public CustomerResponse getCustomerByIdUsingHandle(Integer customerId) {  
 log.info("Getting customer by id {} using handle.", 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()))  
 .handle((response, ex) -> {  
 log.info("Executing exception handler for purchase transaction CF...");  
 if (ex != null) {  
 log.error("Received exception {}, returning empty list.", ex.getMessage());  
 return Collections.EMPTY\_SET;  
 }  
 return response;  
 });  
 CompletableFuture<CustomerResponse> customerResponseCompletableFuture = customerResponseCF  
 .thenCombine(purchaseTransactionsCF, (customerResponse, purchaseTransactions) -> {  
 customerResponse.setPurchaseTransactions(purchaseTransactions);  
 return customerResponse;  
 });  
 CustomerResponse response = customerResponseCompletableFuture.join();  
 return response;  
}

Similar to the previous example is the use of the handle method. Unlike exceptionally which takes a [Function](https://javarevisited.blogspot.com/2018/01/what-is-functional-interface-in-java-8.html), the handle method accepts a [BiFunction](https://www.java67.com/2022/12/top-5-functional-interface-every-java.html" \t "_blank). In other words here we get one more object beside the exception and it is a response itself.

public CustomerResponse getCustomerByIdUsingWhenComplete(Integer customerId) {  
 log.info("Getting customer by id {} using when complete.", 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()))  
 .whenComplete((response, ex) -> {  
 log.info("Executing whenComplete for purchase transaction CF...");  
 if (ex != null) {  
 log.error("Received exception {}, throwing exception to consumer", ex.getMessage());  
 }  
 });  
 CompletableFuture<CustomerResponse> customerResponseCompletableFuture = customerResponseCF  
 .thenCombine(purchaseTransactionsCF, (customerResponse, purchaseTransactions) -> {  
 customerResponse.setPurchaseTransactions(purchaseTransactions);  
 return customerResponse;  
 });  
 CustomerResponse response = customerResponseCompletableFuture.join();  
 return response;  
}

public CustomerResponse getCustomerByIdUsingExceptionallyRethrow(Integer customerId) {  
 log.info("Getting customer by id {} using exceptionally and rethrow.", 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 {}, throwing new exception!", ex.getMessage());  
 throw new IllegalArgumentException();  
 });  
 CompletableFuture<CustomerResponse> customerResponseCompletableFuture = customerResponseCF  
 .thenCombine(purchaseTransactionsCF, (customerResponse, purchaseTransactions) -> {  
 customerResponse.setPurchaseTransactions(purchaseTransactions);  
 return customerResponse;  
 });  
 CustomerResponse response = customerResponseCompletableFuture.join();  
 return response;  
}

public CustomerResponse getCustomerByIdWithOrTimeout(Integer customerId) {  
 log.info("Getting customer by id {} with orTimeout", customerId);  
 int timeOut = getTimeOut(customerId);  
 log.info("CF timeout is {}", timeOut);  
 CompletableFuture<CustomerResponse> customerResponseCF = CompletableFuture.supplyAsync(  
 () -> CustomerResponse.valueOf(customerClient.getCustomerById(customerId)));  
 CompletableFuture<Set<PurchaseTransactionResponse>> purchaseTransactionsCF = CompletableFuture.supplyAsync(  
 () -> purchaseTransactionClient  
 .getPurchaseTransactionsByCustomerId(customerId, false)  
 .stream()  
 .map(PurchaseTransactionResponse::valueOf)  
 .collect(Collectors.toSet()))  
 .orTimeout(timeOut, TimeUnit.SECONDS);  
 CompletableFuture<CustomerResponse> customerResponseCompletableFuture = customerResponseCF  
 .thenCombine(purchaseTransactionsCF, (customerResponse, purchaseTransactions) -> {  
 customerResponse.setPurchaseTransactions(purchaseTransactions);  
 return customerResponse;  
 });  
 CustomerResponse response = customerResponseCompletableFuture.join();  
 return response;  
}

public CustomerResponse getCustomerByIdWithCompleteOnTimeout(Integer customerId) {  
 log.info("Getting customer by id {} with completeOnTimeout.", customerId);  
 int timeOut = getTimeOut(customerId);  
 log.info("CF timeout is {}", timeOut);  
 CompletableFuture<CustomerResponse> customerResponseCF = CompletableFuture.supplyAsync(  
 () -> CustomerResponse.valueOf(customerClient.getCustomerById(customerId)));  
 CompletableFuture<Set<PurchaseTransactionResponse>> purchaseTransactionsCF = CompletableFuture.supplyAsync(  
 () -> purchaseTransactionClient  
 .getPurchaseTransactionsByCustomerId(customerId, false)  
 .stream()  
 .map(PurchaseTransactionResponse::valueOf)  
 .collect(Collectors.toSet()))  
 .completeOnTimeout(Set.of(), timeOut, TimeUnit.SECONDS);  
 CompletableFuture<CustomerResponse> customerResponseCompletableFuture = customerResponseCF  
 .thenCombine(purchaseTransactionsCF, (customerResponse, purchaseTransactions) -> {  
 customerResponse.setPurchaseTransactions(purchaseTransactions);  
 return customerResponse;  
 });  
 CustomerResponse response = customerResponseCompletableFuture.join();  
 return response;  
}