");

   emp.setConfidentialInfo("password");

   //Serialize the object

   oos.writeObject(emp);

   oos.close();

} catch (Exception e)

{

   System.out.println(e);

}

try

{

   ObjectInputStream ooi = new ObjectInputStream(new FileInputStream("empInfo.ser"));

   //Read the object back

   Employee readEmpInfo = (Employee) ooi.readObject();

   System.out.println(readEmpInfo.getFirstName());

   System.out.println(readEmpInfo.getLastName());

   System.out.println(readEmpInfo.getConfidentialInfo());

   ooi.close();

} catch (Exception e)

{

   System.out.println(e);

}

private String           firstName;

private String           lastName;

//final field 1

public final transient String confidentialInfo = "password";

//final field 2

public final transient Logger logger = Logger.getLogger("demo");

marked the “confidentialInfo” to transient; and still the field was serialized. For similar declaration, logger was not serialized. Why?

Reason is that whenever any final field/reference is evaluated as “[constant expression](https://docs.oracle.com/javase/specs/jls/se7/html/jls-15.html" \l "jls-15.28" \o "constant expression" \t "https://howtodoinjava.com/java/keywords/transient-keyword-in-java-with-real-time-example/_blank)“, it is serialized by JVM ignoring the presence of transient keyword.

In above example, value “password” is a constant expression and instance of logger “demo” is reference. So by rule, confidentialInfo was persisted where as logger was not.

Are you thinking, what if I remove “transient” from both fields? Well, then fields implementing Serializable references will persist otherwise not. So, if you remove transient in above code, String (which implements Serializable) will be persisted; where as Logger (which does NOT implements Serializable) will not be persisted AND “java.io.NotSerializableException” will be thrown.

**62 when to go for join() method**

If current thread execution need to completed with out starting new one then use join

If t is a Thread object whose thread is currently executing, then t. **join**() will make sure that t is terminated before the next instruction is executed by the program. If there are multiple threads calling the **join**() **methods** that means overloading on **join** allows the programmer to specify a waiting period.

**63 excutoer service thread pool , excutign thread using excutators ways of excuterservice creating based on requirement, to check success excution , callable feature**

**64 synchronization methods and blocks of threads**

**65 synzchronize on ststic method**

**66 thread intercommuncation methods**

**67 ilegal state exception**

An **IllegalStateException**is an **unchecked exception**in Java. This exception may arise in our java program mostly if we are dealing with the collection framework of **java.util.package**. There are many collections like **List, Queue, Tree, Map**out of which **List**and **Queues**(Queue and Deque) to throw this **IllegalStateException**at particular conditions.

* An **IllegalStateExceptionexception**will be thrown, when we try to invoke a particular method at an inappropriate time.
* In case of**java.util.List** collection, we use **next()** method of the **ListIterator**interface to traverse through the **java.util.List.** If we call the **remove()** method of the **ListIterator**interface before calling the **next()**method then this exception will be thrown as it will leave the **List**collection in an **unstable state**.
* If we want to modify a particular object we will use the **set()**method of the **ListIterator**interface
* In the case of **queues**, if we try to add an element to a **Queue**, then we must ensure that the queue is not full. If this queue is full then we cannot add that element, then it will cause an **IllegalStateExceptionexception**to be thrown.

public class IllegalStateException {

public static void main(String[] args) {

List list = new LinkedList();

list.add("Welcome");

list.add("to");

list.add("Tutorials");

list.add("Point");

ListIterator lIterator = list.listIterator();

lIterator.next();

lIterator.remove();// modifying the list

lIterator.set("Tutorix");

System.out.println(list);

}

}

**68 wyas of creating thread**

**69 can we start a thread twice**

We can call thread start twice no compile time error but in run time 1st start will run 2nd start will give run time error java.lang.IllegalThreadStateException

**70 can we call run method directly**

**71 Thread local**

**78 Deadlock situtaion and analyzing fixing**

**79 Stream API internal functioning**

**80 Custom Exception class creation**

If a exception is checked or unchecked it always at occurs at run time only there is no chance of exception at compile time but checked exception make sure particular part is handled before it is handled so it will restrict during compile time to handle it. It will check if progammer handling or not

1. All **exceptions** must be a child of Throwable.
2. If you want to write a **checked exception** that is automatically enforced by the Handle or Declare Rule, you need to extend the **Exception** class.
3. If you want to write a runtime **exception**, you need to extend the RuntimeException class.

Throws method in method declaration is telling that some one to handle this method call who ever is calling that method need to handle it. Who ever is calling that method need to handle it.

Any checked exception has to be listed in throws part of creating method or try catch

If dnt want handle buy any one better use try catch in same method and handle it

Exception which come from RunTime exception or unchecked exception

Like its not like that we can use ArrayIndexBoundExcpetion has also throws but that is purrly programmal mistake but where has file not found is Envirnoment so need to be made checked exception

If the exception is recoverable - it should be checked. If the exception is not recoverable and the program must halt - it should be unchecked.

Checked exceptions are useful for recoverable cases where you want to provide information to the caller (i.e. insufficient permissions, file not found, etc). Unchecked exceptions are used rarely, if at all, for informing the user or programmer of serious errors or unexpected conditions during run-time.

class CustomCheckedExcpetion extends Exception {

private static final long serialVersionUID = -2695348327843261339L;

public CustomCheckedExcpetion(String msg) {

super(msg);

}

}

public class CustomeExceptionDemo {

private String name;

private int age;

public static boolean containsAlphabet(String name) {

for (int i = 0; i < name.length(); i++) {

char ch = name.charAt(i);

if (!(ch >= 'a' && ch <= 'z')) {

return false;

}

}

return true;

}

public CustomeExceptionDemo(String name, int age) throws CustomCheckedExcpetion {

if (!containsAlphabet(name) && name != null) {

String msg = "Improper name (Should contain only characters between a to z (all small))";

CustomCheckedExcpetion exName = new CustomCheckedExcpetion(msg);

throw exName;

}

this.name = name;

this.age = age;

}

public void display() {

System.out.println("Name of the Student: " + this.name);

System.out.println("Age of the Student: " + this.age);

}

public static void main(String args[]) throws CustomCheckedExcpetion {

Scanner sc = new Scanner(System.in);

System.out.println("Enter the name of the person: ");

String name = sc.next();

System.out.println("Enter the age of the person: ");

int age = sc.nextInt();

CustomeExceptionDemo obj = new CustomeExceptionDemo(name, age);

obj.display();

}

}

class CustomUnCheckedExcpetion extends RuntimeException {

private static final long serialVersionUID = -2695348327843261339L;

public CustomUnCheckedExcpetion(String msg) {

super(msg);

}

}

public class CustomeExceptionDemo2 {

private String name;

private int age;

public static boolean containsAlphabet(String name) {

for (int i = 0; i < name.length(); i++) {

char ch = name.charAt(i);

if (!(ch >= 'a' && ch <= 'z')) {

return false;

}

}

return true;

}

public CustomeExceptionDemo2(String name, int age) {

if (!containsAlphabet(name) && name != null) {

String msg = "Improper name (Should contain only characters between a to z (all small))";

CustomUnCheckedExcpetion exName = new CustomUnCheckedExcpetion(msg);

throw exName;

}

this.name = name;

this.age = age;

}

public void display() {

System.out.println("Name of the Student: " + this.name);

System.out.println("Age of the Student: " + this.age);

}

public static void main(String args[]) {

Scanner sc = new Scanner(System.in);

System.out.println("Enter the name of the person: ");

String name = sc.next();

System.out.println("Enter the age of the person: ");

int age = sc.nextInt();

CustomeExceptionDemo2 obj = new CustomeExceptionDemo2(name, age);

obj.display();

}

}

**81 Constructor chainaing**

**82 Double lock wr it is required**

This pattern reduces the number of lock acquisitions by simply checking the locking condition beforehand. As a result of this, there's usually a performance boost.

**83 2 thread created at same time excuting same time and accessing singlton object how many instance of singletoen will create**

**84 Linksed list internal working**

**85 can we insert null into TreeSet**

No we cant insert null to TreeSet it gives Null pointer exception in run time if it is inserted insert.Even if tis first or last it will throw error in run time

TreeSet implements of the Set interface that uses a tree for storage. Objects are stored in a sorted and ascending order.Access and retrieval times are quite fast, which makes TreeSet an excellent choice when storing large amounts of sorted information that must be found quickly.

The reason is, if you look at the internal implementation of the TreeSet, it uses natural ordering, that means TreeSet uses Comparable interface by default to sort its value by comparing other value.

TreeSet provides an implementation of the Set interface that uses a tree for storage. Objects are stored in a sorted and ascending order.TreeSet adds elements to it according to their natural order. This internally compares the elements with each other using the compareTo (or compare) method.

If you try to compare any object with a null value using one of these methods, a NullPointerException will be thrown.

Therefore, if you try to add null values to a TreeSet it generates a NullPointerException at the run time.

**86 Sleep and yield**

-**Sleep** : particular thread will sleep its execution for fixed number of time after the time it will continue its not gonna stop execution it will be in sleep state

- **join** : when ever join is applied to an thread it make sure after its completion only cpu can access another thread. Next thread wont be starting until the current thread part is completed. Total lock will be released after thread one is completed. We can set time to join method also

-**yield** : thread which is applied like this will pause for some time and it allows other thread to run it. If there is no thread available in the queue for further running then this thread will start its execution run

- **setPriority** : it for the thread on which priority it has to run , Default priority is 5 for all thread which can be rearranged. Its in descending order means priority one will be lowest of all. With high priority will execute first we can set priority in number base also, HIG\_PRIORITY=10, NOR\_PRIORITY=5,MIN\_PRIORITY=1

-**setName**: default when we use Thread.currentThread().getName() we will get default thread name assigned to it like 1st thread will be **Thead-0** and thread 2 will be **Thread-1** . we can set our own name for that thread using this method

**-setDemon** : we can set Demon thread it is helper thread to support some process . ex garbage collector is demon process it is low priority thread it will not have a life if dependent on user defined thread. If no thread running then no value fr demon thread

class SampleThread1 extends Thread {

@Override

public void run() {

for (int i = 0; i < 10; i++) {

System.err.println("Thread2 Name is for value:" + i + ":" + Thread.currentThread().getName() + "-ID-"

+ Thread.currentThread().getId() + "-Priority-" + Thread.currentThread().getPriority());

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}

class SampleThread2 extends Thread {

@Override

public void run() {

for (int i = 0; i < 10; i++) {

System.err.println("Thread2 Name is for value:" + i + ":" + Thread.currentThread().getName() + "-ID-"

+ Thread.currentThread().getId() + "-Priority-" + Thread.currentThread().getPriority());

}

}

}

class SampleThread3 extends Thread {

@Override

public void run() {

for (int i = 0; i < 10; i++) {

System.err.println("Thread2 Name is for value:" + i + ":" + Thread.currentThread().getName() + "-ID-"

+ Thread.currentThread().getId() + "-Priority-" + Thread.currentThread().getPriority());

}

}

}

public class ThreadDemo {

public static void main(String[] args) throws InterruptedException {

SampleThread1 thread1 = new SampleThread1();

SampleThread1 thread2 = new SampleThread1();

// thread1.start();

// thread1.join();

// thread2.start();

SampleThread2 thread3 = new SampleThread2();

SampleThread2 thread4 = new SampleThread2();

// thread3.start();

// thread3.yield();

// thread4.start();

SampleThread3 thread5 = new SampleThread3();

SampleThread3 thread6 = new SampleThread3();

thread5.setPriority(Thread.MAX\_PRIORITY);

thread5.setName("MYTHREAD1");

thread5.setDaemon(true);

thread5.start();

// thread5.start();

thread6.start();

// thread6.start();

SampleThread3 thread7 = new SampleThread3();

SampleThread3 thread8 = new SampleThread3();

thread7.setDaemon(true);

thread7.start();

// thread8.start();

}

}

**87 Executor service**

[ExecutorService](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorService.html) is a framework provided by the JDK which simplifies the execution of tasks in asynchronous mode. Generally speaking, ExecutorService automatically provides a pool of threads and API for assigning tasks to it.

**88 throw and throws**

**89 HASHSET how it will eliminate duplciate (internal it uses hashmap)**

**HASHset works on simialr principle of hashing intrenally like Hashmap HashSet** works with equals() and hashCode() method to **check for duplicate** element when you try to add an element.

I have overrided hashcode() and equals() method. When the **duplicate object** is encountered, the hashcode value will be same, then equals method is executed. If both **objects** are equal then it returns true and **object** will not be added to **HashSet**.

**90 Array index bound of exception**

Thrown to indicate that an array has been accessed with an illegal index. The index is either negative or greater than or equal to the size of the array.