What are the main features of OOPs?

The main feature of the OOPs, also known as 4 pillars or basic principles of OOPs are as follows:

Encapsulation

Data Abstraction

Polymorphism

Inheritance

<b>What is Encapsulation</b>

Data hiding: A language feature to restrict access to members of an object. For example, private and protected members in C++.

Bundling of data and methods together: Data and methods that operate on that data are bundled together. For example,

the data members and member methods that operate on them are wrapped into a single unit known as a class.

<b>What is Abstraction? </b>

Abstraction is similar to data encapsulation and is very important in OOP. It means showing only the necessary information and hiding the other irrelevant information from the user. Abstraction is implemented using classes and interfaces.

What is Inheritance? What is its purpose?

The idea of inheritance is simple, a class is derived from another class and uses data and implementation of that other class. The class which is derived is called child or derived or subclass and the class from which the child class is derived is called parent or base or superclass.

The main purpose of Inheritance is to increase code reusability. It is also used to achieve Runtime Polymorphism.

What is Polymorphism? and types of Polymorphism?

The word "Polymorphism" means having many forms. It is the property of some code to behave differently for different contexts. For example, in C++ language, we can define multiple functions having the same name but different working depending on the context.

Polymorphism can be classified into two types based on the time when the call to the object or function is resolved. They are as follows:

Compile Time Polymorphism

Runtime Polymorphism

A) Compile-Time Polymorphism

Compile time polymorphism, also known as static polymorphism or early binding is the type of polymorphism where the binding of the call to its code is done at the compile time. Method overloading or operator overloading are examples of compile-time polymorphism.

B) Runtime Polymorphism

Also known as dynamic polymorphism or late binding, runtime polymorphism is the type of polymorphism where the actual implementation of the function is determined during the runtime or execution. Function overriding is an example of this method.

What different types of Inheritance are there

Single Inheritance: Child class derived directly from the base class

Multiple Inheritance: Child class derived from multiple base classes.

Multilevel Inheritance: Child class derived from the class which is also derived from another base class.

Hierarchical Inheritance: Multiple child classes derived from a single base class.

Hybrid Inheritance: Inheritance consisting of multiple inheritance types of the above specified.

Java does not support multiple inheritance.

How is an abstract class different from an interface?

Both abstract classes and interfaces are special types of classes that just include the declaration of the methods, not their implementation.

Abstract Class

A class that is abstract can have both abstract and non-abstract methods.

An abstract class can have final, non-final, static and non-static variables.

Abstract class doesn't support multiple inheritance

Interface

An interface can only have abstract methods.

The interface has only static and final variables.

An interface supports multiple inheritance.

<b>Lifecycle and States of a Thread in Java</b>

New State

Runnable State

Blocked State

Waiting State

Timed Waiting State

Terminated State

Life Cycle of a Thread

New Thread: When a new thread is created, it is in the new state. The thread has not yet started to run when the thread is in this state. When a thread lies in the new state, its code is yet to be run and has not started to execute.

Runnable State: A thread that is ready to run is moved to a runnable state.

It is the responsibility of the thread scheduler to give the thread, time to run.

Each and every thread get a small amount of time to run. After running for a while,

a thread pauses and gives up the CPU so that other threads can run.

Blocked: The thread will be in blocked state when it is trying to acquire a lock

The thread will move from the blocked state to runnable state when it acquires the lock.

Waiting state: The thread will be in waiting state when it calls wait() method or join() method. It will move to the runnable state when other thread will notify or that thread will be terminated.

Timed Waiting: A thread lies in a timed waiting state when it calls a method with a time-out parameter.

Terminated State: A thread terminates

Because it exits normally. This happens when the code of the thread has been entirely executed by the program.

Because there occurred some unusual erroneous event, like a segmentation fault or an unhandled exception.

Marker Interface in Java

In Java, a **marker Interface is an empty interface that has no fields or methods.** It is used just to mark or tag a class to tell Java or other programs something special about that class. These interfaces do not have any methods inside but act as metadata to provide information about the class.

Examples of marker interfaces include **Serializable, Cloneable, and Remote.**

Java Functional Interfaces

A **functional interface in Java** is an interface that contains only one abstract method. Functional interfaces can have multiple **default** or **static** methods, but **only one abstract method**. **Runnable**, **ActionListener**, and **Comparator** are common examples of **Java functional interfaces**.

Collections

ArrayList vs LinkedList

ArrayList

ArrayList **internally uses a dynamic array to store its elements.**

The elements are stored in contiguous memory locations.

When it comes to manipulation, *ArrayList* **can be slower.**

When you remove an element from an *ArrayList*, all subsequent elements must be shifted in memory to maintain the contiguous structure, making operations like deletion or insertion at arbitrary positions expensive.

ArrayList consumes less memory compared to *LinkedList* because it only stores the data itself in a single block of memory.

ArrayList is better for storing and accessing data

LinkedList

LinkedList **internally uses a doubly linked list.** Each element is stored in a node, and each node contains a reference to both the next and the previous element in the list. This allows easier insertion and removal of elements, but the elements are not stored in contiguous memory.

LinkedList **is faster** for manipulation operations like adding or removing elements, especially in the middle of the list.

There's no need to shift elements in memory.

LinkedList, on the other hand, consumes more memory. This is because each element is stored in a node, and each node contains two additional references — one for the next element and one for the previous element.

LinkedList is better for manipulating data

Java HashMap

A `HashMap` stores items in **key/value pairs**, where each key maps to a specific value.

It is part of the `java.util` package and implements the `Map` interface.

Instead of accessing elements by an index (like with ArrayList), we can use a **key** to retrieve its associated **value**.

Internal Working of HashMap in Java

Hashing in HashMap

Hashing is the process of converting an object into an integer by using the hashCode() method. It's necessary to write the hashCode() method properly for better performance of the HashMap. HashMap uses the **hashCode()** method to determine the bucket location for a key.

override hashCode() method returns the first character's ASCII value as hash code. So, whenever the first character of the key is same, the hash code will be the same.

HashMap also allows a null key, so hash code of null will always be 0.

The hashCode() method is used to get the hash code of an object. hashCode() method of the object class returns the memory reference of an object in integer form.

In HashMap, hashCode() is used to calculate the bucket and therefore calculate the index.

equals() Method

The equals() method is used to check whether 2 objects are equal or not. This method is provided by the Object class. You can override this in your class to provide your implementation.
HashMap uses equals() to compare the key to whether they are equal or not. If the equals() method return true, they are equal otherwise not equal.

Buckets

A bucket is an element of the HashMap array. It is used to store nodes. Two or more nodes can have the same bucket. In that case, a link list structure is used to connect the nodes. Buckets are different in capacity.

Load Factor and Capacity:
*capacity = number of buckets * load factor*
A single bucket can have more than one node, it depends on the hashCode() method.

Internal Working of put() Method in HashMap

When we insert a key-value pair into a HashMap using the put() method

map.put(new Key("vishal"), 20);

Steps:

- Calculate hash code of Key {"vishal"}. It will be generated as 118.
- Calculate index by using index method it will be 6.
- Create a node object as:

Inserting Second Key-Value Pair:
Now, putting the other pair that is,
map.put(new Key("sachin"), 30);

Steps:

- Calculate hashCode of Key {"sachin"}. It will be generated as 115.
- Calculate index by using index method it will be 3.
- Create a node object as:

nserting Third Key-Value Pair (Collision Example):
Now, putting another pair that is,
map.put(new Key("vaibhav"), 40);

Steps:

- Calculate hash code of Key {"vaibhav"}. It will be generated as 118.
- Calculate index by using index method it will be 6.
- Create a node object as:

Place this object at index 6 if no other object is presented there. In this case, a node object is found at index 6 - this is a case of collision. In that case, check via the hashCode() and equals() method if both the keys are the same. If keys are the same, replace the value with the current value. Otherwise, connect this node object to the previous node object via linked list and both are stored at index 6.
Now HashMap becomes:

Java Set Interface

The Set interface is part of the Java Collections Framework and is used to store a collection of **unique elements**.

Common classes that implement Set:

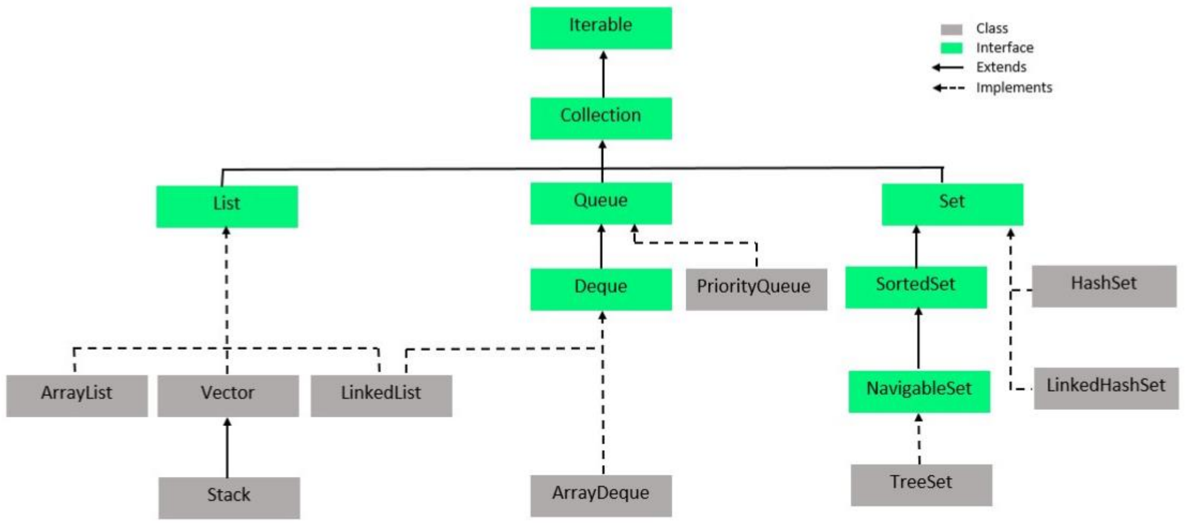
- HashSet - fast and unordered
- TreeSet - sorted set
- LinkedHashSet - ordered by insertion

```
var set = Map.of("1", "1", "2", "2", "3", "3", "4", "4", "5", "5");
    String three = set.get(new String("3"));
    Assertions.assertSame("3", three);
```

Set vs. List

List	Set
Allows duplicates	Does not allow duplicates
Maintains order	Does not guarantee order
Access by index	No index-based access

Hierarchy of Java Collections



ConcurrentModificationException in Java

java.util.ConcurrentModificationException is a very common exception when working with Java collection classes.

Collection will be changed while traversing over it using iterator, the `iterator.next()` will throw **ConcurrentModificationException**.

```
Iterator<String> it = myList.iterator();
                while (it.hasNext()) {
                        String value = it.next();
                        System.out.println("List Value:" + value);
                        if (value.equals("3"))
                                myList.remove(value);
                }
```

To Avoid ConcurrentModificationException

we can use **ConcurrentHashMap** and **CopyOnWriteArrayList** classes. This is the recommended approach to avoid concurrent modification exception.

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

                myList.add("1");
                myList.add("2");
Iterator<String> it = myList.iterator();
                while (it.hasNext()) {
                        String value = it.next();
                        System.out.println("List Value:" + value);
                        if (value.equals("1")) {
                                myList.remove("2");
                                myList.add("6");
                                myList.add("7");
                        }
                }
```

```
Map<String, String> myMap = new ConcurrentHashMap<String, String>();
                myMap.put("1", "1");
                myMap.put("2", "2");
                myMap.put("3", "3");

                Iterator<String> it1 = myMap.keySet().iterator();
                while (it1.hasNext()) {
                        String key = it1.next();
                        System.out.println("Map Value:" + myMap.get(key));
                        if (key.equals("1")) {
                                myMap.remove("3");
                                myMap.put("4", "4");
                                myMap.put("5", "5");
                        }
                }
```