

CompletableFuture



What is CompletableFuture?



Why CompletableFuture ?



How to create CompletableFuture



How to use runAsync and supplyAsyn method with usecase

What is CompletableFuture

CompletableFuture : A new era of asynchronous programming

- Using Asynchronous programming you can write non blocking code where concurrently you can run N no of task in separate thread without blocking main thread .
- When the task is complete, it notifies to the main thread (whether the task was completed or failed).



Why CompletableFuture ?

- There are different ways to implement asynchronous programming in Java using any of the below mechanisms for example you can use Futures, ExecutorService, Callback interfaces, Thread Pools, etc.
1. It cannot be manually completed
 2. Multiple Futures cannot be chained together
 3. We can not combine multiple Futures together
 4. No Proper Exception Handling Mechanism

```
7 > public class WhyNotFuture {  
8  
9 >     public static void main(String[] args) {  
10  
11         ExecutorService service=Executors.newFixedThreadPool( nThreads: 10 );  
12         serv  
13         v service ExecutorService  
14         } service.submit(Callable<T> task) 12% Future<T>  
15         } service.execute(Runnable command) 10% void  
16         } service.shutdown() 6% void  
17         P Press Enter to insert, Tab to replace Next Tip  
18         try {  
19             TimeUnit.MINUTES.sleep(min);  
20         } catch (InterruptedException e) {  
21             e.printStackTrace();  
22         }  
23     }  
}
```

The screenshot shows the IntelliJ IDEA interface with a tooltip for the `cancel` method of a `Future` object. The tooltip provides information about the method's parameters and return type. The code in the editor is as follows:

```
public static void main(String[] args) throws ExecutionException, InterruptedException {  
    ExecutorService service=Executors.newFixedThreadPool( nThreads: 10 );  
    Future<List<Integer>> future = service.submit(() -> {  
        //your doing api call  
        System.out.println("Thread : "+Thread.currentThread().getName());  
        delay( min: 1 );  
        return Arrays.asList(1, 2, 3, 4);  
    });  
  
    List<Integer> list = future.get();  
    fut  
    v future Future<List<Integer>>  
    b future.get() 26% List<Integer>  
    c future.cancel(boolean mayInterruptIfR... boolean  
    Press Ctrl+Space to see non-imported classes Next Tip
```

There is no method to complete execution forcefully.

The screenshot shows the IntelliJ IDEA interface with the output of the program in the Run tab. The output shows the list [1, 2, 3, 4] printed to the console. The code in the editor is as follows:

```
public static void main(String[] args) throws ExecutionException, InterruptedException {  
    ExecutorService service=Executors.newFixedThreadPool( nThreads: 10 );  
    Future<List<Integer>> future = service.submit(() -> {  
        //your doing api call  
        System.out.println("Thread : "+Thread.currentThread().getName());  
        return Arrays.asList(1, 2, 3, 4);  
    });  
  
    List<Integer> list = future.get();  
    |  
    System.out.println(list);  
}
```

We can not pass the output of future to another thread (future) for execution

```

Future<List<Integer>> future1 = service.submit(() -> {
    //your doing api call
    System.out.println("Thread : "+Thread.currentThread().getName());
    return Arrays.asList(1, 2, 3, 4);
});

Future<List<Integer>> future2 = service.submit(() -> {
    //your doing api call
    System.out.println("Thread : "+Thread.currentThread().getName());
    return Arrays.asList(1, 2, 3, 4);
});

Future<List<Integer>> future3 = service.submit(() -> {
    //your doing api call
    System.out.println("Thread : "+Thread.currentThread().getName());
    return Arrays.asList(1, 2, 3, 4);
});

```

future1+future2+future3

When we have more than one future object we can combine them together..

```

completablefutureexample src main java com javatechie async WhyNotFuture
WhyNotFuture.java
12 Future<List<Integer>> future1 = service.submit(() -> {
13     //your doing api call
14     System.out.println("Thread : "+Thread.currentThread().getName());
15     System.out.println(10/0);
16     return Arrays.asList(1, 2, 3, 4);
17 });
18
19
20
WhyNotFuture > main() > 0 -> ...
Run: WhyNotFuture
1: "C:\Program Files\Java\jdk1.8.0_181\bin\java.exe" ...
Thread : pool-1-thread-1
Exception in thread "main" java.util.concurrent.ExecutionException: java.lang.ArithmetricException: / by zero
at java.util.concurrent.FutureTask.report(FutureTask.java:122)
at java.util.concurrent.FutureTask.get(FutureTask.java:192)

```

No exception handling for future object.

```

23
24     CompletableFuture<String> completableFuture=new CompletableFuture<>();
25     completableFuture.get();
26     completableFuture.complete(value: "return some dummy data....");
27
28 }

```

completableFuture.complete method to force the close the thread with default message / response if it is taking more time.

runAsync() Vs supplyAsync()

If we want to run some background task asynchronously and **do not want to return** anything from that task, then use CompletableFuture.runAsync() method. It takes a Runnable object and returns CompletableFuture<Void>.

1. CompletableFuture.runAsync(Runnable)
2. CompletableFuture.runAsync(Runnable, Executor)

If we want to run some background task asynchronously and **want to return** anything from that task, we should use CompletableFuture.supplyAsync(). It takes a Supplier<T> and returns CompletableFuture<T> where T is the type of the value obtained by calling the given supplier.

1. CompletableFuture.supplyAsync(Supplier<T>)
2. CompletableFuture.supplyAsync(Supplier<T>, Executor)

```
public class RunAsyncDemo {  
  
    public void saveEmployees(File jsonFile) throws ExecutionException, InterruptedException {  
        ObjectMapper mapper = new ObjectMapper();  
        CompletableFuture<Void> runAsyncFuture = CompletableFuture.runAsync(new Runnable() {  
  
            @Override  
            public void run() {  
                try {  
                    List<Employee> employees = mapper  
                        .readValue(jsonFile, new TypeReference<List<Employee>>() {});  
                    //write logic to save list of employee to database  
                    //repository.saveAll(employees);  
                    System.out.println("Thread : " + Thread.currentThread().getName());  
                    employees.stream().forEach(System.out::println);  
  
                    System.out.println("Thread : " + Thread.currentThread().getName());  
                    employees.stream().forEach(System.out::println);  
                } catch (IOException e) {  
                    e.printStackTrace();  
                }  
            }  
        });  
  
        return runAsyncFuture.get();  
    }  
  
    public static void main(String[] args) throws ExecutionException, InterruptedException {  
        RunAsyncDemo runAsyncDemo = new RunAsyncDemo();  
        runAsyncDemo.saveEmployees(new File( pathname: "employees.json"));  
    }  
}
```

```
"C:\Program Files\Java\jdk1.8.0_181\bin\java.exe" ...
Thread : ForkJoinPool.commonPool-worker-1
Employee(employeeId=59-385-1088, firstName=Zacharias, lastName=Snavely)
Employee(employeeId=73-274-6476, firstName=Kyle, lastName=Jenner)
Employee(employeeId=85-939-9584, firstName=Axe, lastName=Kinsella)
Employee(employeeId=08-180-8292, firstName=Robinet, lastName=Harmsen)
```

Global thread pool

```
public Void saveEmployeesWithCustomExecutor(File jsonFile) throws ExecutionException, InterruptedException {
    ObjectMapper mapper = new ObjectMapper();
    Executor executor = Executors.newFixedThreadPool(5);
    CompletableFuture<Void> runAsyncFuture = CompletableFuture.runAsync(
        () -> {
            try {
                List<Employee> employees = mapper
                    .readValue(jsonFile, new TypeReference<List<Employee>>() {
                });
                //write logic to save List of employee to database
                //repository.saveAll(employees);
                System.out.println("Thread : " + Thread.currentThread().getName());
                System.out.println(employees.size());
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    ),executor);

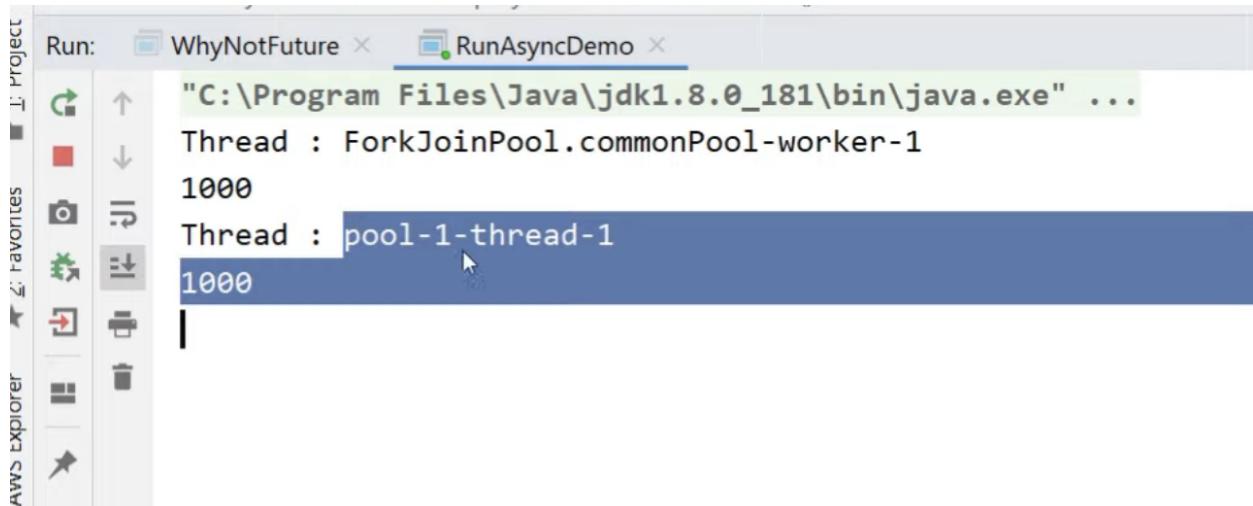
    return runAsyncFuture.get();
}
```

Custom thread pool

```
    }
},executor);

return runAsyncFuture.get();
}

public static void main(String[] args) throws ExecutionException, InterruptedException {
    RunAsyncDemo runAsyncDemo = new RunAsyncDemo();
    runAsyncDemo.saveEmployees(new File( pathname: "employees.json"));
    runAsyncDemo.saveEmployeesWithCustomExecutor(new File( pathname: "employees.json"));
}
```



If our thread is blocked then complete it manually

```
runAsyncFuture.toCompletableFuture();
```

Now let us see supplyAsync



```
SupplyAsyncDemo.java  Supplier.java  EmployeeDatabase.java
8 import java.util.concurrent.ExecutionException;
9 import java.util.function.Supplier;
10
11 public class SupplyAsyncDemo {
12
13     public List<Employee> getEmployees() throws ExecutionException, InterruptedException {
14         CompletableFuture<List<Employee>> listCompletableFuture = CompletableFuture
15             .supplyAsync(() -> {
16                 System.out.println("Executed by : " + Thread.currentThread().getName());
17                 return EmployeeDatabase.fetchEmployees();
18             });
19         return listCompletableFuture.get();
20     }
21
22
23     public static void main(String[] args) throws ExecutionException, InterruptedException {
24         SupplyAsyncDemo supplyAsyncDemo = new SupplyAsyncDemo();
25         List<Employee> employees = supplyAsyncDemo.getEmployees();
26         employees.stream().forEach(System.out::println);
27     }
28 }
```

```
WhyNotFuture  SupplyAsyncDemo
"C:\Program Files\Java\jdk1.8.0_181\bin\java.exe" ...
Executed by : ForkJoinPool.commonPool-worker-1
Employee(employeeId=59-385-1088, firstName=Zacharias, lastName=Kline)
Employee(employeeId=73-274-6476, firstName=Kyle, lastName=Hausner)
```

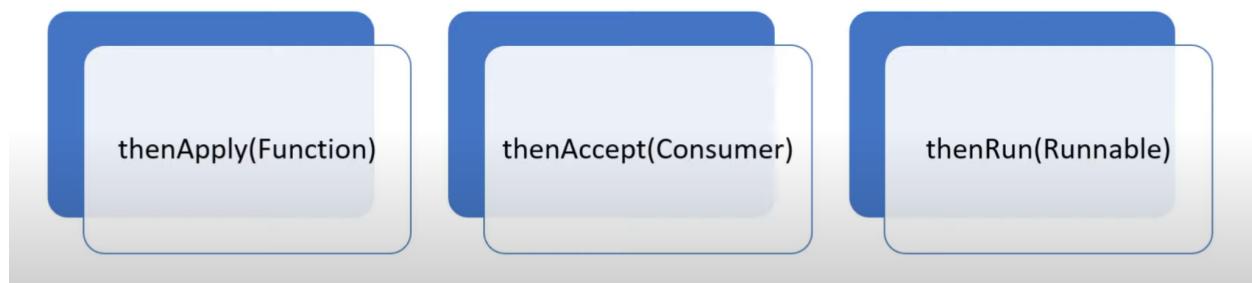
Global thread pool

With overload method where it accept executor

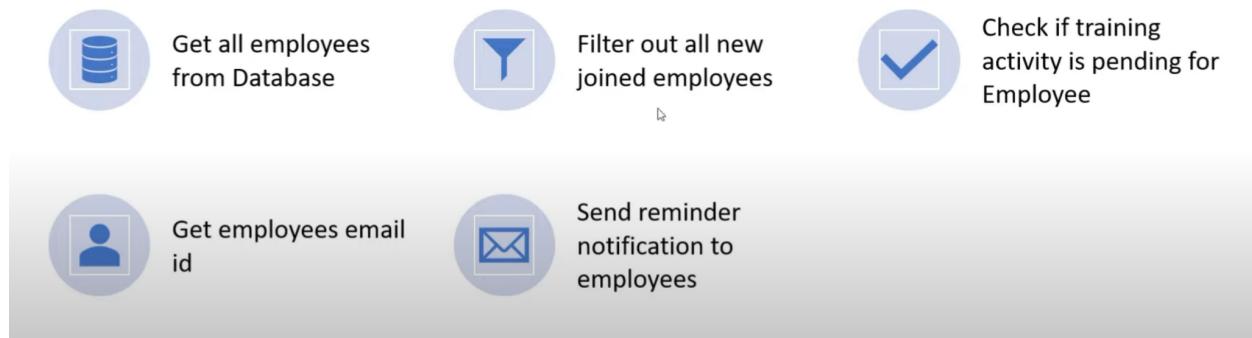
```
SupplyAsyncDemo.java  Supplier.java  EmployeeDatabase.java
3 public class SupplyAsyncDemo {
4
5     public List<Employee> getEmployees() throws ExecutionException, InterruptedException {
6         Executor executor = Executors.newCachedThreadPool();
7         CompletableFuture<List<Employee>> listCompletableFuture = CompletableFuture
8             .supplyAsync(() -> {
9                 System.out.println("Executed by : " + Thread.currentThread().getName());
10                return EmployeeDatabase.fetchEmployees();
11            },executor);
12        return listCompletableFuture.get();
13    }
14
15
16    public static void main(String[] args) throws ExecutionException, InterruptedException {
17        SupplyAsyncDemo supplyAsyncDemo = new SupplyAsyncDemo();
18        List<Employee> employees = supplyAsyncDemo.getEmployees();
19        employees.stream().forEach(System.out::println);
20    }
21 }
```

```
SupplyAsyncDemo > getEmployees() > () -> {...}
Run: WhyNotFuture <-- SupplyAsyncDemo <-- ...
"C:\Program Files\Java\jdk1.8.0_181\bin\java.exe" ...
Executed by : pool-1-thread-1
Employee(employeeId=59-385-1088, firstName=Zacharias, lastName=Smith)
Employee(employeeId=73-274-6476, firstName=Kyle, lastName=Friis)
```

thenApply(), thenAccept() & thenRun()



Use case : Employee training reminder



SupplyAsyncDemo

SupplyAsyncDemo.java employees.json CompletableFuture.java

```
1 [  
2 {  
3     "employeeId": "59-385-1088",  
4     "firstName": "Zacharias",  
5     "lastName": "Schwerin",  
6     "email": "zschwerin0@altervista.org",  
7     "gender": "Male",  
8     "newJoiner": "TRUE",  
9     "learningPending": "FALSE",  
10    "salary": 101146,  
11    "rating": 0  
12 },  
13 {
```

EmployeeReminderService.java CompletableFuture.java Consumer.java Function.java employees.json EmployeeDatabase.java

```
9 public class EmployeeReminderService {  
10  
11  
12     public CompletableFuture<Void> sendReminderToEmployee() {  
13         CompletableFuture<Void> voidCompletableFuture = CompletableFuture.supplyAsync(() -> {  
14             System.out.println("fetchEmployee : " + Thread.currentThread().getName());  
15             return EmployeeDatabase.fetchEmployees();  
16         }).thenApply((employees) -> {  
17             System.out.println("filter new joiner employee : " + Thread.currentThread().getName());  
18             return employees.stream()  
19                 .filter(employee -> "TRUE".equals(employee.getNewJoiner()))  
20                 .collect(Collectors.toList());  
21         }).thenApply((employees) -> {  
22             System.out.println("filter training not complete employee : " + Thread.currentThread().getName());  
23             return employees.stream()  
24                 .filter(employee -> "TRUE".equals(employee.getLearningPending()))  
25                 .collect(Collectors.toList());  
26         }).thenApply((employees) -> {  
27             System.out.println("get emails : " + Thread.currentThread().getName());  
28             return employees.stream().map(Employee::getEmail).collect(Collectors.toList());  
29         }).thenAccept((emails) -> {  
30             System.out.println("get emails : " + Thread.currentThread().getName());  
31         });  
32     }  
33 }
```

```

29         return employees.stream().map(Employee::getEmail).collect(Collectors.toList());
30     }).thenAccept((emails) -> {
31         System.out.println("get emails : " + Thread.currentThread().getName());
32         emails.forEach(EmployeeReminderService::sendEmail);
33     });
34     return voidCompletableFuture;
35 }
36
37
38     public static void sendEmail(String email) {
39         System.out.println("sending training reminder email to : " + email);
40     }
41
42 >   public static void main(String[] args) throws ExecutionException, InterruptedException {
43
44     EmployeeReminderService service=new EmployeeReminderService();
45     service.sendReminderToEmployee().get();
46 }
47
48 }
```

In everywhere global thread is being used

```

EmployeeReminderService ×
↑ "C:\Program Files\Java\jdk1.8.0_181\bin\java.exe" ...
↓ fetchEmployee : ForkJoinPool.commonPool-worker-1
≡ filter new joiner employee : ForkJoinPool.commonPool-worker-1
≡ filter training not complete employee : ForkJoinPool.commonPool-worker-1
≡ get emails : ForkJoinPool.commonPool-worker-1
≡ get emails : ForkJoinPool.commonPool-worker-1
≡ sending training reminder email to : jbraybrooke6@prnewswire.com
≡ sending training reminder email to : ebaverstock7@ehow.com
≡ sending training reminder email to : eoultrame@tinyurl.com
≡ sending training reminder email to : lkiddk@shareasale.com
≡ sending training reminder email to : egilhoolieo@salon.com
```

Now thenApplyAsync and thenAcceptAccept methods in place of thenApply and thenAccept now we see new threads are assigned from the global pool. The pool assignment is based on jvm thread scheduler. We can predict thread assignments..

```
public CompletableFuture<Void> sendReminderToEmployee() {
    CompletableFuture<Void> voidCompletableFuture = CompletableFuture.supplyAsync(() -> {
        System.out.println("fetchEmployee : " + Thread.currentThread().getName());
        return EmployeeDatabase.fetchEmployees();
    }).thenApplyAsync((employees) -> {
        System.out.println("filter new joiner employee : " + Thread.currentThread().getName());
        return employees.stream()
            .filter(employee -> "TRUE".equals(employee.getNewJoiner()))
            .collect(Collectors.toList());
    }).thenApplyAsync((employees) -> {
        System.out.println("filter training not complete employee : " + Thread.currentThread().getName());
        return employees.stream()
            .filter(employee -> "TRUE".equals(employee.getLearningPending()))
            .collect(Collectors.toList());
    }).thenApplyAsync((employees) -> {
        System.out.println("get emails : " + Thread.currentThread().getName());
        return employees.stream().map(Employee::getEmail).collect(Collectors.toList());
    }).thenAcceptAsvnc((emails) -> {
        EmployeeReminderService > sendReminderToEmployee() > (emails) -> {...}
    });
}
```

The screenshot shows the Java code for the `sendReminderToEmployee` method with several lines highlighted in yellow, indicating they were executed by different threads. The terminal output below shows the printed messages and email addresses sent, each preceded by the thread name (e.g., `ForkJoinPool.commonPool-worker-1`).

```
EmployeeReminderService.java × CompletableFuture.java × Consumer.java × Function.java × employees.json × EmployeeDatabase.java ×
30     }).thenAcceptAsvnc((emails) -> {
31         System.out.println("send email : " + Thread.currentThread().getName());
32         emails.forEach(EmployeeReminderService::sendEmail);
33     );
34     return voidCompletableFuture;
35 }
36 EmployeeReminderService > sendReminderToEmployee() > (emails) -> {...}

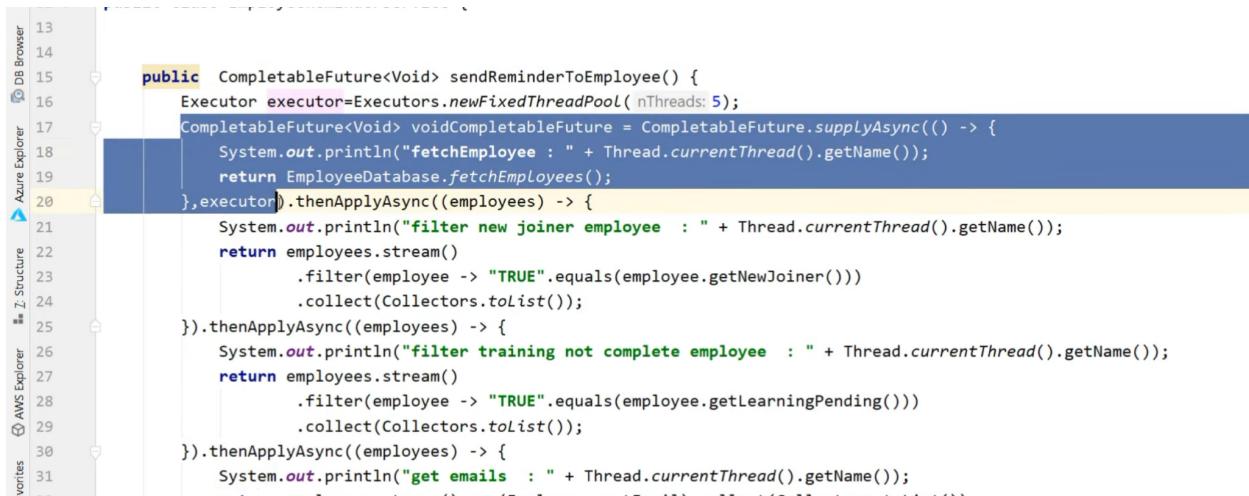
Run: EmployeeReminderService ×
▶ C:\Program Files\Java\jdk1.8.0_181\bin\java.exe" ...
fetchEmployee : ForkJoinPool.commonPool-worker-1
filter new joiner employee : ForkJoinPool.commonPool-worker-1
filter training not complete employee : ForkJoinPool.commonPool-worker-1
get emails : ForkJoinPool.commonPool-worker-1
send email : ForkJoinPool.commonPool-worker-1
sending training reminder email to : jbraybrooke6@prnewswire.com
sending training reminder email to : ebaverstock7@ehow.com
sending training reminder email to : eoultrame@tinyurl.com
sending training reminder email to : lkiddk@shareasale.com
```

When run again and again we can notice different threads are assigned.

This screenshot shows the same code and terminal output as the previous one, but with some lines highlighted in blue, indicating they were executed by the same thread (worker-2). The terminal output shows the printed messages and email addresses sent.

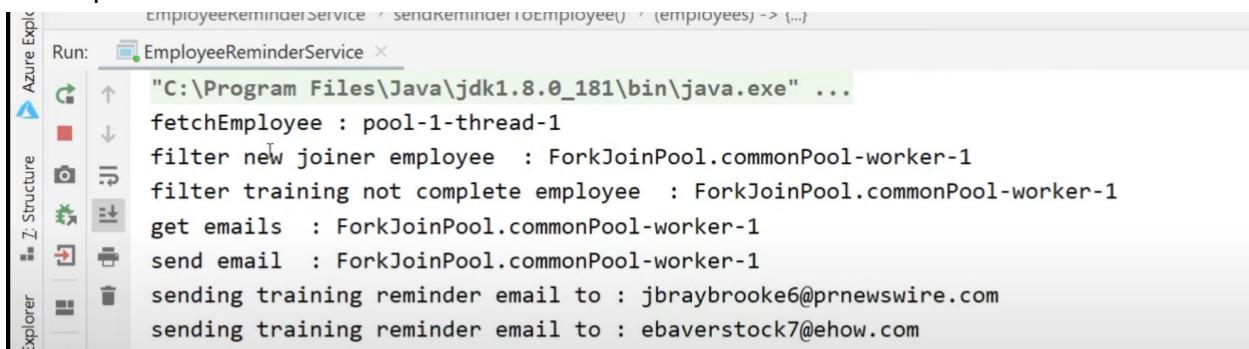
```
EmployeeReminderService ×
C:\Program Files\Java\jdk1.8.0_181\bin\java.exe" ...
fetchEmployee : ForkJoinPool.commonPool-worker-1
filter new joiner employee : ForkJoinPool.commonPool-worker-1
filter training not complete employee : ForkJoinPool.commonPool-worker-1
get emails : ForkJoinPool.commonPool-worker-2
get emails : ForkJoinPool.commonPool-worker-2
sending training reminder email to : jbraybrooke6@prnewswire.com
sending training reminder email to : ebaverstock7@ehow.com
sending training reminder email to : eoultrame@tinyurl.com
sending training reminder email to : lkiddk@shareasale.com
```

To see this let us introduce custom executor



```
13
14
15     public CompletableFuture<Void> sendReminderToEmployee() {
16         Executor executor=Executors.newFixedThreadPool( nThreads: 5);
17         CompletableFuture<Void> voidCompletableFuture = CompletableFuture.supplyAsync(() -> {
18             System.out.println("fetchEmployee : " + Thread.currentThread().getName());
19             return EmployeeDatabase.fetchEmployees();
20         },executor).thenApplyAsync((employees) -> {
21             System.out.println("filter new joiner employee : " + Thread.currentThread().getName());
22             return employees.stream()
23                 .filter(employee -> "TRUE".equals(employee.getNewJoiner()))
24                 .collect(Collectors.toList());
25         }).thenApplyAsync((employees) -> {
26             System.out.println("filter training not complete employee : " + Thread.currentThread().getName());
27             return employees.stream()
28                 .filter(employee -> "TRUE".equals(employee.getLearningPending()))
29                 .collect(Collectors.toList());
30         }).thenApplyAsync((employees) -> {
31             System.out.println("get emails : " + Thread.currentThread().getName());
```

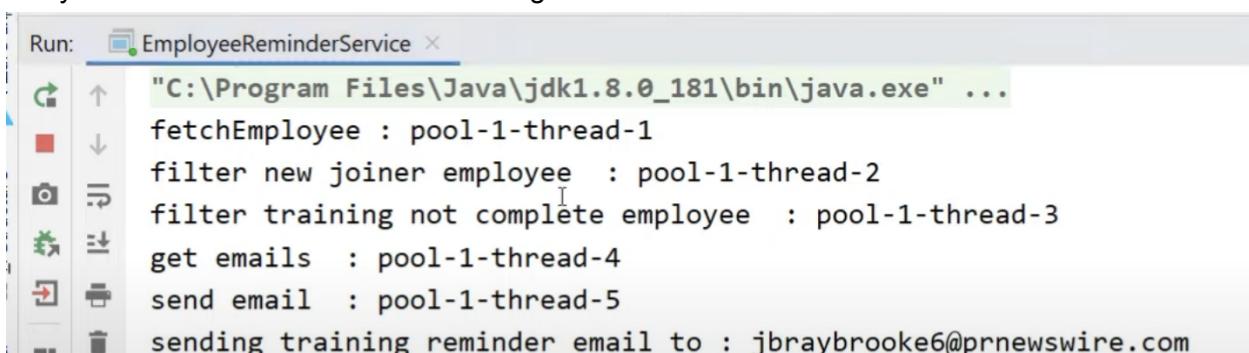
For the highlighted function the thread is assigned from the custom executor rest from the global thread pool.



Run: EmployeeReminderService

```
"C:\Program Files\Java\jdk1.8.0_181\bin\java.exe" ...
fetchEmployee : pool-1-thread-1
filter new joiner employee : ForkJoinPool.commonPool-worker-1
filter training not complete employee : ForkJoinPool.commonPool-worker-1
get emails : ForkJoinPool.commonPool-worker-1
send email : ForkJoinPool.commonPool-worker-1
sending training reminder email to : jbraybrooke6@prnewswire.com
sending training reminder email to : ebaverstock7@ehow.com
```

Let us introduce the custom executor for all the functions we see below behavior where for every function new custom thread is assigned.

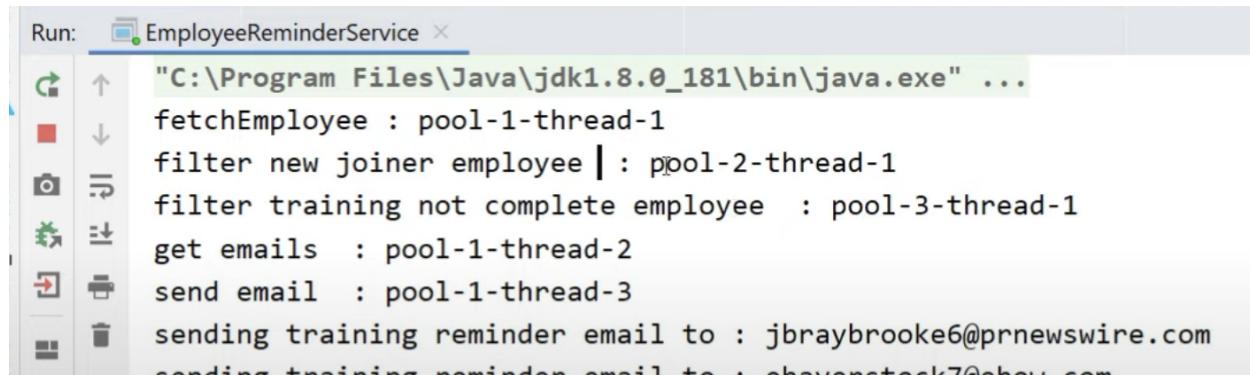


Run: EmployeeReminderService

```
"C:\Program Files\Java\jdk1.8.0_181\bin\java.exe" ...
fetchEmployee : pool-1-thread-1
filter new joiner employee : pool-1-thread-1
filter training not complete employee : pool-1-thread-1
get emails : pool-1-thread-1
send email : pool-1-thread-1
sending training reminder email to : jbraybrooke6@prnewswire.com
```

```
public CompletableFuture<Void> sendReminderToEmployee() {  
  
    Executor executor=Executors.newFixedThreadPool( nThreads: 5);  
    Executor executor1=Executors.newFixedThreadPool( nThreads: 5);  
    Executor executor2=Executors.newFixedThreadPool( nThreads: 5);
```

Let us create different custom thread pools and overload them and see the behavior



The screenshot shows a terminal window titled "EmployeeReminderService" with the command "C:\Program Files\Java\jdk1.8.0_181\bin\java.exe" ... entered. The log output is as follows:

```
fetchEmployee : pool-1-thread-1  
filter new joiner employee | : pool-2-thread-1  
filter training not complete employee : pool-3-thread-1  
get emails : pool-1-thread-2  
send email : pool-1-thread-3  
sending training reminder email to : jbraybrooke6@prnewswire.com  
sending training reminder email to : abhayonstock7@yahoo.com
```