✅ `CompletableFuture` — Cheat Sheet (Grouped by Category)

🏁 1. Start/Run Asynchronous Computations

Method	Arguments	Return Type	✅ Use When	🚫 Avoid When
`supplyAsync`	`Supplier<T>` or `Supplier<T>, Executor`	`CompletableFuture<T>`	You need to run a task that **returns a result** asynchronously	If task is trivial and blocking the thread
`runAsync`	`Runnable` or `Runnable, Executor`	`CompletableFuture<Void>`	You need to run a task that **does NOT return a result**	If you need result from task

---

🔗 2. Chaining / Intermediate Operations

Method	Arguments	Return Type	✅ Use When	🚫 Avoid When
`thenApply`	`Function<T,R>`	`CompletableFuture<R>`	You want to **transform** a result (sync)	Heavy/long transformations (prefer async)
`thenApplyAsync`	`Function<T,R>`, (optionally `Executor`)	`CompletableFuture<R>`	Same as above but **on a different thread**	For very small, fast transformations
`thenAccept`	`Consumer<T>`	`CompletableFuture<Void>`	You want to **consume** result without transforming	You need further chaining after
`thenAcceptAsync`	`Consumer<T>`, (optionally `Executor`)	`CompletableFuture<Void>`	Same as above but **async**	For ultra-light consumers

Method	Arguments	Return Type	✅ Use When	🚫 Avoid When
`thenRun`	`Runnable`	`CompletableFuture<Void>`	You want to **run a task after** completion but **don't need the result**	If you actually need the result
`thenRunAsync`	`Runnable`, (optionally `Executor`)	`CompletableFuture<Void>`	Same as above but **async**	For simple follow-up tasks
`thenCompose`	`Function<T, CompletionStage<R>>`	`CompletableFuture<R>`	You want to **chain dependent futures**	When futures are independent
`thenCombine`	`CompletionStage<U>, BiFunction<T,U,R>`	`CompletableFuture<R>`	Merge results from two independent futures	If dependency is one-way
`thenAcceptBoth`	`CompletionStage<U>, BiConsumer<T,U>`	`CompletableFuture<Void>`	Run a task with two futures' results	If you need a return value
`runAfterBoth`	`CompletionStage<?>, Runnable`	`CompletableFuture<Void>`	Action after two tasks complete (no results needed)	If you need any results
`applyToEither`	`CompletionStage<T>, Function<T,U>`	`CompletableFuture<U>`	Pick the result of **whoever finishes first** and transform	If both results are critical
`acceptEither`	`CompletionStage<T>, Consumer<T>`	`CompletableFuture<Void>`	Pick the result of **whichever future completes first** and consume	If both must be considered
`runAfterEither`	`CompletionStage<?>, Runnable`	`CompletableFuture<Void>`	Run something after **either completes**	If you must use result

Method	Arguments	Return Type	✅ Use When	🚫 Avoid When
			(don't need result)	

🏁 3. Terminal / Handling / Exception Management

Method	Arguments	Return Type	✅ Use When	🚫 Avoid When
`get`	(blocking)	`T`	You must wait and get the result	If you want non-blocking
`join`	(blocking)	`T`	Like `get()`, but wraps exceptions into unchecked	Same as above
`getNow`	`T valueIfAbsent`	`T`	You want **non-blocking immediate result** if available	When guaranteed completion is needed
`complete`	`T value`	`boolean`	You want to **manually complete** the future	Normal completion expected
`completeExceptionally`	`Throwable ex`	`boolean`	You want to **manually fail** the future	When normal execution is still possible
`exceptionally`	`Function<Throwable, T>`	`CompletableFuture<T>`	Handle error by **providing fallback** value	When strict exception throwing needed
`handle`	`BiFunction<T, Throwable, R>`	`CompletableFuture<R>`	**Handle both success and error** paths	Very simple pipelines (adds overhead)
`handleAsync`	`BiFunction<T, Throwable,`	`CompletableFuture<R>`	**Handle both success**	Very simple

Method	Arguments	Return Type	✅ Use When	🚫 Avoid When
	`R>`, (optionally `Executor`)		**and error** paths but on another thread.	pipelines (adds overhead)
`whenComplete`	`BiConsumer<T, Throwable>`	`CompletableFuture<T>`	Perform side effects (logging, cleanup) regardless of success/failure	If you want to modify value
`whenCompleteAsync`	`BiConsumer<T, Throwable>`, (optionally `Executor`)	`CompletableFuture<T>`	Same as above but **on another thread**	For simple side effects

---

Method	Arguments	Return Type	Completion Condition	When to Use	When to Avoid
`allOf`	`CompletableFuture<?> ... futures`	`CompletableFuture<Void>`	Completes when **all** futures complete (success or exception)	When you need **all tasks to complete** before proceeding	If you only need **one future to complete** (use `anyOf`)
`anyOf`	`CompletableFuture<?> ... futures`	`CompletableFuture<Object>`	Completes when **any one** future completes (success or exception)	When you need to proceed as soon as **one task finishes**	When you need **all futures to complete** before proceeding

📝 Bonus Quick Tips:

- **Always prefer async methods (`xxxAsync`)** when doing heavy computation / IO.
- **Use `thenCompose`** to **flatten** futures (no nested CompletableFuture<CompletableFuture<T>>).
- **Use `thenCombine`** when two independent tasks must contribute to one final result.
- **Exceptionally useful** for error fallback, **handle** is good for audit/logging + fallback.