ChatGPT

Java Concurrency: Future vs CompletableFuture

What is `Future` in Java?

`Future` is used to represent the result of an asynchronous computation. It is part of `java.util.concurrent` package.

Key Features:

- Represents the result of a background task
- Provides blocking methods to get the result
- Works with `ExecutorService`

Important Methods:

- `get()` — blocks until the result is available
- `get(timeout, unit)` — waits for a specific time
- `isDone()` — returns true if the task is completed
- `isCancelled()` — checks if the task was cancelled
- `cancel(true/false)` — attempts to cancel the task

Example:

```
ExecutorService executor = Executors.newSingleThreadExecutor();
Future<String> future = executor.submit(() -> {
    Thread.sleep(2000);
    return "Hello from Future!";
});
String result = future.get(); // blocking
```

What is `CompletableFuture`?

Introduced in Java 8, `CompletableFuture` provides a non-blocking way to handle asynchronous computations and chain multiple tasks.

Key Features:

- Asynchronous & non-blocking
- Can chain multiple tasks
- Combine multiple futures

- Built-in exception handling

Important Methods:

- `supplyAsync()` / `runAsync()` — start async task
- `thenApply()` — transform result
- `thenAccept()` — consume result
- `thenRun()` — run action after task completes
- `thenCombine()` — combine two futures
- `allOf()` / `anyOf()` — handle multiple futures
- `exceptionally()`, `handle()` — handle exceptions

Example:

```
CompletableFuture.supplyAsync(() -> "Hello")
    .thenApply(s -> s + " World")
    .thenAccept(System.out::println); // non-blocking
```

📚 Real-World Use Case: Async Dashboard

Scenario:

Fetch data from multiple remote services in parallel: - User info - Account balance - Order history

📊 CompletableFuture Example:

```
CompletableFuture<String> userFuture = CompletableFuture.supplyAsync(() ->
fetchUserInfo());
CompletableFuture<String> balanceFuture = CompletableFuture.supplyAsync(() ->
fetchAccountBalance());
CompletableFuture<String> orderFuture = CompletableFuture.supplyAsync(() ->
fetchOrderHistory());

CompletableFuture<Void> all = CompletableFuture.allOf(userFuture,
balanceFuture, orderFuture);
all.thenRun(() -> {
    String user = userFuture.join();
    String balance = balanceFuture.join();
    String orders = orderFuture.join();
    System.out.println(user + ", " + balance + ", " + orders);
});
```

🛡 With Exception Handling:

```
CompletableFuture<String> balanceFuture = CompletableFuture
    .supplyAsync(() -> fetchAccountBalance())
    .exceptionally(ex -> "Balance unavailable");
```

🔍 Summary Comparison

Feature	`Future`	`CompletableFuture`
Blocking get	Yes	Optional (get/join)
Non-blocking chaining	No	Yes
Combine multiple futures	No	Yes
Exception handling	No	Yes
Works well with streams	No	Yes

Use `CompletableFuture` for modern, scalable, and non-blocking asynchronous workflows.