✅ 1. Concurrency vs Parallelism (Simple Definition)

🟦 Concurrency

- Ability to **deal with multiple tasks at the same time**, but **not necessarily at the same instant**.
- Tasks take **turns** using the CPU (time slicing).

👉 Think of it as **multitasking** on a **single CPU**.

🟩 Parallelism

- Ability to **execute multiple tasks simultaneously**, **at the same exact time**.
- Requires **multiple CPU cores**.

👉 Think of it as **multiple people doing work together**.

⚡ Quick Comparison Table

Feature	Concurrency	Parallelism
CPU requirement	Can run on 1 core	Requires 2+ cores
Execution	Tasks **interleave**	Tasks **run together**
Goal	Responsiveness	Speed
Example	Switching between tasks	Running tasks at same time

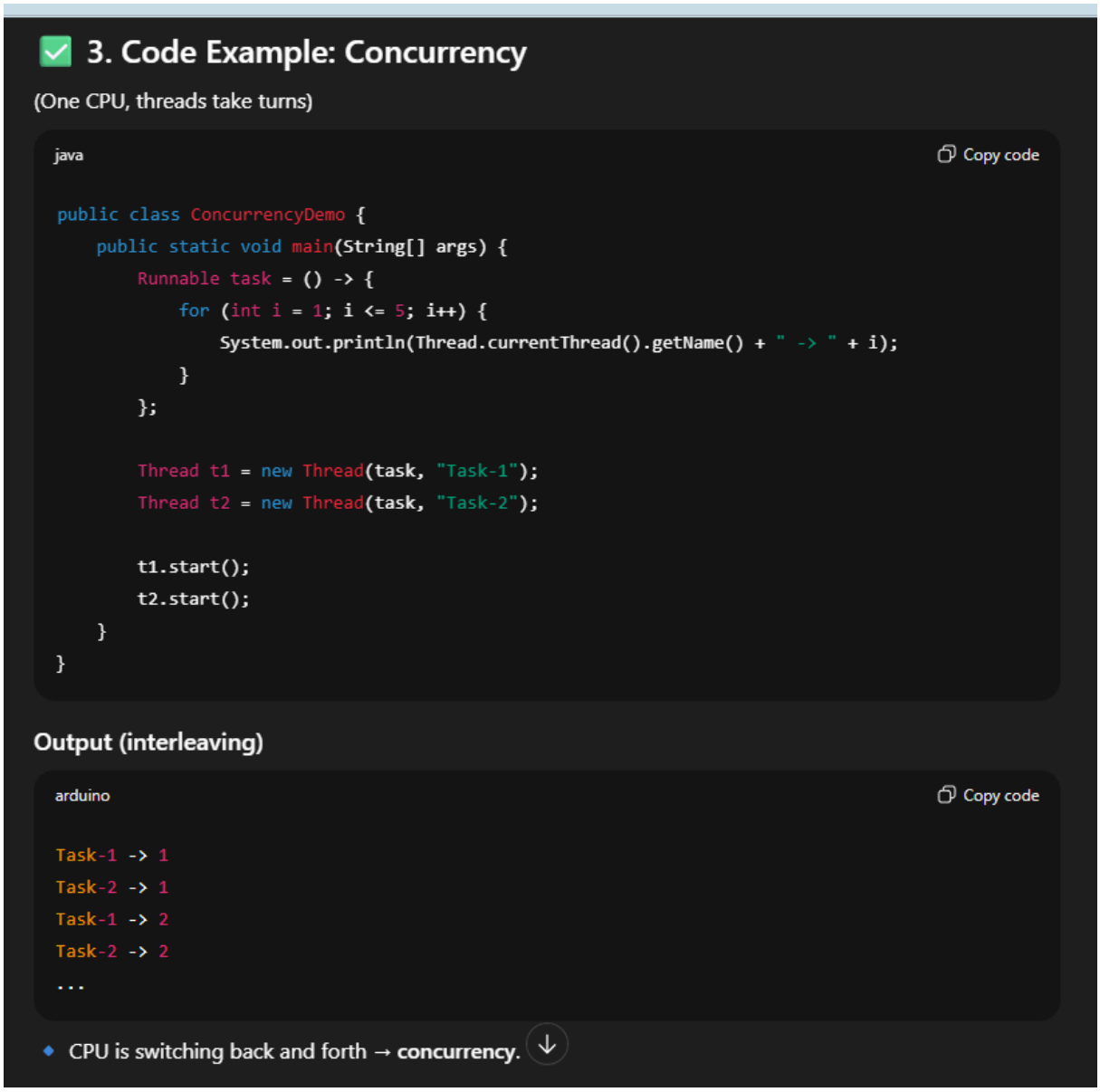
✅ 2. Real-life Analogy

Concurrency analogy (single waiter)

One waiter handles **5 tables** by **switching** between them.
He **appears** to handle all 5 at once — but actually serves one at a time.

Parallelism analogy (five waiters)

Five waiters serve **five tables simultaneously**.
All tasks happen **at the same time**.



✅ 4. Code Example: Parallelism

To *force* parallelism, you need **multiple CPU cores** and an **ExecutorService** with multiple threads.

java Copy code

```java
ExecutorService executor = Executors.newFixedThreadPool(4);

for (int i = 1; i <= 4; i++) {
    final int taskId = i;
    executor.submit(() -> {
        System.out.println("Running Task " + taskId +
                " on thread " + Thread.currentThread().getName());
    });
}

executor.shutdown();
```

Expected Output (tasks run truly in parallel on different cores):

arduino Copy code

```arduino
Running Task 1 on thread pool-1-thread-1
Running Task 2 on thread pool-1-thread-2
Running Task 3 on thread pool-1-thread-3
Running Task 4 on thread pool-1-thread-4
```

✨ 5. When to Use Concurrency?

When you want to make the **system faster or responsive** even with **one CPU**.

✔ Real-life uses of concurrency

App/System	Why concurrency?
WhatsApp	Sending + receiving messages simultaneously
Web servers	Handle thousands of requests by switching between them
UI apps	UI thread + background worker
Online shopping site	Multiple user requests handled via thread pool

---

✨ 6. When to Use Parallelism?

When you want things to run **much faster** by using **multiple CPU cores**.

✔ Real-life uses of parallelism

Scenario	How parallelism helps
Video processing	Each frame processed by separate core
Machine learning	Matrix operations in parallel
Big data (Hadoop/Spark)	Tasks distributed across clusters
Web crawler	100 threads fetching URLs simultaneously
Sorting large arrays	Parallel sort

🔥 7. Real-life Problem When Concurrency Is Needed

Problem: A food delivery app (like Zomato)

You need to process:

- New order creation
- Payment verification
- Assign driver
- Notification sending
- Live tracking updates

These tasks happen **together** but not at the **same exact millisecond**.

👉 This is **concurrency** using message queues + thread pool.

🔥 8. Real-life Problem When Parallelism Is Needed

Problem: Processing 10K images

You want to compress all images FAST.

Solution:
Use **parallel streams** / ExecutorService with 8 threads on an 8-core machine.

Each thread compresses separate images.

👉 This makes it 8x faster → **parallelism**.

🧠 Final Summary (Interview Ready)

Term	Key idea	CPU cores	Example
Concurrency	Tasks start, pause, resume	1 or many	Server switching between requests
Parallelism	Tasks run at **same instant**	2+ cores	Image processing, ML, scientific computing

✅ 1. What is Parallelism? (Super Simple Idea)

Parallelism = doing multiple tasks at the EXACT SAME TIME.

This only happens when you have **multiple CPU cores**.

Example:

- Your laptop has **4 cores**.

- It can run **4 tasks simultaneously**.

So:

- Thread-1 → runs on Core 1
- Thread-2 → runs on Core 2
- Thread-3 → runs on Core 3
- Thread-4 → runs on Core 4

All **run at the same moment**.

👉 This is NOT possible on single-core CPU.

🧠 2. Concurrency vs Parallelism (super-short)

Concurrency = switching

Your brain switches between:

- WhatsApp
- Instagram
- YouTube

But one at a time (very fast switching).

Parallelism = simultaneous

Your two eyes can see two objects at the same time.
That's parallelism.

🛠️ 3. Real Life Parallelism Examples

✔ Example 1: Making Rotis

You and your mother both make rotis.

- You roll the roti
- She cooks it on tawa

Both tasks are happening **together** → parallelism.

✔ Example 2: Washing utensils

One person washes, another wipes → simultaneous.

✔ Example 3: Four barbers cutting hair

4 customers at the same time.
Parallel work → multiple workers.

🖥️ 4. Computer Parallelism Example (Beginner friendly)

Imagine you have:

- 4 CPU cores
- 4 tasks

If each task takes **5 seconds**, then:

Without parallelism (sequential):

Total time = 5 + 5 + 5 + 5 = **20 seconds**

With parallelism (4 cores):

All 4 tasks run at the SAME TIME
Total time = **5 seconds**

That *huge* speed improvement = **parallelism**.

🚀 5. NOW: Java Parallelism in Correct Simple Way

To run tasks in parallel you need:

1. **Multiple threads**
2. **Enough CPU cores**

In Java, we use:

```
ExecutorService executor = Executors.newFixedThreadPool(4);
```

This simply means:

- Create **4 worker threads**
- They can run **4 tasks at the same time**

(Just like 4 barbers in a shop.)

🔍 6. Simple Parallel Example in Java (Easiest Version)

java Copy code

```java
ExecutorService executor = Executors.newFixedThreadPool(4);

Runnable task = () -> {
    System.out.println(Thread.currentThread().getName() + " is working...");
    try { Thread.sleep(2000); } catch (Exception e) {}
};

executor.submit(task);
executor.submit(task);
executor.submit(task);
executor.submit(task);

executor.shutdown();
```

What happens?

What happens?

We submitted 4 tasks.
4 threads are available.

So:

- Task 1 runs on thread-1
- Task 2 runs on thread-2
- Task 3 runs on thread-3
- Task 4 runs on thread-4

All **executing exactly at same time**.

Sample output:

```
pool-1-thread-1 is working...
pool-1-thread-3 is working...
pool-1-thread-4 is working...
pool-1-thread-2 is working...
```

Notice:
All printed almost at the **same time** → parallelism.

📌 7. Why is ExecutorService required for parallelism?

Because:

- `new Thread()` creates threads one-by-one (not scalable)
- ExecutorService creates a **thread pool**
- Thread pool allows many tasks to run parallel efficiently

It works like:

Thread pool = a group of workers (threads) waiting for tasks.

🎨 8. Visual Diagram (Mental Model)

```
CPU Core 1 ---> Thread-1 ---> Task A
CPU Core 2 ---> Thread-2 ---> Task B
CPU Core 3 ---> Thread-3 ---> Task C
CPU Core 4 ---> Thread-4 ---> Task D
```

All happening **together**.

🧪 9. Parallelism Proof (with sleep)

Try this code:

java Copy code

```java
ExecutorService executor = Executors.newFixedThreadPool(4);

long start = System.currentTimeMillis();

for (int i = 1; i <= 4; i++) {
    executor.submit(() -> {
        try { Thread.sleep(2000); } catch (Exception e) {}
    });
}

executor.shutdown();
executor.awaitTermination(10, TimeUnit.SECONDS);

long end = System.currentTimeMillis();
System.out.println("Total time: " + (end - start));
```

Expected Result

Total time ≈ **2000ms (2 seconds)**
NOT 8000ms.

That proves:

- All 4 tasks slept for 2s **simultaneously**
- not sequentially

This is **parallelism**.

🟢 10. Real-World Use Case of Parallelism

Image Processing App (e.g., Instagram filters)

You want to apply a filter to 100 images.

Without parallelism (single thread):

- takes 100 × 2 seconds = 200 seconds

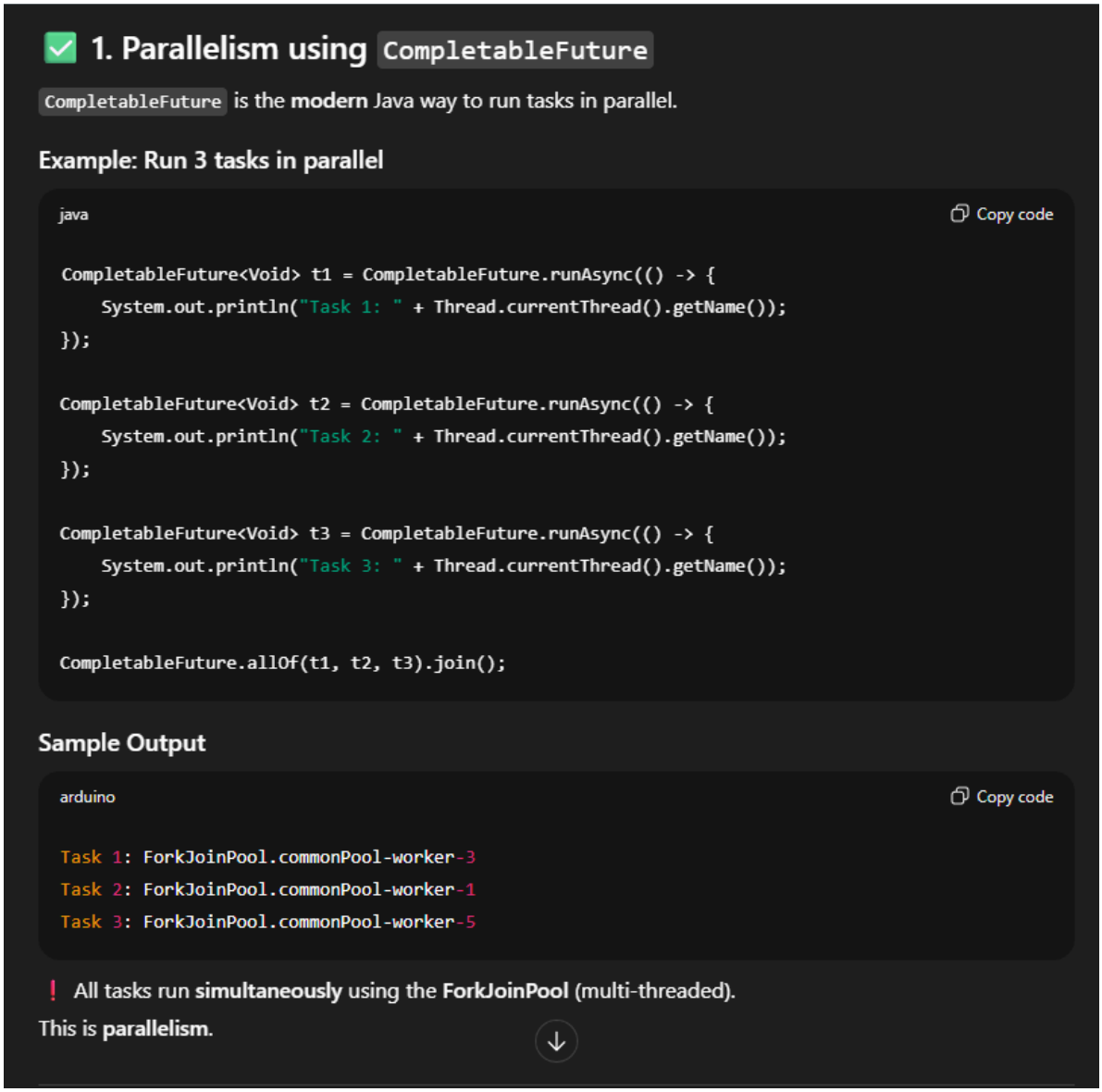
With 4 threads:

- processes 4 images at a time
- time becomes ≈ 50 seconds

Huge performance boost → parallelism.

⭐ Final Summary (Very Easy)

Concept	Meaning	Example
Concurrency	switching between tasks	single waiter for 10 tables
Parallelism	doing tasks at the same time	4 barbers → 4 customers



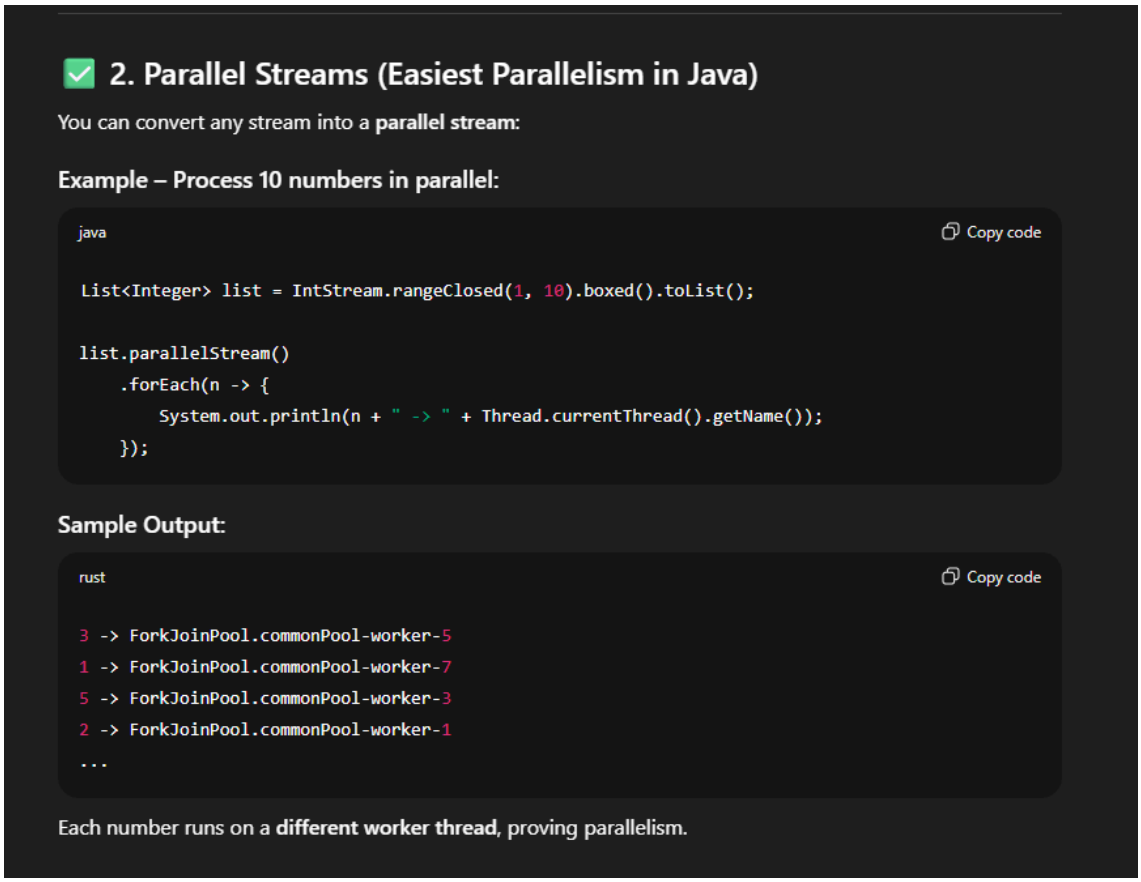
🟦 Why CompletableFuture is better than Future?

- No blocking (`Future.get()` blocks)
- Supports chaining (`thenApply`, `thenAccept`, `thenCompose`)
- Excellent for **async REST calls**, **parallel computation**, etc.

✅ 2. Parallel Streams (Easiest Parallelism in Java)

You can convert any stream into a **parallel stream**:

Example – Process 10 numbers in parallel:



☑ When to use parallel streams?

Good for:

- CPU-heavy tasks
- Pure functions

- Large lists/arrays

Avoid when:

- Tasks modify shared data
- Tasks involve I/O
- Order is important

🚀 3. Thread vs ExecutorService (must understand)

🔴 `new Thread()` (traditional approach)

```
new Thread(() -> {
    // work
}).start();
```

Problems:

- No reuse of threads
- Creates too many threads → slows system
- No task scheduling
- No thread pooling
- Hard to manage & monitor

🟢 ExecutorService (modern approach)

```
ExecutorService executor = Executors.newFixedThreadPool(4);
executor.submit(() -> {
```

```
    // work
});
executor.shutdown();
```

Advantages:

Feature	Thread	ExecutorService
Thread reuse	❌ No	✔ Yes
Task queue	❌ No	✔ Yes
Thread pool	❌ No	✔ Yes
Better performance	❌	✔✔✔
Easier parallelism	❌	✔
Controls #threads	❌	✔

ExecutorService is ALWAYS preferred in real applications.

💡 Simple Analogy

Thread	ExecutorService
Hiring a new person for every job	Having a fixed team of workers
Slow & expensive	Fast & efficient

⭐ 4. How Java decides which thread runs on which CPU core?

Java does **NOT** decide this.
The **Operating System** (Windows/Linux/Mac) does it.

Steps:

1. Java creates a thread
2. Java tells the OS: “I have this thread ready to run”
3. The OS scheduler decides:
    - which CPU core will run the thread
    - when to pause it
    - when to resume it
    - how long to let it run

The OS ensures:

- load balancing
- fair CPU sharing
- minimizing context switching

🧠 CPU Core Scheduling (Simple Visualization)

```
Thread-1 → Core 1
Thread-2 → Core 2
Thread-3 → Core 3
Thread-4 → Core 4
Thread-5 → waits (no free core)
```

If there are:

- 4 cores

- 6 threads

Then 4 threads run **parallel**,
2 threads **wait or get scheduled later**.

✅ 1. Generic Type (e.g., `List<T>` , `List<String>` , `Box<Integer>`)

What it means

You specify a **concrete type** while creating the object.

Example:

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

Here:

- The type is **fixed**.
- You can **add** elements of that exact type (`String` only).
- You can **get** elements safely as `String`.

Properties

Feature	Generic Type
Can add elements	✅ Yes
Can read elements	✅ Yes

Type is fixed	✅ Yes
Flexible?	❌ No

Example

```
List<Number> numbers = new ArrayList<>();

numbers.add(10);        // OK

numbers.add(10.5);      // OK
```

✅ 2. Wildcard Generic Type (`?`, `? extends T`, `? super T`)

What it means

Wildcard allows **flexible / unknown** types.

Used mainly in **method parameters**, not in object creation.

Example:

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

Here:

- You **don't know** the exact type inside the list.
- You **cannot add anything** except `null`.
- You can read only as `Object`.

🎯 Why use wildcards?

For **flexibility** in method parameters.

Three wildcard types:

1️⃣ `List<?>` → Unknown type (read-only)

```
void printList(List<?> list) {
    for (Object obj : list) {
        System.out.println(obj);
    }
}
```

- Can pass `List<String>`, `List<Integer>`, `List<Double>`
- **Cannot add** anything to the list (except null).

2️⃣ `List<? extends Number>` → Upper bounded wildcard

Means: the list contains some type that **extends Number**.

```
List<? extends Number> list = new ArrayList<Integer>();
```

Rules

- You can read elements as Number
- **Cannot add** anything (except null) because exact type is unknown.

Used for **reading**.

3️⃣ `List<? super Integer>` → Lower bounded wildcard

Means: list contains some type that is a **superclass of Integer**.

```
List<? super Integer> list = new ArrayList<Number>();
```

Rules

- You can **add Integer** (and its subclasses).
- While reading, you get `Object`.

Used for **writing**.

🔥 Summary Table

Feature	Generic Type (`List<String>`)	Wildcard (`List<?>`, `List<? extends T>`, `List<? super T>`)
Type is fixed	✅ Yes	❌ No
Read elements	✅ Yes	⚠️ Yes, but with restrictions
Add elements	✅ Yes	❌ No for `?` and `? extends` ✅ Allowed for `? super`

Used in	Creating objects & variables	Mostly method parameters
Flexibility	❌ Low	✅ High
Use case	You know exact type	You want generic method that accepts many types

---

🎯 When to use what?

Use Generic Type when:

✔ You know the exact type
✔ You want to add and read freely

Use Wildcard when:

✔ You want your method to accept many types
✔ You want flexibility
✔ You want **read-only** or **write-only** structure

👍 Quick Memory Trick

PECS → Producer Extends, Consumer Super

- `? extends T` → list produces T (you read T)
- `? super T` → list consumes T (you add T)

```java
public static void main(String args[]) {

    ThreadPoolExecutor poolExecutor = new ThreadPoolExecutor( corePoolSize: 3, maximumPoolSize: 3,
            new ArrayBlockingQueue<>( capacity: 10), Executors.defaultThreadFactory(), new ThreadP

    //UseCase1
    Future<?> futureObject1 = poolExecutor.submit(() -> {
        System.out.println("Task1 with Runnable");
    });

    try {
        Object object = futureObject1.get();
        System.out.println(object == null);
    }catch (Exception e){
    }

    //UseCase2
    List<Integer> output = new ArrayList<>();
    Future<List<Integer>> futureObject2 = poolExecutor.submit(() -> {
        output.add(100);
        System.out.println("Task2 with Runnable and Return object");
    }, output);

    try {
        List<Integer> outputFromFutureObject2 = futureObject2.get();
        System.out.println(outputFromFutureObject2.get(0));
    } catch (Exception e) {
    }

    //UseCase3
    Future<List<Integer>> futureObject3 = poolExecutor.submit(() -> {
        System.out.println("Task3 with Callable");
        List<Integer> listObj = new ArrayList<>();
        listObj.add(200);
        return listObj;
    });

    try {
        List<Integer> outputFromFutureObject3 = futureObject3.get();
        System.out.println(outputFromFutureObject3.get(0));
    } catch (Exception e) {
    }
}
```

✅ UseCase 1 – Runnable always returns `null`

Code:

```
Future<?> futureObject1 = poolExecutor.submit(() -> {
    System.out.println("Task1 with Runnable");
});
Object object = futureObject1.get();  // ALWAYS null
```

Why?

Because:

- You passed a **Runnable**, not a Callable.
- Runnable does **not** return anything.
- Therefore executor service internally returns **null**.

✔ *Correct behavior*
✔ *As per Java API*

✅ UseCase 2 – Runnable + Pre-Created Result Object

Code:

```
List<Integer> output = new ArrayList<>();
Future<List<Integer>> futureObject2 = poolExecutor.submit(() -> {
    output.add(100);
    System.out.println("Task2 with Runnable and Return object");
}, output);
```

Why this works?

Because here you call this version:

```
<T> Future<T> submit(Runnable task, T result)
```

This method:

- Runs the runnable
- Returns the **same result object** you passed

Important:

- The **Runnable still does NOT return** anything.
- But YOU provide a return object → `output`
- Executor simply gives you the **same reference back**.

✔ *Correct behavior*
✔ *Return value is manually supplied*
✔ *Used for shared mutable results*

✅ UseCase 3 – Callable returning ArrayList

Code:

```
Future<List<Integer>> futureObject3 = poolExecutor.submit(() -> {
    System.out.println("Task3 with Callable");
    List<Integer> listObj = new ArrayList<>();
    listObj.add(200);
    return listObj;
});
```

Why this works?

Because:

- You passed a **Callable**
- Callable returns a value

- Whatever you `return` reaches `future.get()`

✔ *Correct behavior*
✔ *This is the real way to return values from background tasks*

🔥 So what is the difference between UseCase 2 and UseCase 3?

Feature	UseCase 2	UseCase 3
Uses Runnable	✅ Yes	❌ No
Uses Callable	❌ No	✅ Yes
How result is returned?	Executor returns the **object you provide**	Callable returns **its own object**
Can you compute and return new object?	❌ No (Runnable doesn't return)	✅ Yes
Type safety	Low	High
Recommended?	Rarely	✔ Yes (preferred way)
Does task create its own return object?	❌ No	✔ Yes

UseCase 2 = "Take this object → modify it → return it back"

UseCase 3 = "Create new object → compute → return it"

✔ Which one should you use generally?

Always UseCase 3 (Callable)

Cleaner, predictable, thread-safe, and used everywhere in real-world apps.

UseCase 2 is rarely used in practice because:

- You are passing a mutable object
- Thread safety issues can happen
- Return object is not "computed", just "provided"

🎯 Summary:

Case	Task Type	Return Mechanism
1	Runnable	Always returns `null`
2	Runnable	Returns **given object**
3	Callable	Returns **computed object**

✅ Async Programming (Easy Definition)

Async programming = Start a task → don't wait → continue → get the result later.

It is the opposite of **synchronous programming**, where you wait for each task to finish before starting the next one.

🔥 Example to Understand

☕ Synchronous (Blocking)

You order a coffee → stand there and wait → nothing else happens.

```
orderCoffee()
wait()
drinkCoffee()
```

🛵 Asynchronous (Non-Blocking)

You order a coffee → then sit or use your phone → when coffee is ready, they call you.

```
orderCoffeeAsync()
→ continue doing other work
→ callback: coffeeReady()
```

You didn't block your time.

🧠 KEY IDEAS in Async Programming

1️⃣ Non-blocking

The calling thread **does NOT wait** for the task to finish.

2️⃣ Concurrency without extra threads

In async model, few threads handle many tasks using callbacks, events, and futures.

3️⃣ Future / Promise / Callback

Instead of waiting, your program gets notified later:

- Java → `Future`, `CompletableFuture`
- JavaScript → `Promise`, `async/await`
- Python → `asyncio`

4️⃣ Better performance

Useful when waiting on:

- Network calls
- File/DB I/O

- HTTP API requests
- Disk reads/writes

Threads don’t sit idle waiting.

⚙️ Async vs Multithreading (Most Developers Confuse This)

Concept	Async Programming	Multithreading
Uses multiple threads?	❌ Optional	✅ Yes
Key idea	Non-blocking	Parallel execution
Example	CompletableFuture	Thread, ExecutorService
Useful for	I/O-heavy tasks	CPU-heavy tasks

➡ Async ≠ Multithreading
➡ Async can run on **a single thread** (e.g., Node.js)
➡ Async focuses on **not blocking**

📌 When is Async Programming Useful?

✔ Network calls (API calls)
✔ File system I/O
✔ Logging
✔ Database queries
✔ Message queues
✔ Microservices communication

Not useful for:

- CPU-heavy operations (use multithreading instead)

🎯 One-Line Definition to Use in Interview

Async programming is a non-blocking way to execute tasks where a thread initiates work and continues without waiting, and the result is handled later using futures, callbacks, or events.