# **CompletableFuture in Java**

CompletableFuture provides a powerful and flexible way to write asynchronous, non-blocking code. It was introduced in Java 8 and has become popular due to its ease of use and ability to handle complex asynchronous workflows

### **What is CompletableFuture?**

CompletableFuture is a class in [java.util.concurrent](https://www.geeksforgeeks.org/java-util-concurrent-package/) package that implements the Future and CompletionStage Interface. It represents a future result of an asynchronous computation. It can be thought of as a container that holds the result of an asynchronous operation that is being executed in a different thread. It provides a number of methods to perform various operations on the result of the async computation.

### **Creating a CompletableFuture**

To create an instance of CompletableFuture, we can use the static method **supplyAsync** provided by CompletableFuture class which takes Supplier as an argument. Supplier is a Functional Interface that takes no value and returns a result.

* Java

| **import** java.util.concurrent.\*;    **class** GFG {  **public** **static** **void** main(String[] args) **throws** Exception  {  CompletableFuture<String> greetingFuture  = CompletableFuture.supplyAsync(() -> {  // some async computation  **return** "Hello from CompletableFuture";  });    System.out.println(greetingFuture.get());  }  } |
| --- |

**Output:**

Hello from CompletableFuture

This creates a CompletableFuture that will execute the lambda function passed to **supplyAsync** in a separate thread. And after the execution, the result lambda function is returned by CompletableFuture Object

### **Composing CompletableFuture**

One of the powerful features of CompletableFuture is its ability to compose multiple asynchronous operations. We can use methods like **thenApply**, **thenCombine**, **thenCompose** to perform operations on the result of one CompletableFuture and create a new CompletableFuture as a result.

* Java

| /\*package whatever //do not write package name here \*/    **import** java.util.concurrent.\*;    **class** GFG {  **public** **static** **void** main(String[] args) **throws** Exception  {  CompletableFuture<String> helloFuture  = CompletableFuture.supplyAsync(() -> "Hello");  CompletableFuture<String> greetingFuture  = CompletableFuture.supplyAsync(() -> "World");    CompletableFuture<String> combinedFuture  = helloFuture.thenCombine(  greetingFuture, (m1, m2) -> m1 + " " + m2);    System.out.println(combinedFuture.get());  }  } |
| --- |

**Output:**

Hello World

This creates two instances of CompletableFuture that return “hello” and “world”. And using **thenCombine**, the result of both the CompletableFutures are concatenated and returned as a final result.

### **Handling Exception in CompletableFuture**

CompletableFuture provides methods like **exceptionally** and **handle** to handle exceptions and errors that might happen during asynchronous computation and provide a fallback value or perform some alternative operation.

* Java

| **import** java.util.concurrent.\*;    **class** GFG {  **public** **static** **void** main(String[] args) **throws** Exception  {  CompletableFuture<Integer> resultFuture  // java.lang.ArithmeticException: / by zero  = CompletableFuture.supplyAsync(() -> 10 / 0)  .exceptionally(ex -> 0);    // 0 - returned by exceptionally block  System.out.println(resultFuture.get());  }  } |
| --- |

**Output:**

0

Inside **supplyAsync**, when 10 is divided by 0, It will throw ArithmeticException and control will go to **exceptionally** block and which in turn returns 0.

### **Conclusion**

In summary, CompletableFuture provides a powerful and flexible way to write asynchronous, non-blocking code in Java. We can use it to compose multiple asynchronous operations, handle errors and exceptions, and combine multiple CompletableFutures into one. By using CompletableFuture, we can write more efficient and scalable code that can take advantage of multi-core processors and handle complex asynchronous workflows.