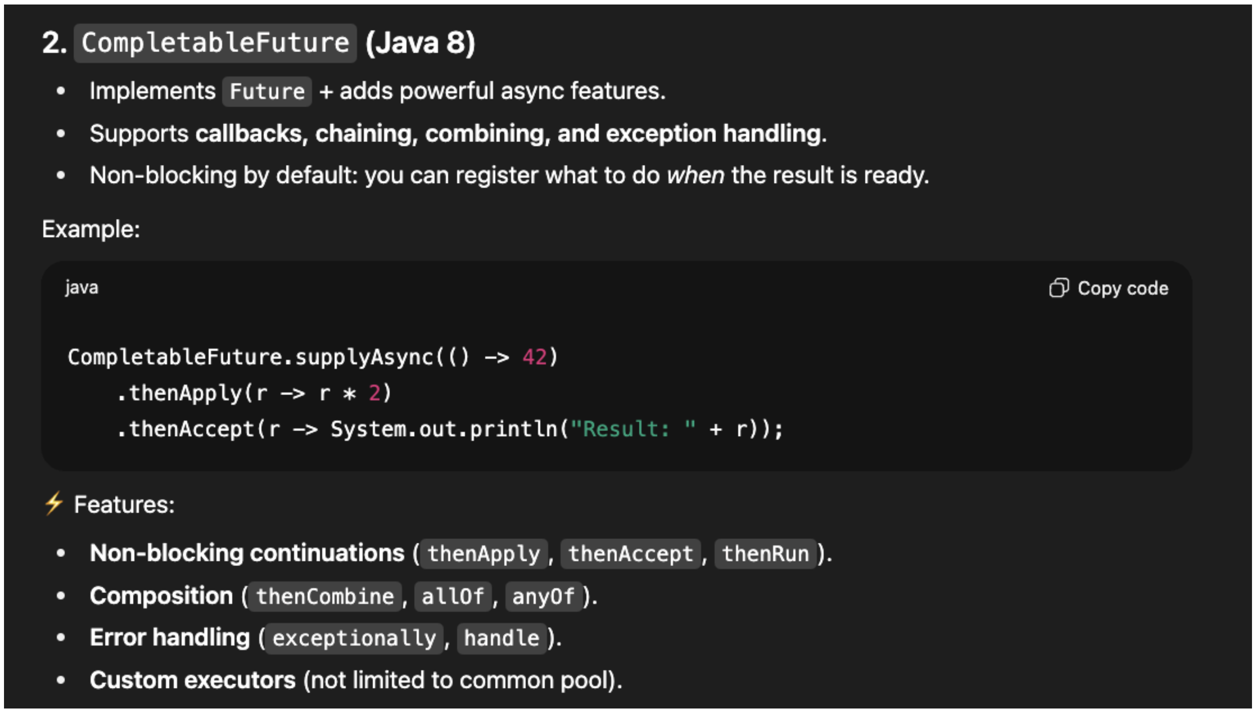


2. `CompletableFuture` (Java 8)

- Implements `Future` + adds powerful async features.
- Supports **callbacks, chaining, combining, and exception handling**.
- Non-blocking by default: you can register what to do *when* the result is ready.

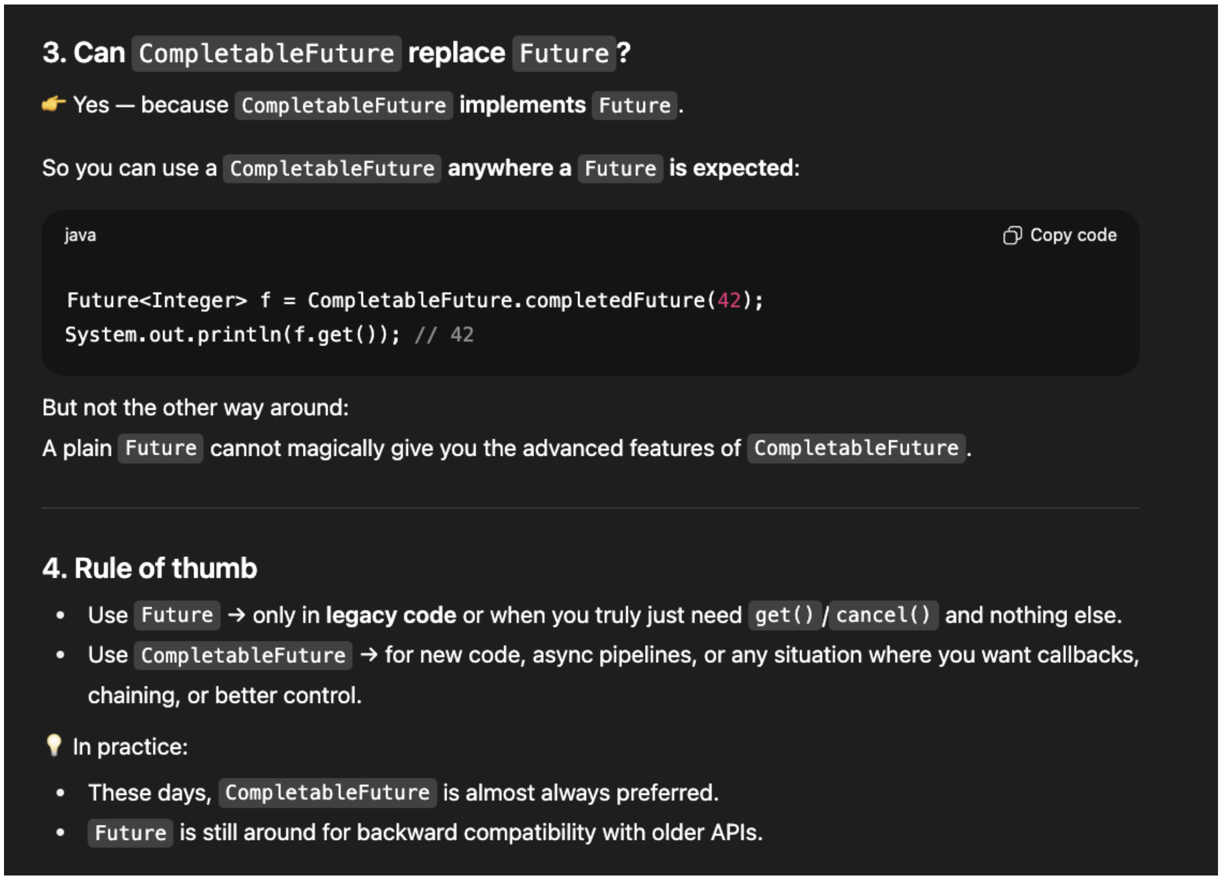
Example:

java Copy code

```java
CompletableFuture.supplyAsync(() -> 42)
    .thenApply(r -> r * 2)
    .thenAccept(r -> System.out.println("Result: " + r));
```

⚡ Features:

- **Non-blocking continuations** (`thenApply`, `thenAccept`, `thenRun`).
- **Composition** (`thenCombine`, `allOf`, `anyOf`).
- **Error handling** (`exceptionally`, `handle`).
- **Custom executors** (not limited to common pool).



3. Can `CompletableFuture` replace `Future`?

👉 Yes — because `CompletableFuture` **implements** `Future`.

So you can use a `CompletableFuture` **anywhere a** `Future` **is expected:**

java Copy code

```java
Future<Integer> f = CompletableFuture.completedFuture(42);
System.out.println(f.get()); // 42
```

But not the other way around:
A plain `Future` cannot magically give you the advanced features of `CompletableFuture`.

---

4. Rule of thumb

- Use `Future` → only in **legacy code** or when you truly just need `get()` / `cancel()` and nothing else.
- Use `CompletableFuture` → for new code, async pipelines, or any situation where you want callbacks, chaining, or better control.

💡 In practice:

- These days, `CompletableFuture` is almost always preferred.
- `Future` is still around for backward compatibility with older APIs.

1. Thread do not return value, Future return value