Java notes

In java if you don't initialize the variables with the assignment then they give the error
Javac won't compile if there is any error or any potentially dangerous warning

In **Java**, class-level (instance/static) variables **do** get default values:

```
class Test {
   int a;          // instance variable → defaults to 0
   static int b;   // static variable → defaults to 0
}
```

In C++ in case if it is written it will get a default NULL value and start running , not in the case of Java it will give error

Difference b/w METHOD & CLASS

Class is something that is used in oops basically that describes what and all will it contain.
Method is basically functions (C++ ke functions ko Java me methods bolte hain) **Something that will call a piece of reusable code when used with ();**

STATIC Methods in Java

→ makes a variable or a method THAT BELONGS TO A PARTICULAR CLASS RATHER THAN TO ANY SPECIFIC OBJECT . USED FOR UTILITY METHODS OR SHARED OBJECTS.

1) What does static mean in Java?
static marks a class-level member.
That means the field or method belongs to the class itself, not to any particular object (instance) of that class.
There is exactly one copy of a static field per class, shared by all instances.
static members are loaded/initialized when the class is loaded by the JVM (class initialization time), not when a method runs or an object is created.
2) Where different things live (memory model, conceptually)
Static fields → stored in the class area (often called "method area" or "permgen/metaspace" depending on JVM). They exist for the lifetime of the loaded class.

Instance fields → stored in objects on the heap. Each object has its own copy.

Local variables (variables declared inside methods) → live on the call stack (stack frames) while the method executes. They disappear when the method returns.

So: static → class area (long-lived). Local variables → stack (short-lived).

3) Why you cannot write static inside a method

The static modifier defines a class-level property. A method body is a local scope that creates short-lived stack variables for that execution. These are two incompatible concepts:

You cannot create a long-lived class member from inside a short-lived local scope.

Java's grammar and language rules simply disallow the static modifier on local variable declarations. The compiler enforces this.

scanner.next() will read only the first word that is being taken input

scanner.nextLine() will take in the entire line of input

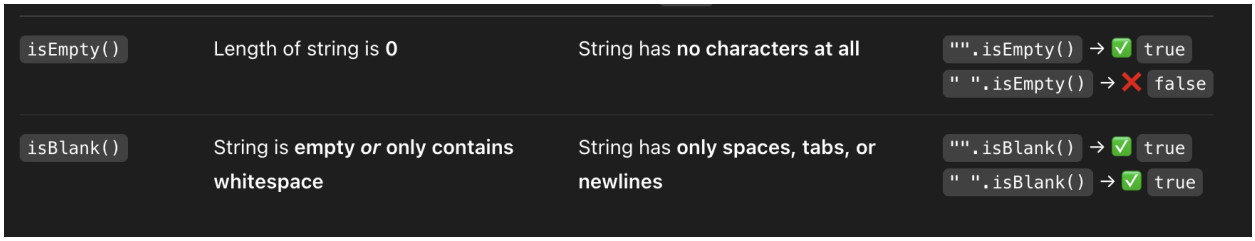
BUFFER

Using nextInt() and then taking input by nextLine() will cause a buffer issue as the "\n" enter character enters the nextLine() . There are two ways to resolve it either use nextLine() after the main nextLine() or use Integer.parseInt(scanner.nextLine());

```
System.out.printf("%04d\n",a);
```

Here the printf %04d means that it is an integer and the 04 means that if a=24 then it will print out 0024 as 0 will be the placeholder

DIFFERENCE BETWEEN IsEmpty() and IsBlank()

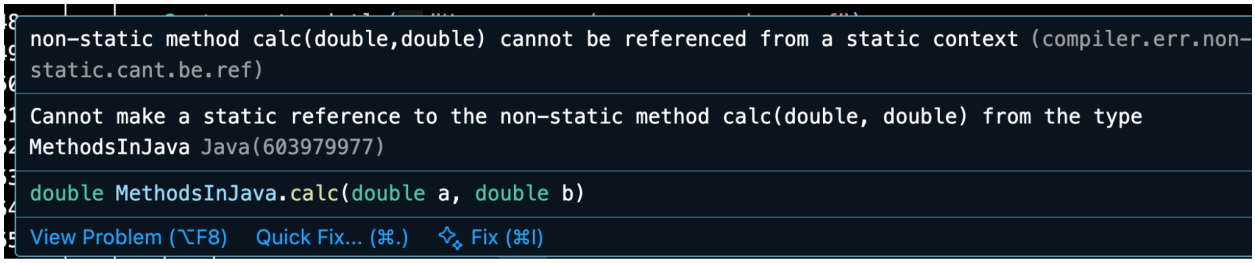


System.out.println(email.substring(4)); (This will start printing from the 4th index till the last index that is specified)

Break - breaks the loop

Continue - skips just the current iteration of the loop

Overloading - Same name but different signature (ya different parameters)



Refer this error - static methods cant have non static functions

Order of precedence for the scope of variables -
Locals > Class Based > Global static

Always use `Integer` with collections like `List`, `Map`, `Set`, etc.,
and use plain `int` for direct numeric operations.

Type	Category	Can be used in collections (like `List`)?	Stores	Example
`int`	**Primitive type**	❌ No	Raw numeric value (4 bytes)	`int x = 5;`
`Integer`	**Wrapper class (Object)**	✅ Yes	Object that wraps an `int`	`Integer x = 5;`

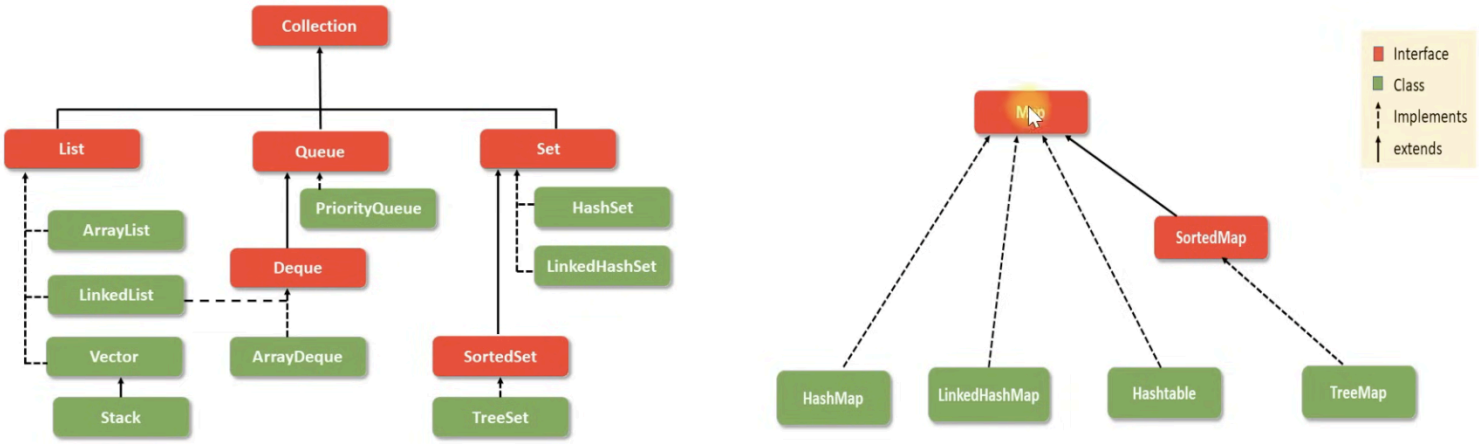
```
List<String> unstrlst = List.of(
    "blue",
    "yellow",
    "black"
);
//  unstrlst.add("orange");
```

This will create a IMMUTABLE List in python means if you uncomment the bottom part then it will cause an error Immutable exception error

THREAD SAFE

In Java, "thread-safe" refers to code, classes, or data structures that can be safely used and shared by multiple threads concurrently without leading to data corruption, inconsistent states, or unexpected behavior. Essentially, a thread-safe component guarantees correct operation even when multiple threads access and modify shared resources simultaneously.

JAVA COLLECTIONS FRAMEWORK



LISTS - Has implementations like arraylist, stack , queue
Difference between List ArrayList Vector LinkedList →

Type	Implements / Extends	Thread-Safe?	Internal Working	When to Use
List (interface)	Parent interface of all list types	—	Just defines methods (no implementation)	Used as reference type `List<String> list = new ArrayList<>();`
ArrayList	`implements List`	❌ Not synchronized (not thread-safe)	Uses **dynamic array** internally	Fast for searching / random access
Vector	`extends AbstractList` & `implements List`	✅ Thread-safe (synchronized)	Uses **dynamic array** (like ArrayList) but slower due to synchronization	Use only if synchronization required
LinkedList	`implements List, Deque`	❌ Not synchronized	Uses **doubly linked list**	Fast insertion/deletion (add/remove)

Important Note → TO KEEP IN MIND - List<String> list = new ArrayList<>(); YOU Must have noticed somethings like this or like Queue<String> qu = new LinkedList<>(); This is basically

done to ensure polymorphism property in Java . It basically means “Make me a list-type variable, but the actual working object behind it is an ArrayList.”

You can replace that `ArrayList` with any other class that also implements `List`: like list = new LinkedList<>(); OR list = new Vector<>();

Difference between Class Abstract class and Interface

Feature	Class	Abstract Class	Interface
What it is	Blueprint to create objects	Partially implemented class (some complete + some incomplete methods)	Fully abstract type (only meth declarations before Java 8)
Can we create object?	✅ Yes	❌ No	❌ No
Contains methods?	Concrete (complete) methods	Both abstract + concrete methods	Only abstract methods (till Java 7), from Java 8 → default & static allowed
Keyword	`class`	`abstract class`	`interface`
Variables	Normal instance variables	Normal instance variables	Public, static, final (by default)
Method modifiers	Anything (public/private/protected)	Anything	Public & abstract by default
Inheritance	`extends` another class (single)	`extends` another class / abstract class (single)	`extends` other interfaces (multiple allowed)
Implementation	—	Subclass **extends** abstract class and **implements** abstract methods	Class **implements** interface
Purpose	To define full functionality	To share partial common behavior	To define a contract or rule

Copy table

RECORDS IN JAVA

In Java, a record is a special kind of class declaration. Its primary purpose is to provide a concise syntax for defining immutable data-carrying classes, effectively reducing boilerplate code.

Here's what records in Java mean:

Immutable Data Carriers: Records are designed to hold data and are inherently immutable. This means that once a record object is created, its internal state (the values of its components) cannot be changed.

Constructors IN JAVA

An important note in java is suppose that you have some main class of Student say →

```
class Student
{
   String name;
   int rollno;
   double gpa;
   boolean gender; // true for male false for female
   Student(String name, int rollno)
   {
       this.name = name;
       this.rollno = rollno;
   }
}
```

Suppose if we make any class like this and only declare only 2 values for these, then java by default will assign this one some default values like for gpa it will give 0.0 for boolean it will give false;

Super keyword IN JAVA (No need for empty constructors)

Super = refers to the parent class , used in constructors and method overriding and calls the parent constructor to initialize the attributes .

Now the parent **no longer has a default (empty) constructor**.

So the compiler can't automatically call `super();` anymore.

Syntax of super() (for constructors)

General form:

super(parameter1, parameter2, ...);

Rules:

Must be the first line in a constructor of the child class.

Calls the constructor of the parent class (superclass).

EXAMPLE →

The parameters you pass must match one of the parent's constructors.

```
class Parent {
    Parent(String name) {
        System.out.println("Parent constructor called for " + name);
    }
}

class Child extends Parent {
    Child(String name) {
        super(name); // calls Parent(String)
        System.out.println("Child constructor called for " + name);
    }
}
```

Abstract methods are used to hide the implementation details. We cant instantiate it as in the form of an object . Contains abstract methods which must be implemented. Contains concrete methods that are inherited .

ABSTRACT METHODS DONT HAVE BODY

SIMPLE JUST abstract void display(); finish

```
abstract class Shape
{
   String name;
   int length;
}
```

```
Shape is abstract; cannot be instantiated(compiler.err.abstract.cant.be.instantiated)
Cannot instantiate the type ShapeJava(16777373)
```

This comes on creating the constructor of the Shape class

All ABSTRACT METHODS WRITTEN MUST BE OVERWRITTEN PROPERLY for no error

🚫 **You *cannot* have an abstract method inside a normal (non-abstract) class.**

That's **illegal in Java**. Because a **normal class must be complete**, i.e., it cannot have unimplemented (abstract) behavior.

Interfaces → A CLASS CAN HAVE MULTIPLE PARENTS create multiple interfaces but will compulsorily have to define the methods used by them

> In an **interface**, any method you declare **without a body** is **implicitly:**
> `public abstract`
>
> **IMPORTANT**

In an **interface**, every method without a body is:`public abstract` by default – and it **cannot** be made `private`, `protected`, or package-private.

An abstract class, on the other hand, is more flexible. It can have: `public`, `protected`, or even `private` methods.
Both abstract (unimplemented) and non-abstract (implemented) methods.

Getters → methods that make a field readable String getNumber() {----}

Setters → Methods that make a field writable

AGGREGATORS AND COMPOSITORS IN JAVA LEARN IT

ENUM IN JAVA LEARN IT

THREADDING - MULTITHREADDING IN JAVA LEARN IT

GENERICS IN JAVA LEARN IT