public CustomerResponse getCustomerByIdUsingExceptionally(Integer customerId) {  
 log.info("Getting customer by id {} using exceptionally.", customerId);  
 CompletableFuture<CustomerResponse> customerResponseCF = CompletableFuture.supplyAsync(  
 () -> CustomerResponse.valueOf(customerClient.getCustomerById(customerId)));  
 CompletableFuture<Set<PurchaseTransactionResponse>> purchaseTransactionsCF = CompletableFuture.supplyAsync(  
 () -> purchaseTransactionClient  
 .getPurchaseTransactionsByCustomerId(customerId, isException(customerId))  
 .stream()  
 .map(PurchaseTransactionResponse::valueOf)  
 .collect(Collectors.toSet()))  
 .exceptionally(ex -> {  
 log.error("Received exception {}, returning empty list.", ex.getMessage());  
 return Set.of();  
 });  
 CompletableFuture<CustomerResponse> customerResponseCompletableFuture = customerResponseCF  
 .thenCombine(purchaseTransactionsCF, (customerResponse, purchaseTransactions) -> {  
 customerResponse.setPurchaseTransactions(purchaseTransactions);  
 return customerResponse;  
 });  
 CustomerResponse response = customerResponseCompletableFuture.join();  
 return response;  
}

When we create a CompletableFuture object we can call on it a method called **exceptionally**. In case of exception occurred during the execution of the ComplitebleFuture this method will be called.

The method receives an exception and returns the same object as CompleatableFuture returns. Using this method we can[catch an exception,](http://www.java67.com/2013/01/java-7-improved-exception-handling-multiple-catch-block.html) log it and return some default value.

As you can see from the code snipped above, the app returns an empty list of purchase transactions in case the call to purchase transaction client throws an exception. It is worth mentioning that this method is called only in case of an exception, when there is no exception this method won't be called.

Looking ahead it is the best way to handle the exceptions in the CompletableFuture in my opinion.