Java 8 Features

Code reduce

shortcut methods

Data strutcure

COllections

8 features

Stream api

Interface avoid null point exception

Lambda Expresion
time Api
Interfaces
Marker interface
Stream api
Optional class
method refrences
Compatable features
Excutor service

Time Api

1 LocalDate — only date

2 LocalTime — only time

3 LocalDateTime — date + time

5 Duration — time difference

4 Period — date difference

6 Format date — DateTimeFormatter

9:30 → 12:
11:30 1:30

1979
2025 → ?

12:45 1:30

🧠 What's a Lambda Expression?

It's a **shortcut** for writing small functions or logic.

Used mainly for:

- Iteration (`forEach`)
- Functional interfaces
- Stream operations
- Sorting, filtering, mapping...

✅ Basic Syntax:

```java
(parameters) -> { body }
```

1°

100%

5

1 or %

✅ 1. **Normal Interface**

✅ 2. **Functional Interface** (`@FunctionalInterface` + Lambda)

✅ 3. **Default Method in Interface**

✅ 4. **Static Method in Interface**

🧠 What is a **Marker Interface** in Java?

A **marker interface** is an interface with **no methods**, no variables — just an empty tag

```java
interface Marker {
    // no methods at all!
}
```

Java uses it as a "**label**" to mark classes with some special behavior.

🎯 Purpose:

> **To mark or flag a class** for a specific purpose, and allow special treatment by JVM or libraries.

🔥 Popular Marker Interfaces in Java:

Interface	What it Marks
`Serializable`	Marks object as savable to file or stream
`Cloneable`	Allows object to be cloned using `.clone()`
`Remote`	Marks object for RMI
`ThreadSafe` (custom)	You can define your own marker to say "This is thread safe" ✅

🔥 Real-World Example: `Serializable`

```java
class Student implements Serializable {
    int id;
    String name;
}
```

When you try to **save the object to file**, Java checks:

> "Hmm... is this class `Serializable`?"

If yes → JVM allows it ✅

If not → 💥 Throws exception ❌

Main Thread 1

💡 What is a **Method Reference**?

It's just a **shorter way** to write a lambda that only calls an existing method.

🔁 Format:

```java
ClassNameOrObject::methodName
```

✅ Example 1: `System.out::println`

🔹 Without Method Reference (normal lambda):

```java
List<String> names = Arrays.asList("Surya", "John", "Alex");

names.forEach(name -> System.out.println(name));
```

✅ With Method Reference:

```java
names.forEach(System.out::println);
```

Same result, just cleaner.

🔥 Think of it like this:

Imagine you have numbers: `[1, 5, 7]`

And you reduce them like:

```java
.reduce(0, (c, e) -> c + e)
```

It works like:

1. Start with `c = 0`
2. Next element `e = 1` → `c = 0 + 1 = 1`
3. Next element `e = 5` → `c = 1 + 5 = 6`
4. Next element `e = 7` → `c = 6 + 7 = 13`

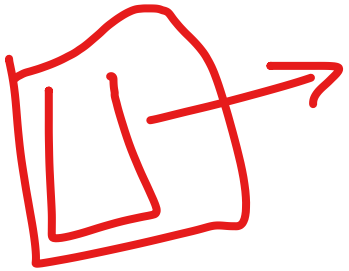
Final result = `13`

=1

1+5=

7

✅ 1. What is `CompletableFuture`?



- It's used for **asynchronous programming** in Java 8
- It runs code **in background thread** and continues without waiting (non-blocking)
- Part of `java.util.concurrent`

💥 Real-life:

> You send a food order (task), then go chill. The restaurant (thread) prepares it while you're doing other things 😎

CompletableFuture.supplyAsync(...)

```java
// Runs this Supplier in background thread 🔥
CompletableFuture<String> future = CompletableFuture.supplyAsync(() -> {
    try {
        Thread.sleep(1000); // simulate delay
    } catch (Exception e) {}
    return "👋 Hello from background!";
});

// Main thread doing something else
System.out.println("✅ Main thread running...");

// Block and wait for result (if you need it)
String result = future.join(); // or future.get()
System.out.println("🚀 Result: " + result);
```

supplyAsync(...) = runs code in **another thread**, not main thread.

Now our main thread keep going ,it will not stop

This line **waits** until the background task is finished

Stream API

DS

Collections

java 8 features

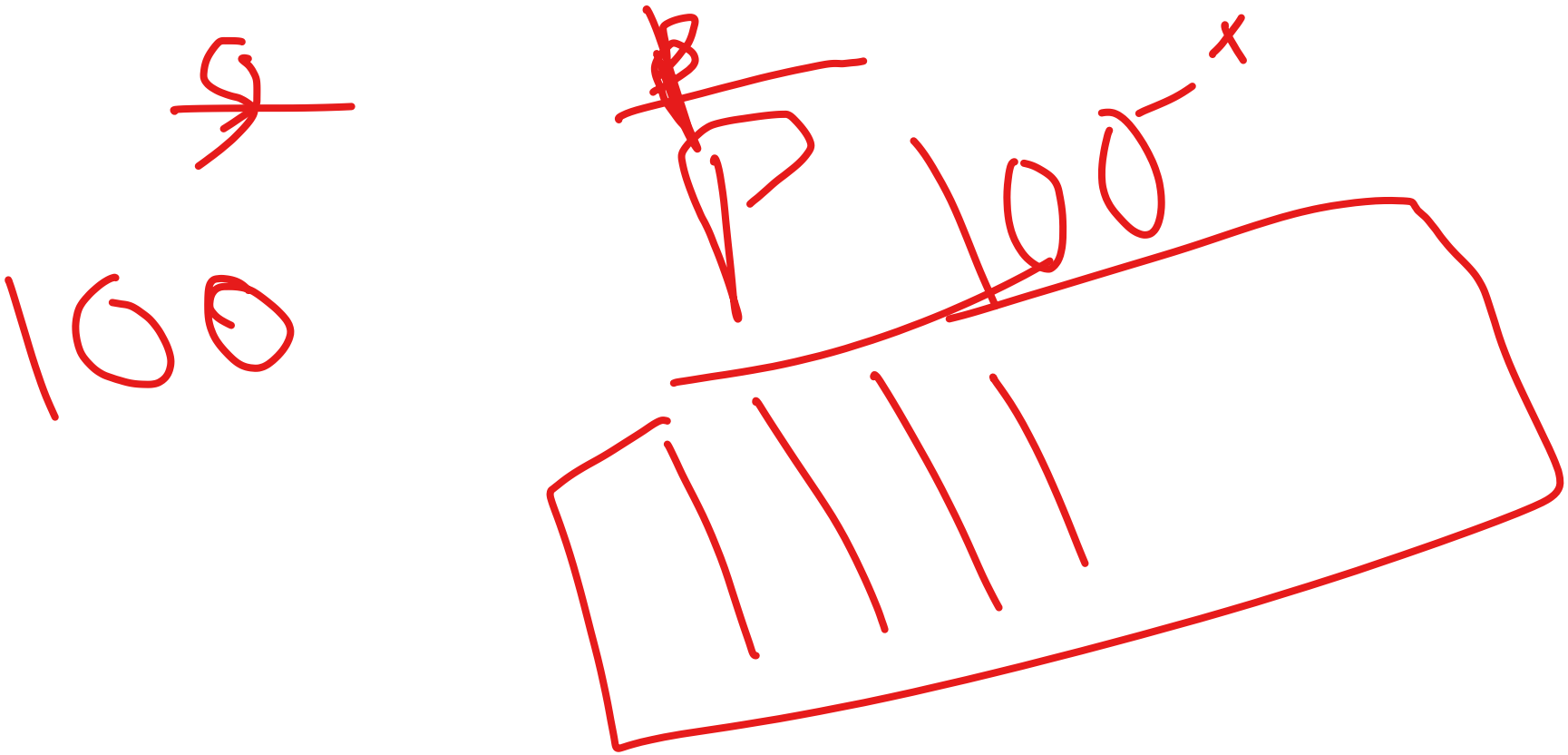
sorted

Map

Filter function

Reduce Function

Count



Optional class

String name=null

name.length()

Optional<USER> findBu userwithID(id);