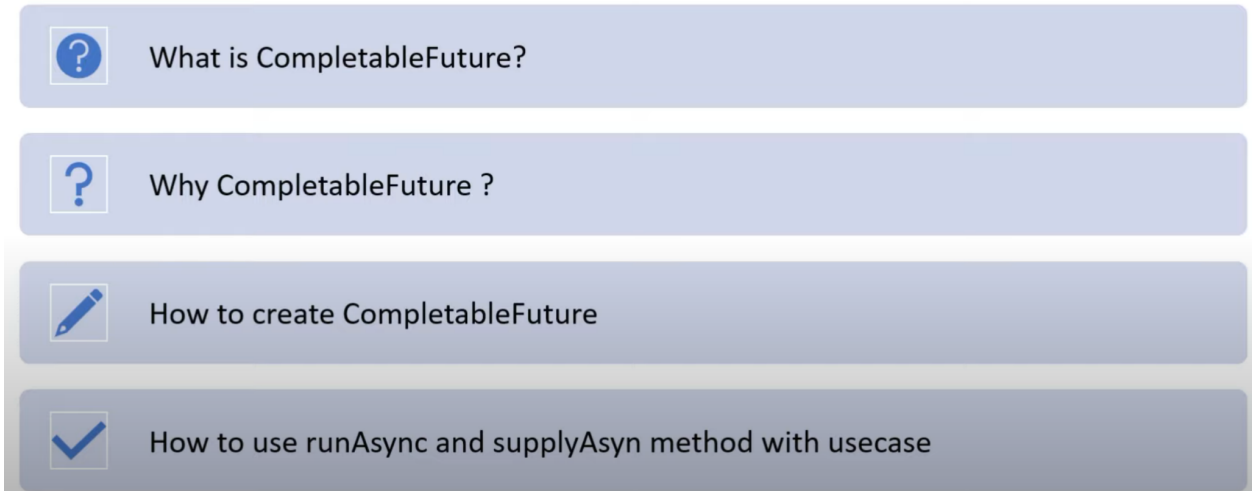
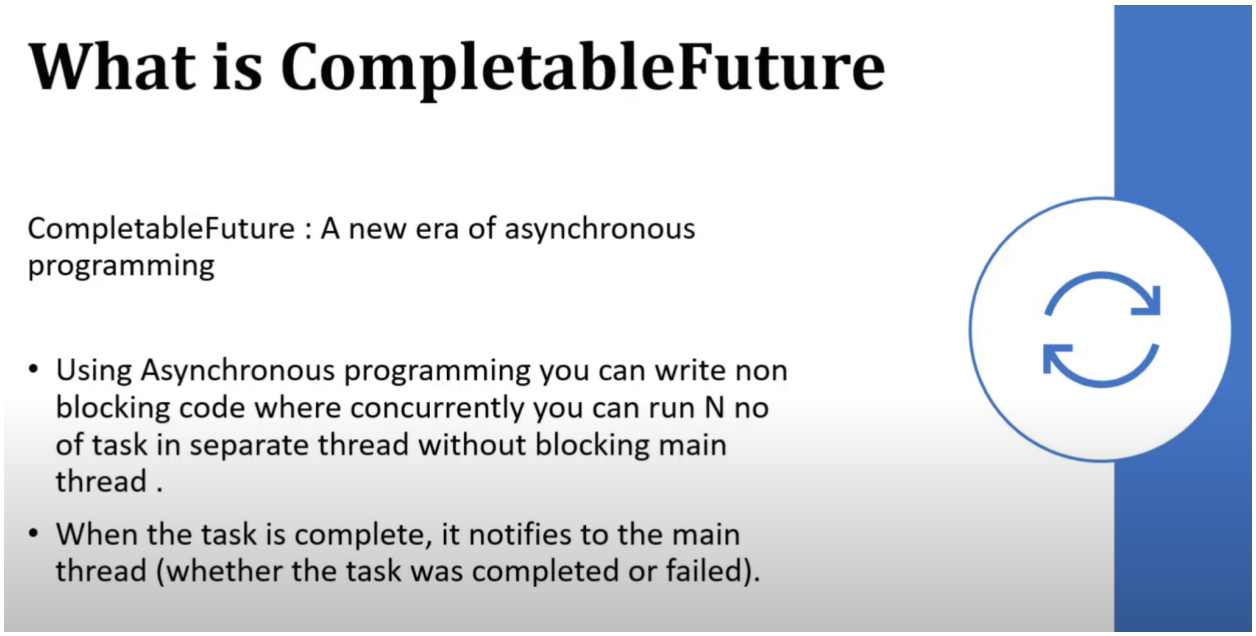
CompletableFeature





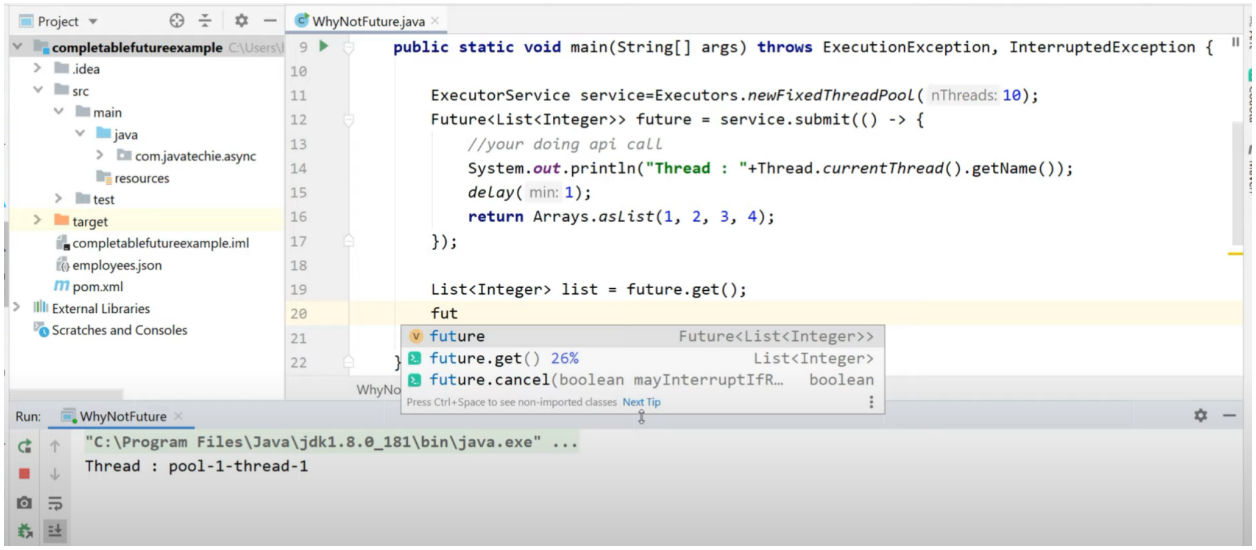
Why CompletableFuture ?

- 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.

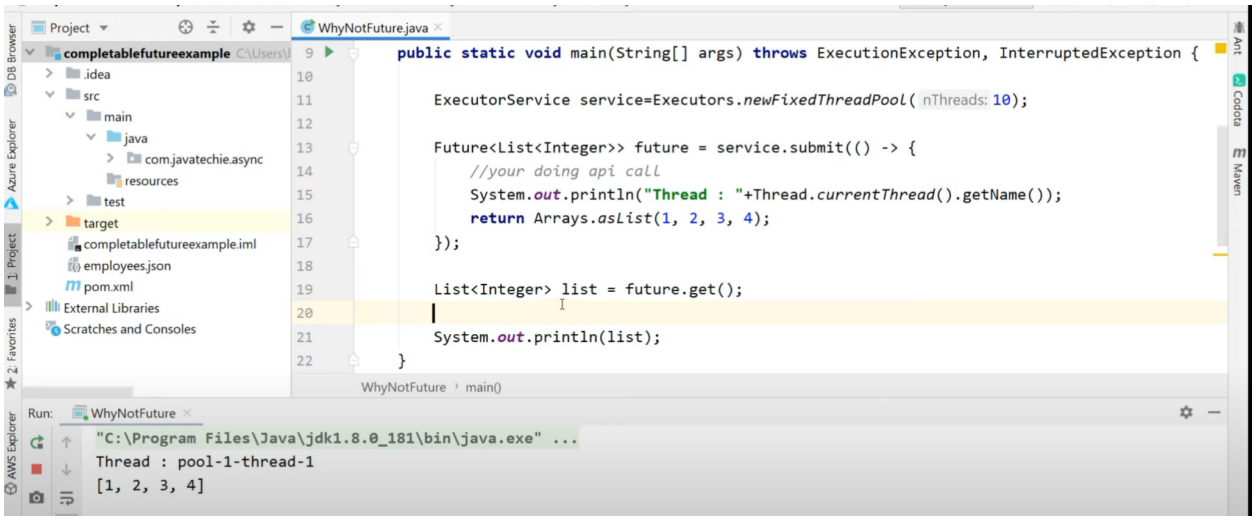
1. It cannot be manually completed
2. Multiple Futures cannot be chained together
3. We can not combine multiple Futures together
4. No Proper Exception Handling Mechanism

```
7  public class WhyNotFuture {
8
9      public static void main(String[] args) {
10
11         ExecutorService service=Executors.newFixedThreadPool( nThreads: 10);
12         serv
13
14     }
15
16     p
17         try {
18             TimeUnit.MINUTES.sleep(min);
19         } catch (InterruptedException e) {
20             e.printStackTrace();
21         }
22     }
23 }
```

```
service                                  ExecutorService
service.submit(Callable<T> task) 12%     Future<T>
service.execute(Runnable command) 10%    void
service.shutdown() 6%                    void
Press Enter to insert, Tab to replace  Next Tip
```



There is no method to complete execution forcefully.



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

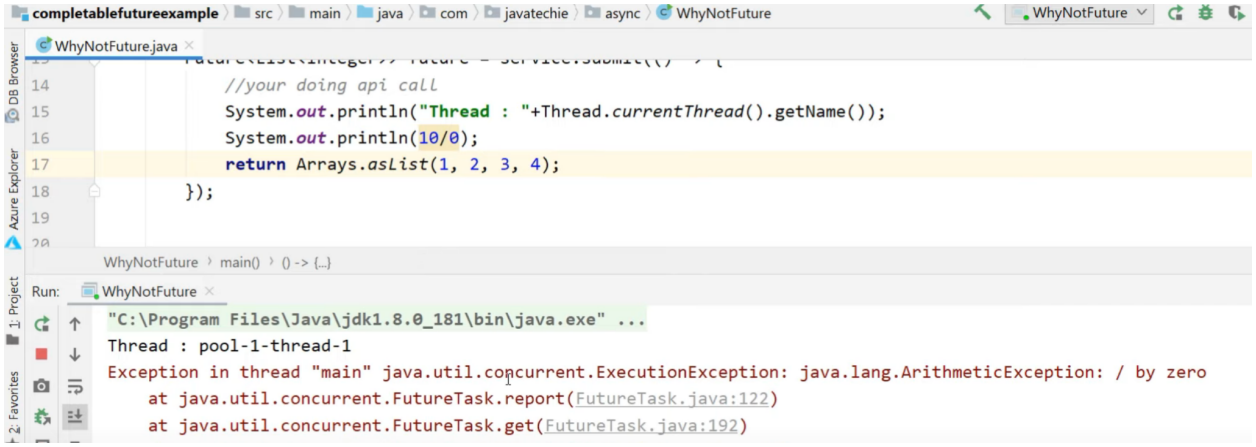
```
Future<List<Integer>> future1 = service.submit(() -> {
    //your doing api call
    System.out.println("Thread : "+Thread.currentThread().getName());
    return Arrays.asList(1, 2, 3, 4);
});

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..



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)

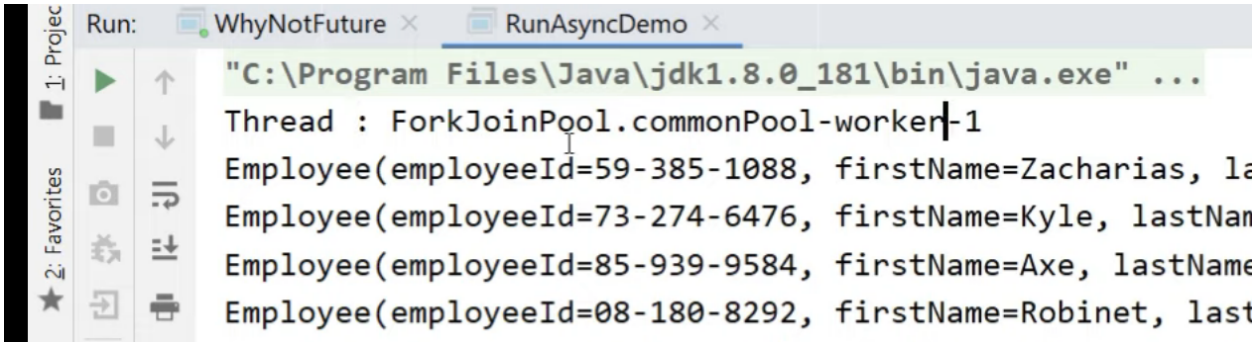
```
public class RunAsyncDemo {

    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 t save list of employee to database
                    //repository.saveAll(employees);
                    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"));
    }
}
```

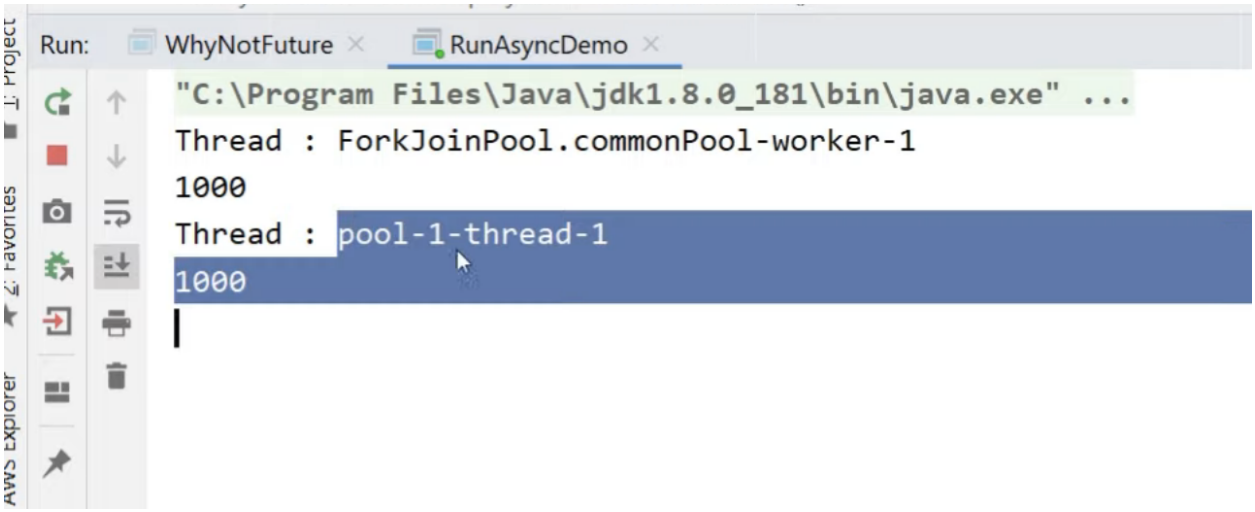


Global thread pool

```
41  public Void saveEmployeesWithCustomE(File jsonFile) throws ExecutionException, InterruptedException {
42      ObjectMapper mapper = new ObjectMapper();
43      Executor executor = Executors.newFixedThreadPool( nThreads: 5);
44      CompletableFuture<Void> runAsyncFuture = CompletableFuture.runAsync(
45              () -> {
46                  try {
47                      List<Employee> employees = mapper
48                              .readValue(jsonFile, new TypeReference<List<Employee>>() {
49                              });
50                      //write logic t save list of employee to database
51                      //repository.saveAll(employees);
52                      System.out.println("Thread : " + Thread.currentThread().getName());
53                      System.out.println(employees.size());
54                  } catch (IOException e) {
55                      e.printStackTrace();
56                  }
57              },executor);
58
59      return runAsyncFuture.get();
RunAsyncDemo › saveEmployeesWithCustom()
```

Custom thread pool

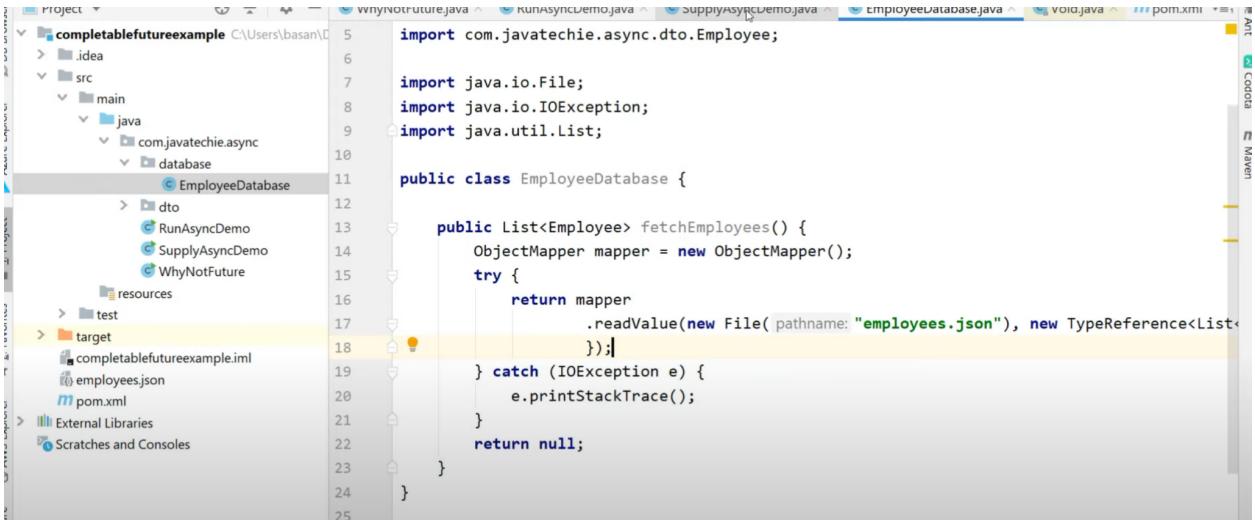
```
56                  }
57              },executor);
58
59      return runAsyncFuture.get();
60  }
61
62  public static void main(String[] args) throws ExecutionException, InterruptedException {
63      RunAsyncDemo runAsyncDemo = new RunAsyncDemo();
64      runAsyncDemo.saveEmployees(new File( pathname: "employees.json"));
65      runAsyncDemo.saveEmployeesWithCustomExecutor(new File( pathname: "employees.json"));
66
67  }
68  }
69
```



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 }
```

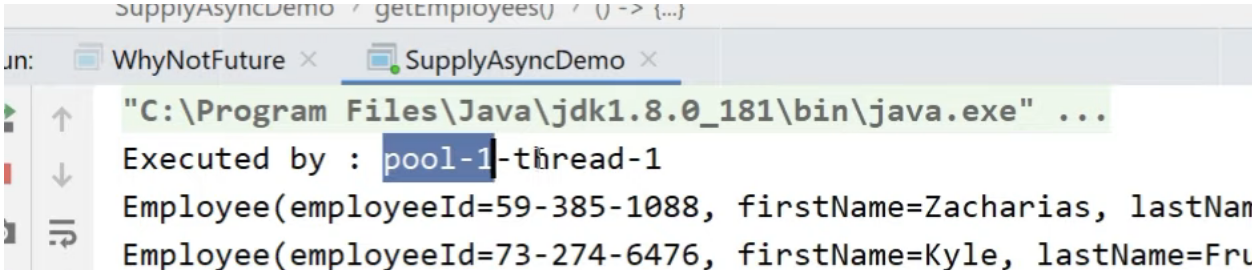
n: WhyNotFuture | SupplyAsyncDemo

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

Global thread pool

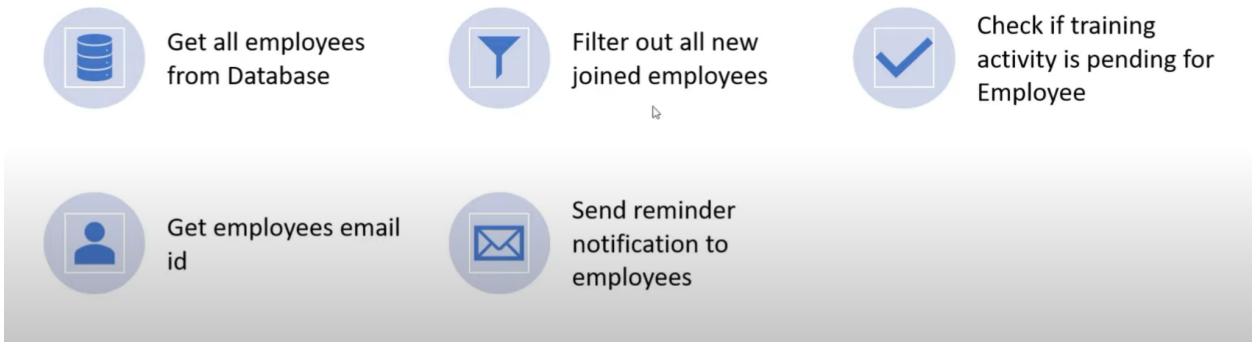
With overload method where it accept executor

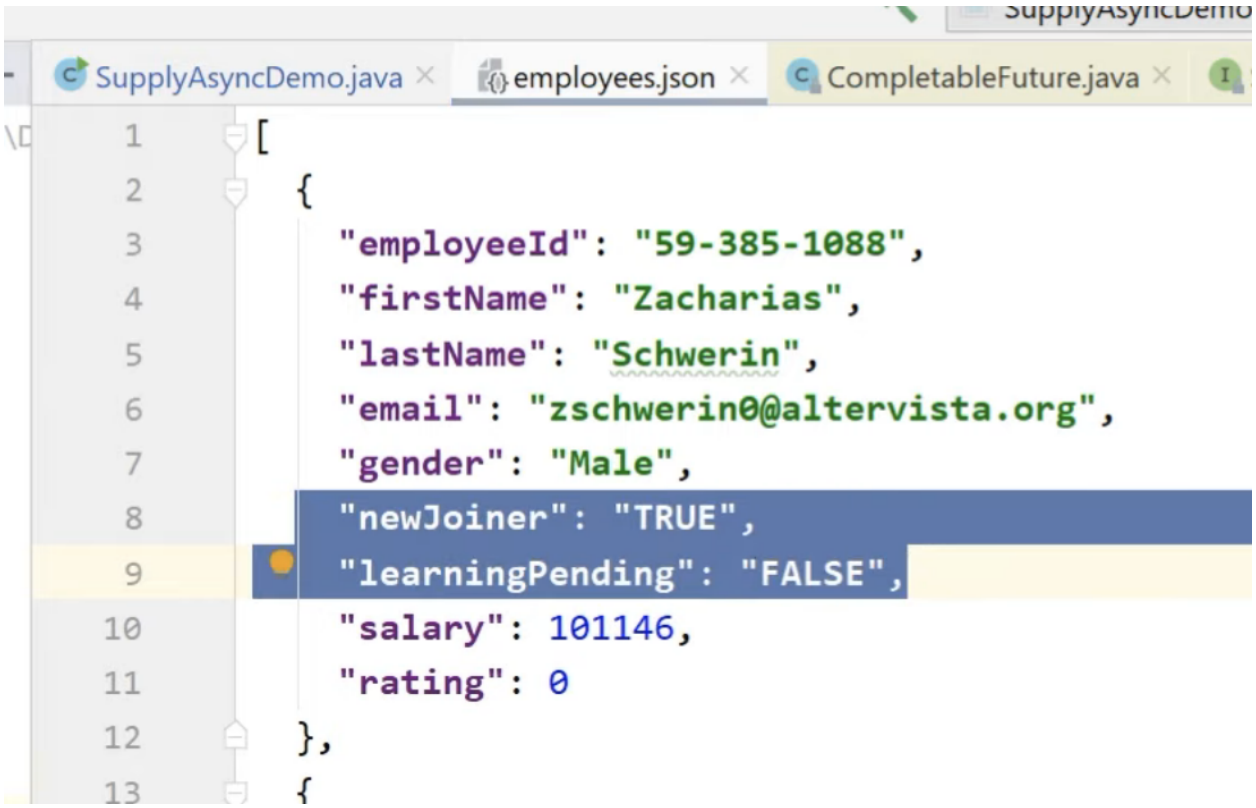
```
3  public class SupplyAsyncDemo {
4
5      public List<Employee> getEmployees() throws ExecutionException, InterruptedException {
6          Executor executor = Executors.newCachedThreadPool();
7          CompletableFuture<List<Employee>> listCompletableFuture = CompletableFuture
8                  .supplyAsync(() -> {
9                      System.out.println("Executed by : " + Thread.currentThread().getName());
0                      return EmployeeDatabase.fetchEmployees();
1                  },executor);
2          return listCompletableFuture.get();
3      }
4
5
6      public static void main(String[] args) throws ExecutionException, InterruptedException {
7          SupplyAsyncDemo supplyAsyncDemo = new SupplyAsyncDemo();
8          List<Employee> employees = supplyAsyncDemo.getEmployees();
9          employees.stream().forEach(System.out::println);
0      }
1  }
```

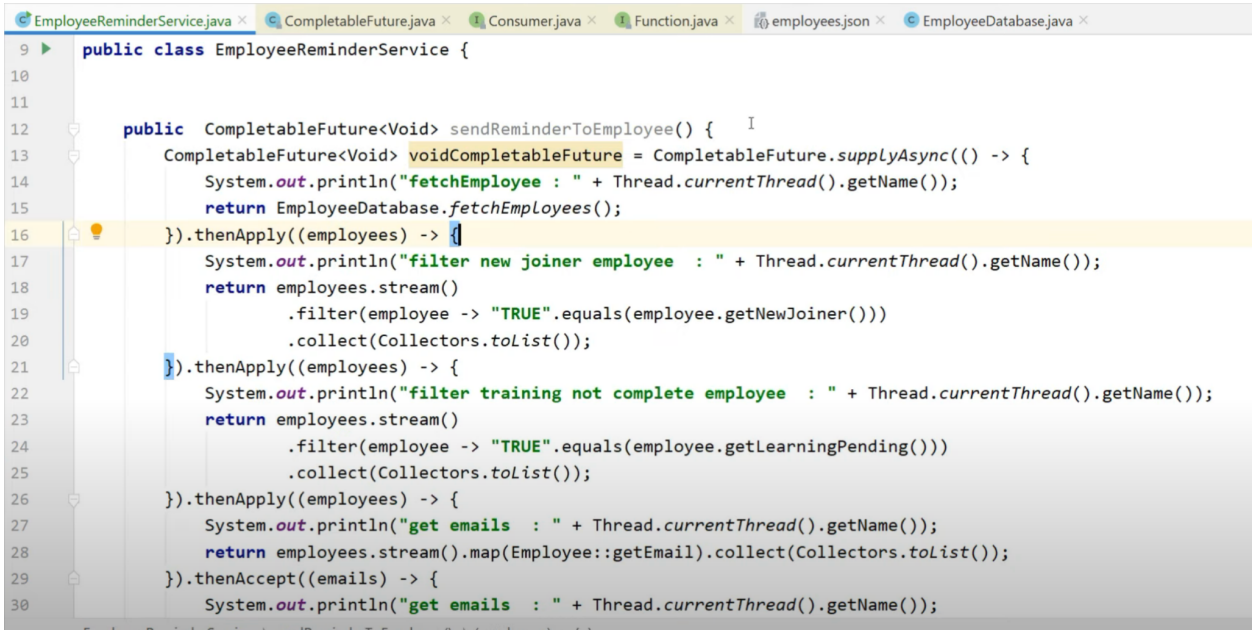


thenApply() , thenAccept() & thenRun()

Use case : Employee training reminder




SupplyAsyncDemo.java
employees.json
CompletableFuture.java
[
{
"employeeId": "59-385-1088",
"firstName": "Zacharias",
"lastName": "Schwerin",
"email": "zschwerin0@altervista.org",
"gender": "Male",
"newJoiner": "TRUE",
"learningPending": "FALSE",
"salary": 101146,
"rating": 0
},
{


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

```
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 thenApplyAsycn and thenApplyAccept 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());
    }).thenAcceptAsync((emails) -> {
EmployeeReminderService › sendReminderToEmployee() › (emails) -> {...}
```

```
EmployeeReminderService.java | CompletableFuture.java | Consumer.java | Function.java | employees.json | EmployeeDatabase.java
30          }).thenAcceptAsync((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.

```
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
```

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-2
filter training not complete employee  : pool-1-thread-3
get emails  : pool-1-thread-4
send email  : pool-1-thread-5
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