Absolutely, Gyanenjay! Here's a **complete guide** to CompletableFuture in Java with **key concepts**, **methods**, **interview questions with answers**, and **real-life scenarios** – all structured in a simple and effective way.

**✅ What is CompletableFuture?**

CompletableFuture is part of Java 8’s java.util.concurrent package. It represents a **future result** of an asynchronous computation and provides **non-blocking**, **composable**, and **reactive** ways to handle async logic.

**🔧 Core Features of CompletableFuture:**

| **Feature** | **Description** |
| --- | --- |
| **Asynchronous execution** | Runs tasks in background threads. |
| **Non-blocking** | Avoids .get() by using callback methods. |
| **Chaining** | Allows sequential task execution (pipeline). |
| **Combining** | Combine multiple futures. |
| **Exception handling** | Gracefully recover from errors. |
| **Manual completion** | You can complete a future manually. |

**🛠️ Most Commonly Used Methods:**

| **Method** | **Purpose** |
| --- | --- |
| supplyAsync(Supplier) | Runs task asynchronously and returns result. |
| runAsync(Runnable) | Runs a void task asynchronously. |
| thenApply(Function) | Transforms result. |
| thenAccept(Consumer) | Consumes result (no return). |
| thenRun(Runnable) | Runs after task completes (no input/output). |
| thenCompose(Function) | Flattens and chains dependent futures. |
| thenCombine(future, BiFunction) | Combines results from 2 futures. |
| exceptionally(Function) | Error recovery logic. |
| whenComplete(BiConsumer) | Runs whether success or failure. |
| complete(value) | Manually complete future. |
| join() | Same as get() but throws unchecked exceptions. |

**🔄 Example: Chaining Async Tasks**

CompletableFuture.supplyAsync(() -> {

return "Hello";

}).thenApply(msg -> {

return msg + " World";

}).thenAccept(finalMsg -> {

System.out.println(finalMsg);

});

**🧩 Combining Two Futures:**

CompletableFuture<Integer> f1 = CompletableFuture.supplyAsync(() -> 10);

CompletableFuture<Integer> f2 = CompletableFuture.supplyAsync(() -> 20);

CompletableFuture<Integer> result = f1.thenCombine(f2, (a, b) -> a + b);

System.out.println(result.get()); // Output: 30

**⚠️ Exception Handling:**

CompletableFuture<Integer> future = CompletableFuture.supplyAsync(() -> {

if (true) throw new RuntimeException("Error");

return 10;

}).exceptionally(ex -> {

System.out.println("Handled: " + ex.getMessage());

return 0;

});

System.out.println(future.get()); // Output: 0

**✅ Real-Life Use Cases**

* Parallel HTTP calls
* Microservices communication
* Background processing
* Chaining DB/API calls
* Retry & fallback logic

**🎯 Top Interview Questions with Answers**

**1. What is CompletableFuture?**

It is an enhanced version of Future introduced in Java 8 that allows asynchronous, non-blocking operations with chaining and better error handling.

**2. What is the difference between Future and CompletableFuture?**

| **Feature** | **Future** | **CompletableFuture** |
| --- | --- | --- |
| Blocking | Yes (get()) | No (uses callbacks) |
| Chaining | No | Yes |
| Exception Handling | Only via try-catch | In-built (exceptionally, handle) |
| Manual Completion | No | Yes |

**3. What is supplyAsync() vs runAsync()?**

* supplyAsync returns a result.

CompletableFuture<String> f = CompletableFuture.supplyAsync(() -> "data");

* runAsync performs a task without returning a result.

CompletableFuture<Void> f = CompletableFuture.runAsync(() -> System.out.println("task"));

**4. What does thenApply do?**

It is used to transform the result of a future.

CompletableFuture.supplyAsync(() -> 5)

.thenApply(i -> i \* 2); // Output: 10

**5. What is thenCompose vs thenCombine?**

| **Method** | **Purpose** | **Example** |
| --- | --- | --- |
| thenCompose | Chains dependent futures | API call 2 depends on API call 1 |
| thenCombine | Merges two independent futures | Add two async results |

**6. How do you handle exceptions in CompletableFuture?**

* exceptionally(): Fallback
* handle(): Process result or error
* whenComplete(): Side effect

**7. What if one CompletableFuture fails in a chain?**

The next stages are **skipped**, unless you use handle() or exceptionally() to recover.

**8. Is CompletableFuture thread-safe?**

Yes. The state transitions (e.g. completion) are thread-safe.

**9. Which thread pool does CompletableFuture use by default?**

The **ForkJoinPool.commonPool()**, unless a custom Executor is supplied.

**10. How do you cancel a CompletableFuture?**

CompletableFuture<String> future = new CompletableFuture<>();

future.cancel(true); // marks as cancelled

**🧠 Summary Cheat Sheet**

* ✅ Use supplyAsync() for async with return.
* 🔗 Chain logic using thenApply, thenCompose, etc.
* 💥 Handle errors via exceptionally or handle.
* ⚡ Combine tasks with thenCombine.
* 🧵 Custom executor = better thread control.

Would you like me to give you a **real-world scenario** using CompletableFuture in a **Spring Boot microservice** (e.g., calling 2 services in parallel)?