**Collection**

What is collection

A screenshot of a computer

AI-generated content may be incorrect.

A collection is simply an object that represents an group of objects, know as its elements.

Collections is a utility class which provides static methods.

Collection is a parent interface for Set,List,Queue.

Iterable is a parent interface for Collection, List, Set, Queue. By using iterable interface we can use for each loop, so this interface is extend by every interface which is mentioned so we can use for each loop. (Iterable has for each loop method defined)

What is collection framework

It provides a set of interfaces and classes that help in managing group of objects.

Before the introduction of the collection framework in JDK 1.2, JDK 1.0 java used to rely on a variety of classes like Vector, Stack,HashTable and arrays to store and manipulate groups of objects. However these classes have several drawbacks.

* Inconsistency: Each class had a different way of managing collections, leading to confusion and steep learning curve.
* Lack of inter-operability: These classes were not designed to work together seamlessly.
* No common interface: There was no common interface for all these classes, which meant you couldn’t write generic algorithms that could operate on different types of collections.

To solve this problem, the collection framework was introduced.

* Unified architecture: A consistent set of interfaces for all collections.
* Inter-operability: Collections can be easily interchanged and manipulated in a uniform way.
* Reusability: Generic algorithms can be written that work with any collection.
* Efficiency: The framework provides efficient algorithms for basic operations like searching, sorting and manipulation.

Key Interfaces in the collection framework

The collection framework is primarily built around a set of interfaces.

* Collection: The root interfaces for all the other collection types.
* List: An ordered collection that can contain duplicate elements (eg: Array list, LinkedList)
* Set: A collection that cannot contain duplicate elements (eg: HashSet, Linked HashSet)
* Map: An interface that represents a collection of key value pairs (eg: HashMap, Linked HashMap)

List

The List interface in Java is a part of the java.util package and is a sub-interface of the Collection interface. It provides a way to store an ordered collection of elements (known as a sequence). Lists allow for precise control over where elements are inserted and can contain duplicate elements.

The List interface is implemented by several classes in the Java Collection Framework, such as *ArrayList*, *LinkedList*, *Vector*, and *Stack*.

**Key Features of the List Interface**

* **Order Preservation**
* **Index-Based Access**
* **Allows Duplicates**

**Array List Class**

An ArrayList is a resizable array implementation of the List interface. Unlike arrays in Java, which have a fixed size, an *ArrayList can change its size dynamically* as elements are added or removed. This flexibility makes it a popular choice when the number of elements in a list isn't known in advance.

Unlike a regular array, which has a fixed size, an ArrayList can grow and shrink as elements are added or removed. This dynamic resizing is achieved by creating a new array when the current array is full and copying the elements to the new array.

Internally, the ArrayList is implemented as an array of Object references. When you add elements to an ArrayList, you're essentially storing these elements in this internal array.

When you create an ArrayList, it has an initial capacity (default is 10). The capacity refers to the size of the internal array that can hold elements before needing to resize.

**Adding Elements**

When we add an element to an ArrayList, the following steps occur

Check Capacity: Before adding the new element, ArrayList checks if there is enough space in the internal array (elementData). If the array is full, it needs to be resized.

Resize if Necessary: If the internal array is full, the ArrayList will create a new array with a larger capacity (usually 1.5 times the current capacity) and copy the elements from the old array to the new array.

Add the Element: The new element is then added to the internal array at the appropriate index, and the size is incremented.

**Resizing the Array**

* Initial Capacity: By default, the initial capacity is 10. This means the internal array can hold 10 elements before it needs to grow.
* Growth Factor: When the internal array is full, a new array is created with a size 1.5 times the old array. This growth factor balances memory efficiency and resizing cost.
* Copying Elements: When resizing occurs, all elements from the old array are copied to the new array, which is an O(n) operation, where n is the number of elements in the ArrayList.

**Removing Elements**

* Check Bounds: The ArrayList first checks if the index is within the valid range.
* Remove the Element: The element is removed, and all elements to the right of the removed element are shifted one position to the left to fill the gap.
* Reduce Size: The size is decremented by 1.

**Declaring ArrayList**

* 1. **List<String> list = Arrays.asList(array);**
* **What it does**:  
  Creates a fixed-size list backed by the given array.
* **Key points**:
  + **Changes to the array** are reflected in the list and vice versa.
  + **Size is fixed** → you cannot add or remove elements (add() / remove() will throw UnsupportedOperationException).
  + You can **replace** elements (set(index, value) works).
  1. **List<String> list = new ArrayList<>();**
* **What it does**:  
  Creates a **mutable**, growable, and resizable list.
* **Key points**:
  + Fully independent — not backed by any array.
  + Can freely add(), remove(), set(), clear(), etc.
  + Most commonly used in real-world code for flexibility.
  1. **List<String> list = List.of(); *(Java 9+)***
* **What it does**:  
  Creates an **immutable** list with given elements.
* **Key points**:
  + Size is fixed and elements **cannot be changed**.
  + Both add(), remove() and set() will throw UnsupportedOperationException.
  + Null elements **not allowed** → NullPointerException if you try.

**Key Points**

If you want complete flexibility, go with new ArrayList<>().  
If you want a fixed-size view of an existing array, use Arrays.asList().  
If you want a read-only list, use List.of().

**Different ways to add elements in ArrayList**

// Different ways to add elements  
ArrayList<Integer> list=new ArrayList<>();  
 ArrayList<Integer> list1=new ArrayList<>();  
 list1.add(1);  
 list1.add(2);  
   
 // normal way  
 list.add(23);  
 // adding using index  
 list.add(1,22);  
 // adding all elements once  
 list.addAll(list1);  
 // adding all elements once using index  
 list.addAll(0,list1);

**Accessing element from Array List**

ArrayList<Integer> list1=new ArrayList<>();  
 list1.add(1);  
 list1.add(2);

list1.get(0);

**Modifying element from Array List**

ArrayList<Integer> list1=new ArrayList<>();  
 list1.add(1);  
 list1.add(2);

list1.set(0,22);

**Removing the elements from ArrayList**

ArrayList<Integer> list1=new ArrayList<>();  
 list1.add(1);  
 list1.add(2);

list.remove(1); // removes the value 2 because its using index

list.remove(Integer.valueOf(1)); // here we are wrapping the class its takes as a object and remove the value 1.

**Converting Arraylist into Array**

ArrayList<Integer> list1 = new ArrayList<>();  
list1.add(1);  
list1.add(2);  
/\*  
Converting arraylist into array just need to declare 0 if the arraylist  
has any size  
 \*/  
Integer[] array = list1.toArray(new Integer[0]);  
  
for (int num:array){  
 System.*out*.println(num);  
}

**Sorting**

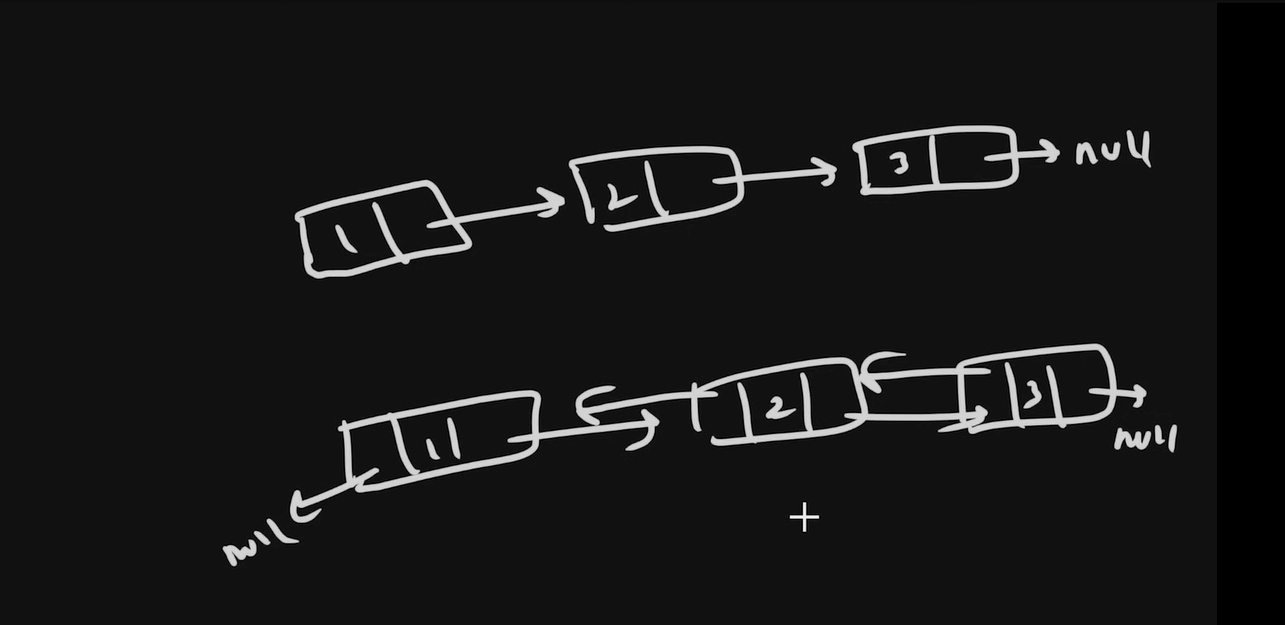
// Sorting  
ArrayList<Integer> list1 = new ArrayList<>();  
list1.add(11);  
list1.add(2);  
// Ascending order  
Collections.*sort*(list1);  
System.*out*.println(list1);  
// descending order  
Collections.*sort*(list1, Collections.*reverseOrder*());  
System.*out*.println(list1);

**Linked List**

The LinkedList class in Java is a part of the Collection framework and implements the List interface. Unlike an ArrayList, which uses a dynamic array to store the elements, a LinkedList stores its elements as nodes in a doubly linked list. This provides different performance characteristics and usage scenarios compared to ArrayList.

A LinkedList is a linear data structure where each element is a separate object called a node. Each node contains two parts:

* Data: The value stored in the node.
* Pointers: Two pointers, one pointing to the next node (next) and the other pointing to the previous node (previous).

****

**The first image is diagram of single linked list or (Linked list) shows how data is strore using pointers. In first block data one is stored and showing the reference of 2 block data, means in first block data will store and address of 2nd data will be there at the end the address is null because there is no other data is there.**

**The second image shows double linked list in that case the first block data is there and address of second block data is there, in second block data is there and first block address is there.**

Performance Considerations

LinkedList has different performance characteristics compared to ArrayList:

* Insertions and Deletions: LinkedList is better for frequent insertions and deletions in the middle of the list because it doesn't require shifting elements, unlike ArrayList.
* Random Access: LinkedList has slower random access (get(int index)) compared to ArrayList because it has to traverse the list from the beginning to reach the desired index.
* Memory Overhead: LinkedList requires more memory than ArrayList because each node in a linked list requires extra memory to store references to the next and previous nodes.

**🔹 Vector (Java Collection Framework)**

* Vector is a **legacy class** in Java (introduced in JDK 1.0).
* It implements **List interface** and is similar to ArrayList.
* The **main difference** is that **Vector is synchronized**, which makes it **thread-safe** but slower compared to ArrayList.
* It stores elements in a **dynamic array** and maintains **insertion order**.

Why not use Vector?

"Vector is synchronized but due to performance overhead it's rarely used now. Instead, we use ArrayList with Collections.synchronizedList() or CopyOnWriteArrayList when thread safety is required."

Declaration of Vector

Vector<Integer> vector=new Vector<>();

**🔹 Stack (Java Collection Framework)**

* Stack is a **subclass of Vector** (introduced in JDK 1.0).
* It follows the **LIFO (Last In First Out)** principle.
* It has special methods like:
  + push() → Add element on top
  + pop() → Remove element from top
  + peek() → See top element without removing
* It is **synchronized** (inherited from Vector).

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

**CopyOnWriteArrayList**

CopyOnWriteArrayList<Integer> list = new CopyOnWriteArrayList<>();

// "Copy on Write" means that whenever a write operation

// like adding or removing an element

// instead of directly modifying the existing list

// a new copy of the list is created, and the modification is applied to that copy

// This ensures that other threads reading the list while it’s being modified are unaffected.

// Read Operations: Fast and direct, since they happen on a stable list without interference from modifications.

// Write Operations: A new copy of the list is created for every modification.

// The reference to the list is then updated so that subsequent reads use this new list.

**🔹 ArrayList vs LinkedList vs Vector vs Stack**

| **Feature** | **ArrayList** | **LinkedList** | **Vector** | **Stack** |
| --- | --- | --- | --- | --- |
| **Data Structure** | Dynamic Array | Doubly Linked List | Dynamic Array | Inherits Vector (Dynamic Array) |
| **Order** | Maintains insertion order | Maintains insertion order | Maintains insertion order | Maintains insertion order |
| **Access (get by index)** | **Fast (O(1))** | **Slow (O(n))** | **Fast (O(1))** | **Fast (O(1))** |
| **Insertion/Deletion** | Slow (O(n), needs shifting) | Fast (O(1) if node ref known) | Slow (O(n), needs shifting) | Slow (O(n), shifting) |
| **Thread Safety** | ❌ Not synchronized | ❌ Not synchronized | ✅ Synchronized | ✅ Synchronized |
| **Use Case** | Frequent access/search | Frequent insert/delete | Legacy thread-safe ArrayList | Legacy LIFO (stack operations) |
| **Special Feature** | Fast random access | Better for manipulating middle elements | Synchronized version of ArrayList | Provides **LIFO** methods (push, pop, peek) |
| **Introduced** | JDK 1.2 | JDK 1.2 | JDK 1.0 (Legacy) | JDK 1.0 (Legacy, but built on Vector) |

**MAP Interface**

In Java, a Map is an object that maps keys to values. It cannot contain duplicate keys, and each key can map to at most one value. *Think of it as a dictionary* where you look up a word (key) to find its definition (value)

**Key Characteristics of the Map Interface**

* **Key-Value Pairs:** Each entry in a Map consists of a key and a value.
* **Unique Keys:** No two entries can have the same key.
* **One Value per Key:** Each key maps to a single value.
* **Order:** Some implementations maintain insertion order (LinkedHashMap), natural order (TreeMap), or no order (HashMap).

HashMap

Key Characteristics

**Unordered:** Does not maintain any order of its elements.

**Allows null Keys and Values:** Can have one null key and multiple null values.

**Not Synchronized:** Not thread-safe; requires external synchronization if used in a multi-threaded context.

**Performance:** Offers constant-time performance (O(1)) for basic operations like get and put, assuming the hash function disperses elements properly.

A diagram of a diagram

AI-generated content may be incorrect.

Value: The data associated with the key

Bucket: A place where key and value are stored. Think of buckets as cells in a list (called Array)

Hash Function: Converts a key into an index (bucket location) for storage.

**Internal Working of HashMap**

**A hash function is an algorithm that takes an input (or "key") and returns a fixed-size string of bytes, typically a numerical value. The output is known as a hash code, hash value, or simply hash.  
*The primary purpose of a hash function is to map data of arbitrary size to data of fixed size***

**How Data is Stored in HashMap**

Step 1: Hashing the Key  
First, the key is passed through a hash function to generate a unique hash code (an integer number). This hash code helps determine where the key-value pair will be stored in the array (called a "bucket array").

Step 2: Calculating the Index  
The hash code is then used to calculate an index in the array (bucket location) using

int index = hashCode % arraySize;

The index decides which bucket will hold this key-value pair.

For example, if the array size is 16, the key’s hash code will be divided by 16, and the remainder will be the index.

A black background with white text

AI-generated content may be incorrect.

map.put("apple", 50);

* "apple" is the key.
* 50 is the value.
* The hash code of "apple" is calculated.
* The index is found using the hash code.
* The pair ("apple", 50) is stored in the corresponding bucket.

**How HashMap Retrieves Data**

* When we call get(key), the HashMap follows these steps:
* **Hashing the Key:** Similar to insertion, the key is hashed using the same hash function to calculate its hash code.
* **Finding the Index:** The hash code is used to find the index of the bucket where the key-value pair is stored.
* **Searching in the Bucket:** Once the correct bucket is found, it checks for the key in that bucket. If it finds the key, it returns the associated value.

A blackboard with white text and orange arrows

AI-generated content may be incorrect.

A blackboard with white chalk and a drawing

AI-generated content may be incorrect.

**Handling Collisions**

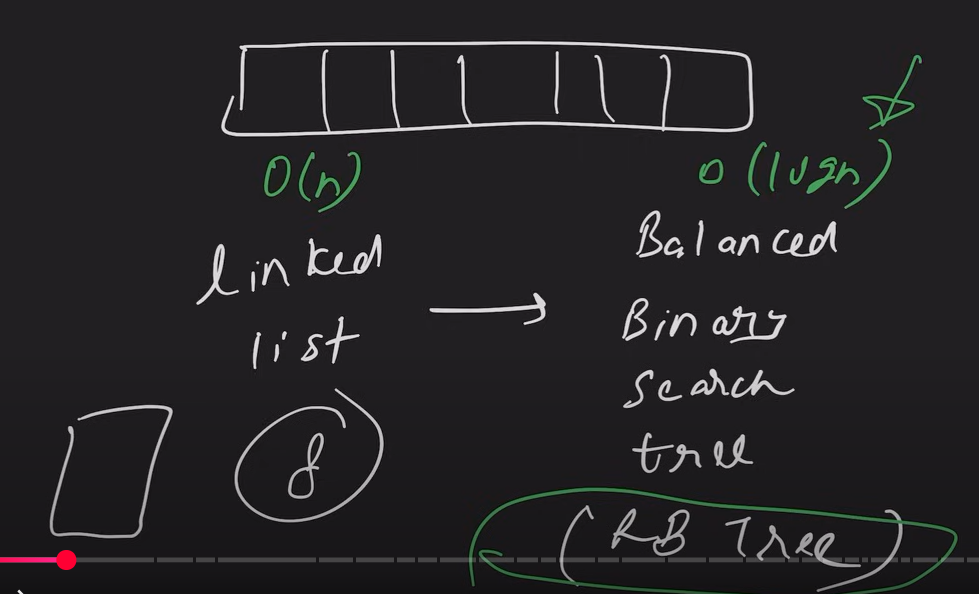
Since different keys can generate the same index (called a collision), HashMap uses a technique to handle this situation. Java's HashMap uses Linked Lists (or balanced trees after Java 8) for this.

If multiple key-value pairs map to the same bucket, they are stored in a linked list inside the bucket.

When a key-value pair is retrieved, the HashMap traverses the linked list, checking each key until it finds a match.

A blackboard with white text and arrows

AI-generated content may be incorrect.



A screenshot of a computer code

AI-generated content may be incorrect.

A black and white text on a black background

AI-generated content may be incorrect.

A black and white text on a black background

AI-generated content may be incorrect.

A black and white text with white text

AI-generated content may be incorrect.

A black background with blue and yellow text

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer screen

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

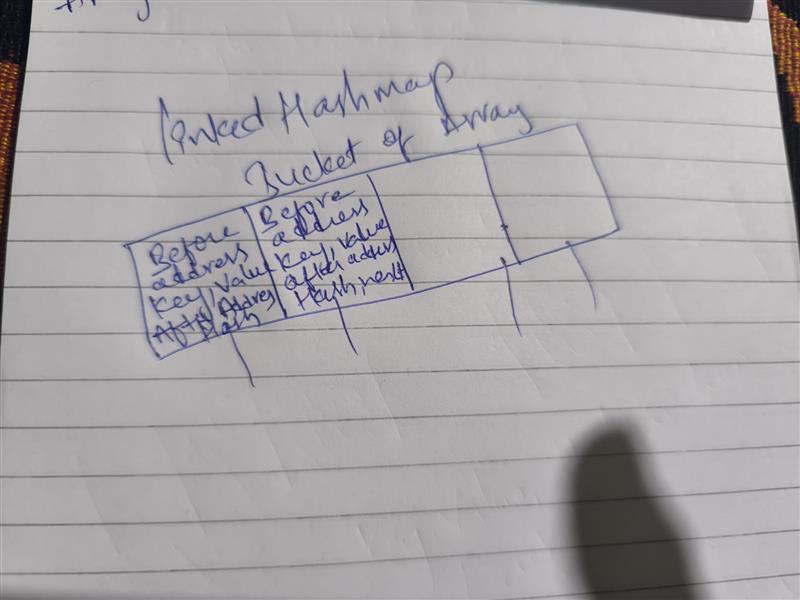
**Linked HashMap**

The internal storing is same a hashmap but extra used doubly linkedlist to maintain insertion order.

LinkedHashmap extends Hashmap and implements Map

All are same as hashmap but linkedhashmap maintain insertion order using doubly linkedlist

Internal storing of LinkedHashMap



**SET**

Hash Set works internally same as HashMap the only different is in hash set the value is stored as dummy value because set doesn’t have key and value. The data what we are giving in hashset stores as key.

**How it works in memory**

* Keys of the HashMap = elements of the HashSet.
* Value of every entry = the same dummy object (PRESENT).
* Buckets, hashing, resizing, load factor, etc., all work **exactly the same as HashMap**.

Linked HashSet works internally same as LinkedHashMap

**1. Internal Implementation**

* **HashSet** → Internally backed by a **HashMap**.
* **LinkedHashSet** → Internally backed by a **LinkedHashMap**.

**🔹 2. Ordering**

* **HashSet** → Does **not** maintain insertion order; elements appear in random/unpredictable order.
* **LinkedHashSet** → Maintains **insertion order** (elements appear in the same order they were inserted).

**🔹 3. Performance**

* Both have:
  + **add, remove, contains** → O(1) average time.
  + But LinkedHashSet has a **slightly higher overhead** (because it maintains a linked list for order).

**🔹 4. Memory Usage**

* **HashSet** → Less memory (just HashMap buckets).
* **LinkedHashSet** → More memory (extra before and after references in each entry to maintain the doubly linked list).

**🔹 5. Use Case**

* **HashSet** → Use when you only care about **uniqueness** of elements, not the order.
* **LinkedHashSet** → Use when you care about **uniqueness + insertion order**.

**OOPS Concepts (Object Oriented Programming System)**

Four pillars of oops

Encapsulation

Inheritance

Polymorphism

Abstraction

OOPs is a **programming paradigm (way of writing programs)** based on the concept of **objects**, which represent **real-world entities**.

* Objects = data (fields/variables) + behavior (methods).
* It makes software more **modular, reusable, maintainable, and scalable**.

**(paradigm** means mindset)

**Advantages of OOPs**

* Code **reusability** (inheritance).
* Code **security** (encapsulation).
* **Flexibility** and **scalability**.
* Easier to **model real-world problems**.

**What is a Method in Java?**

A **method** in Java is a **block of code that performs a specific task**.

**Why are Methods Used in Java? (Important Points)**

1. **Code Reusability**
   * Write once, use many times.
   * Instead of writing the same logic again, you can call the method wherever needed.
2. **Modularity (Divide & Conquer)**
   * Breaks complex problems into smaller pieces (methods).
   * Each method handles one specific task → makes code easier to understand.
3. **Readability & Maintainability**
   * Methods give meaningful names → code becomes more readable.
   * Easier to debug and modify because logic is separated.
4. **Avoids Code Duplication**
   * Common logic (like login, calculation, validation) can be written once and reused across the program.
5. **Testing & Debugging**
   * Individual methods can be tested independently.
   * Helps in writing unit tests easily.
6. **Encapsulation & Abstraction**
   * Hides internal implementation → you just call the method without worrying about how it works.
   * Example: System.out.println() is a method → you don’t need to know its internal code.
7. **Better Collaboration**
   * In large projects, different developers can work on different methods independently.
8. **Supports OOP (Object-Oriented Programming)**
   * In Java, methods often belong to objects (instance methods) → they define behavior of objects.

**1. Encapsulation**

Encapsulation is the process of hiding data by making variables private and exposing controlled access through public methods, ensuring data security, flexibility, and maintainability.

Data Hiding

* Variables are declared private so they cannot be accessed directly from outside the class.

Controlled Access

* Access to variables is provided via public methods (getters, setters, or custom methods).
* This ensures only valid operations happen on data.

Security

* Prevents unauthorized or invalid access to internal data.
* Example: you cannot set balance = -1000 in a BankAccount.

Flexibility

* Internal implementation can change without affecting outside code.
* Example: If you change how marks are calculated, outside classes don’t need changes.

Improved Maintainability

* Code is modular and easy to debug or extend.

Achieved using Classes

* Encapsulation is implemented through private variables + public methods.

Not only getters/setters

* Encapsulation can also be achieved with custom methods that control logic, not just with getters and setters.

Conclusion:

* Getters and Setters are not mandatory.
* What is mandatory: making variables private and controlling access via methods.
* You can use either getters/setters or custom logic methods to achieve encapsulation.

**4.Abstraction**

**Hiding the implementation details and showing the functionality is known as abstraction.**

Abstract class is declared with the abstract keyword.

It may have both abstract & non-abstract methods (methods with bodies).

An abstract is a Java modifier applicable for class & methods in Java but not for variables.

Java abstract class is a class that cannot be instantiated by itself;

it needs to be subclassed by another class to use its properties.

An abstract class is declared using the "abstract" keyword.

Abstract have instance variables so we can create constructor for abstract class

A class can extend only one abstract class

1) An instance of an abstract class cannot be created.

2) Normal constructor is allowed but a constructor with abstract keyword is not allowed.

3) We can have an abstract class without any abstract method.

4) We can have final method but not an abstract final method.

5) We can define static methods in an abstract class.

If a class contains at least one abstract method then compulsory we should declare a class as abstract.

(Methods: abstract, variables)

Methods (public, private, static, final, instance, protected, local, private, final)

🔑 Interview line

“No, we cannot declare variables as abstract in Java because variables do not have implementation — only methods can be abstract.”

**Interface**

Class is a blueprint of object

Interface is a blueprint of class

By using interface, we can achieve multiple inheritance and 100% abstraction

Interface have abstract methods and static constants

After Java 8 interface has static and default methods

Interface doesn’t have instance variables so we cant create constructor for interface

A class can implements multiple interfaces

static void m2(){  
 System.*out*.println("I am static method");  
}  
  
default void m3(){  
 System.*out*.println("I am default method");  
}

After Java 9 private methods are introduced in interfaces

private void m4(){  
 System.*out*.println("I am private ");  
}

**RECURSION**

If a method called itself is known as recursion.

In Recursion the main method called the method until the condition is met, if we don’t write any condition to break the recursion it will called infinite times and throws StackOverFlowError (stack is a memory in java if the method wont stop the memory will full and exception come).

public static void main(String[] args) {  
 Test2 t=new Test2();  
 int num= t.factorial(5);  
 System.*out*.println(num);  
}  
  
public int factorial(int n){  
 if(n==1){  
 return 1;  
 }  
 return n\*factorial(n-1);  
  
}

**What are Packages in Java?**

A **package** in Java is just a **folder (namespace)** that groups related **classes, interfaces, and sub-packages** together.

Think of it like organizing files in your computer:

* Instead of keeping all files on the Desktop, you create folders like **Movies**, **Photos**, **Docs**.
* Similarly, in Java you don’t keep all classes in one place → you organize them into **packages**.

Types of Packages

**Built-in packages** (already provided by Java):

* java.util → for collections, Scanner, Date

**User-defined packages** (you create them yourself).

**Why use Packages?**

* **Avoid name conflicts** (two classes with the same name but in different packages).
* **Code reusability** (can import and use again in other projects).
* **Maintainability** (organized code, easy to manage large projects).
* **Access protection** (with access modifiers + package-level visibility).

**Operator Precedence in Java (Simplified)**

1. **Brackets ( )**
2. **Unary operators** (++, --, !)
3. **Multiplication, Division, Modulus** (\*, /, %)
4. **Addition, Subtraction** (+, -)
5. **Relational Operators** (<, >, <=, >=)
6. **Equality Operators** (==, !=)
7. **Logical AND, OR** (&&, ||)
8. **Assignment** (=, +=, -= etc.)