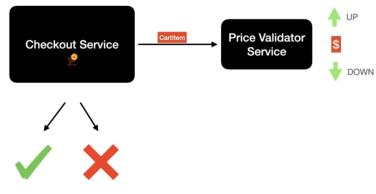
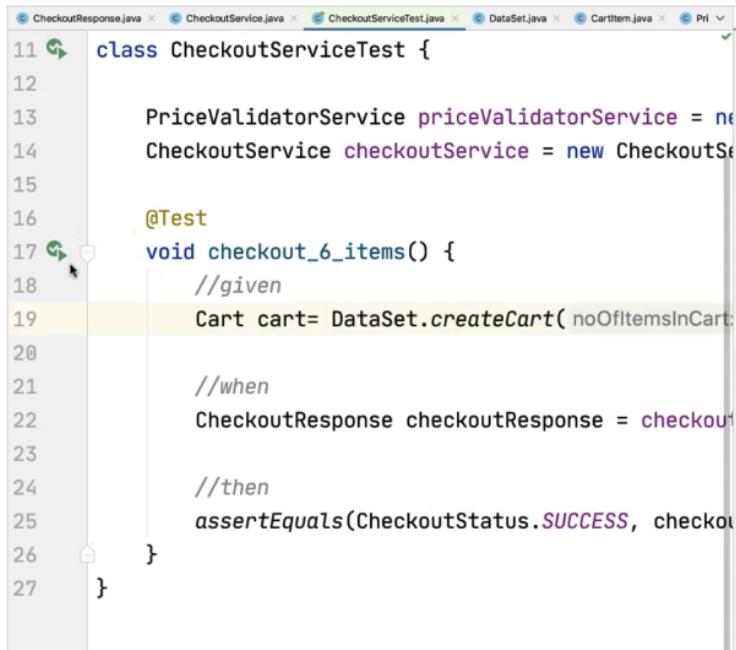


Overview of the Retail Checkout Service

Checkout Service(BackEnd)



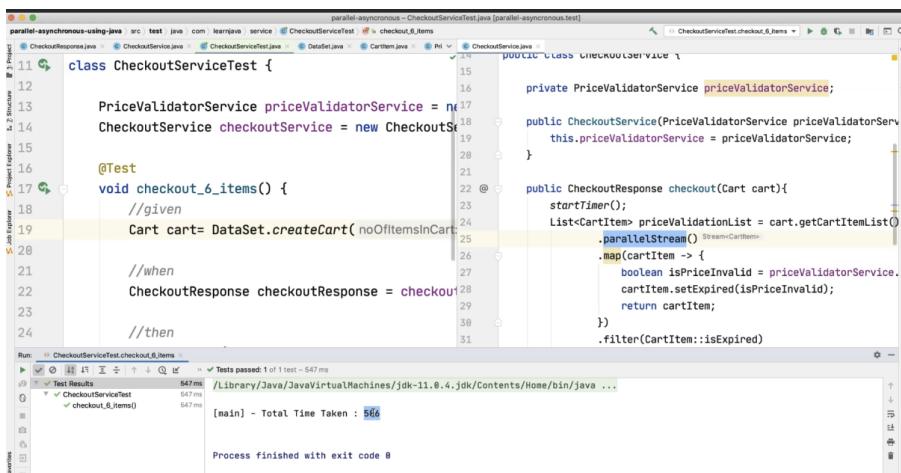
```
22 @
23     public CheckoutResponse checkout(Cart cart){
24         startTimer();
25         List<CartItem> priceValidationList = cart.getCartItemList() List<CartItem>
26             .stream() Stream<CartItem>
27             .map(cartItem -> {
28                 boolean isPriceInvalid = priceValidatorService.isCartItemInvalid(cartItem);
29                 cartItem.setExpired(isPriceInvalid);
30                 return cartItem;
31             })
32             .filter(CartItem::isExpired)
33             .collect(Collectors.toList());
34
35         if(priceValidationList.size()>0){
36             return new CheckoutResponse(CheckoutStatus.FAILURE, priceValidationList);
37         }
38
39         timeTaken();
40         return new CheckoutResponse(CheckoutStatus.SUCCESS);
41     }
42 }
```



```

11 class CheckoutServiceTest {
12
13     PriceValidatorService priceValidatorService = new PriceValidatorService();
14     CheckoutService checkoutService = new CheckoutService(priceValidatorService);
15
16     @Test
17     void checkout_6_items() {
18         //given
19         Cart cart= DataSet.createCart(noOfItemsInCart);
20
21         //when
22         CheckoutResponse checkoutResponse = checkoutService.checkout(cart);
23
24         //then
25         assertEquals(CheckoutStatus.SUCCESS, checkoutResponse.getStatus());
26     }
27 }

```



```

11 class CheckoutServiceTest {
12
13     PriceValidatorService priceValidatorService = new PriceValidatorService();
14     CheckoutService checkoutService = new CheckoutService(priceValidatorService);
15
16     @Test
17     void checkout_6_items() {
18         //given
19         Cart cart= DataSet.createCart(noOfItemsInCart);
20
21         //when
22         CheckoutResponse checkoutResponse = checkoutService.checkout(cart);
23
24         //then
25         assertEquals(CheckoutStatus.SUCCESS, checkoutResponse.getStatus());
26     }
27 }

```

Run: CheckoutServiceTest.checkout_6_items
 Tests passed: 1 of 1 test - 56 ms
 /Library/Java/JavaVirtualMachines/jdk-11.0.4.jdk/Contents/Home/bin/java ...
 [main] - Total Time Taken : 56
 Process finished with exit code 0

How to find the number of cores in a machine. 12 tasks can be performed at a time



```

16 @Test
17 void no_of_cores() {
18     //given
19
20     //when
21     System.out.println("no of cores :" + Runtime.getRuntime().availableProcessors());
22
23     //then
24 }
25
26 @Test

```

Run: CheckoutServiceTest.no_of_cores
 Tests passed: 1 of 1 test - 20 ms
 /Library/Java/JavaVirtualMachines/jdk-11.0.4.jdk/Contents/Home/bin/java ...
 [main] - Total Time Taken : 20 ms
 no of cores :12

If we process more than this it will take double time as shown below.

Screenshot of Eclipse IDE showing a Java test method for `CheckoutService` with 13 items. The code is as follows:

```
37
38
39     @Test
40     void checkout_13_items() {
41         //given
42         Cart cart= DataSet.createCart( noOfItemsInCart: 13 );
43
44         //when
45         CheckoutResponse checkoutResponse = checkoutService.checkout(cart);
46
47         //then
48         assertEquals(CheckoutStatus.FAILURE, checkoutResponse.getCheckoutStatus());
49     }

```

The Run view shows the test results for `CheckoutServiceTest.checkout_13_items()` taking 1s 49ms.

Screenshot of Eclipse IDE showing the same Java test method for `CheckoutService` but with 25 items. The code is as follows:

```
37
38
39     @Test
40     void checkout_13_items() {
41         //given
42         Cart cart= DataSet.createCart( noOfItemsInCart: 25 );
43
44         //when
45         CheckoutResponse checkoutResponse = checkoutService.checkout(cart);
46
47         //then
48         assertEquals(CheckoutStatus.FAILURE, checkoutResponse.getCheckoutStatus());
49     }

```

The Run view shows the test results for `CheckoutServiceTest.checkout_13_items()` taking 1s 561ms.

ParallelStreams - How it works ?

- `parallelStream()`
 - **Split** the data in to chunks
 - **Execute** the data chunks
 - **Combine** the result

parallelStream() - How it works ?

• Split

- Data Source is split in to small data chunks
 - Example - **List Collection** split into chunks of elements to **size 1**
- This is done using **Spliterators**
 - For ArrayList, the **Spliterator** is **ArrayListSpliterator**

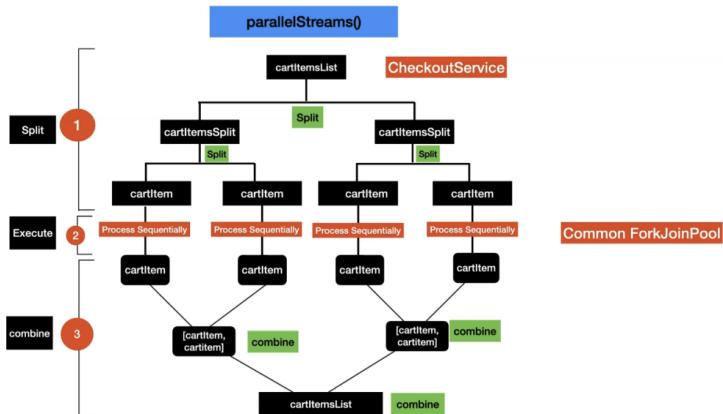
• Execute

- Data chunks are applied to the Stream Pipeline and the **Intermediate** operations executed in a **Common ForkJoin Pool**
 - Watch the **Fork/Join FrameWork Lectures**

• Combine

- Combine the executed results into a final result
 - Combine phase in Streams API maps to **terminal** operations
 - Uses **collect()** and **reduce()** functions
 - **collect(toList())**

parallelStream() - How it works ?



Spliterator in ParallelStreams

- Data source is split in to multiple chunks by the Spliterator
- Each and every collection has a different Spliterator Implementation
- Performance differ based on the implementation

Multiply each value in the collection by a user passed value

[1, 2, 3, 4] -> [2, 4, 6, 8]

Value * 2

For one lack record it took 54 mins.

```

parallel-asynchronous [D:\Dip\Udemy\parallel-asynchronous-programming-in-java\course-codebase\parallel-asynchronous-using-java] - ArrayListSpliteratorExampleTest.java [parallel-asynchronous.test]
parallel-asynchronous-using-java src test java com learnjava parallelstreams ArrayListSpliteratorExample ArrayListSpliteratorExampleTest DataSet.java
ArrayListSpliteratorExample.java ArrayListSpliteratorExampleTest.java DataSet.java
ArrayListSpliteratorExample.java

15
16     @Test
17     void arrayListSpliteratorExample() {
18         //given
19         int size = 1000000;
20         ArrayList<Integer> inputList = DataSet.generateArrayLists();
21
22         //when
23         List<Integer> resultList = arrayListSpliteratorExample.m
24
25         //then
26         assertEquals(size, resultList.size());
27     }
28
29 }
30

public class ArrayListSpliteratorExample {
31
32     public List<Integer> multiplyEachValue(ArrayList<Integer> inputL
33
34         startTimer();
35         Stream<Integer> integerStream = inputList.stream(); //sequential
36
37         List<Integer> resultList = integerStream
38             .map(integer -> integer*multiplyValue)
39             .collect(Collectors.toList());
40
41         timeTaken();
42     }
43
44     return resultList;
45 }

Run: ArrayListSpliteratorExampleTest.arrayListS...
Test Results 132 ms
ArrayListSpliteratorExampleTest 132 ms
arrayListSpliteratorExample 132 ms
[main] - Total Time Taken : 56
Process finished with exit code 0

```

Repeat the test using `@RepeatedAnnotation` as shown below due to JVM and cache it will reduce the performance time as shown below.

```

parallel-asynchronous [D:\Dip\Udemy\parallel-asynchronous-programming-in-java\course-codebase\parallel-asynchronous-using-java] - ArrayListSpliteratorExample.java [parallel-asynchronous.main]
parallel-asynchronous-using-java src main java com learnjava parallelstreams ArrayListSpliteratorExample multiplyEachValue
ArrayListSpliteratorExample.java ArrayListSpliteratorExampleTest DataSet.java
ArrayListSpliteratorExample.java

7
8     import java.util.ArrayList;
9     import java.util.List;
10
11    import static org.junit.jupiter.api.Assertions.*;
12
13    class ArrayListSpliteratorExampleTest {
14
15        ArrayListSpliteratorExample arrayListSpliteratorExample = new
16
17        @RepeatedTest(5)
18        void multiplyEachValue() {
19            //given
20            int size = 1000000;
21            ArrayList<Integer> inputList = DataSet.generateArrayLists();
22
23            //when
24            List<Integer> resultlist = arrayListSpliteratorExample.m
25
26            //then
27            assertEquals(size, resultlist.size());
28        }
29    }
30

public class ArrayListSpliteratorExample {
31
32     public List<Integer> multiplyEachValue(ArrayList<Integer> inputL
33
34         startTimer();
35         Stream<Integer> integerStream = inputList.stream(); //sequential
36
37         List<Integer> resultList = integerStream
38             .map(integer -> integer*multiplyValue)
39             .collect(Collectors.toList());
40
41         timeTaken();
42     }
43
44     return resultList;
45 }


```

```

Run: ArrayListSpliteratorExampleTest.multiplyEachValue...
Test Results 488 ms
ArrayListSpliteratorExampleTest 488 ms
multiplyEachValue() 488 ms
repetition 1 of 5 136 ms
repetition 2 of 5 87 ms
repetition 3 of 5 102 ms
repetition 4 of 5 74 ms
repetition 5 of 5 89 ms
[main] - Total Time Taken : 58
[main] - Total Time Taken : 41
[main] - Total Time Taken : 28
[main] - Total Time Taken : 19
[main] - Total Time Taken : 23

```

Convert sequential to parallel. By default streams are sequential

```

1 package com.learnjava.parallelstreams;
2
3 import com.learnjava.util.DataSet;
4 import org.junit.jupiter.api.RepeatedTest;
5 import org.junit.jupiter.api.Test;
6
7 import java.util.ArrayList;
8 import java.util.List;
9
10 import static org.junit.jupiter.api.Assertions.*;
11
12 class ArrayListSpliteratorExampleTest {
13
14     ArrayListSpliteratorExample arrayListSpliteratorExample = new
15
16     @RepeatedTest(5)
17     void multiplyEachValue() {
18         //given
19         int size = 1000000;
20         ArrayList<Integer> inputList = DataSet.generateArrayList(s
21
22         //when
23         List<Integer> resultlist = arrayListSpliteratorExample.mu
24
25         //then
26         assertEquals(size, resultList.size());
27     }
28
29 }

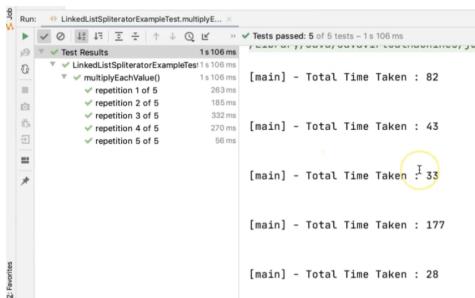
```

```

    'atorExample = new ArrayListSpliteratorExample();
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31

```

The parallel work on arraylist you may not find much difference between sequential and parallel. Because underlying ArrayList structure try same with linkedlist.



LinkedList is taking more time compared to arraylist. So follow

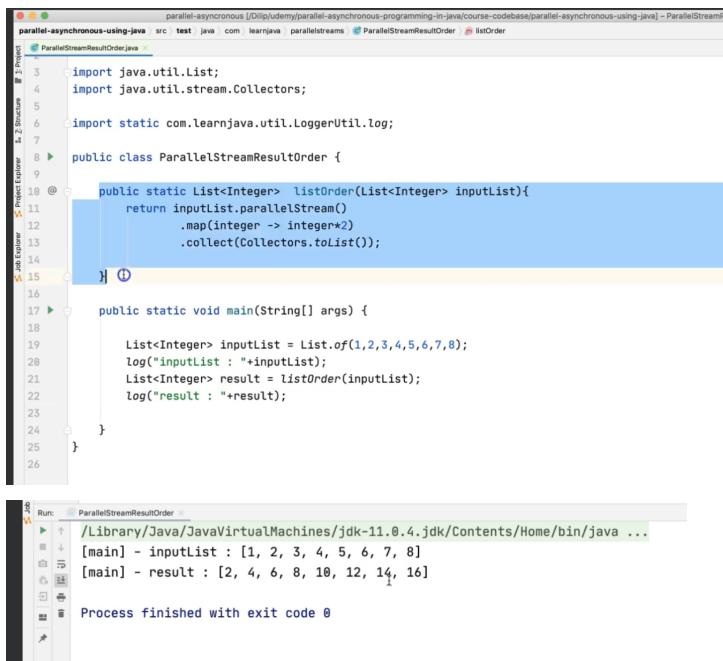
Summary - Spliterator in ParallelStreams

- Invoking **parallelStream()** does not guarantee faster performance of your code
- Need to perform additional steps compared to sequential
- Splitting , Executing and Combining

Recommendation - Always compare the performance before you use parallelStream()

Parallel Streams - Final Computation Result Order

- The order of the collection depends on:
 - Type of Collection
 - Splitter Implementation of the collection
- Example : ArrayList
 - Type of Collection - **Ordered** 
 - Splitter Implementation - Ordered Splitter Implementation
- Example : Set
 - Type of Collection - **UnOrdered** 
 - Splitter Implementation - UnOrdered Splitter Implementation



```
import java.util.List;
import java.util.stream.Collectors;

import static com.learnjava.util.LoggerUtil.log;

public class ParallelStreamResultOrder {

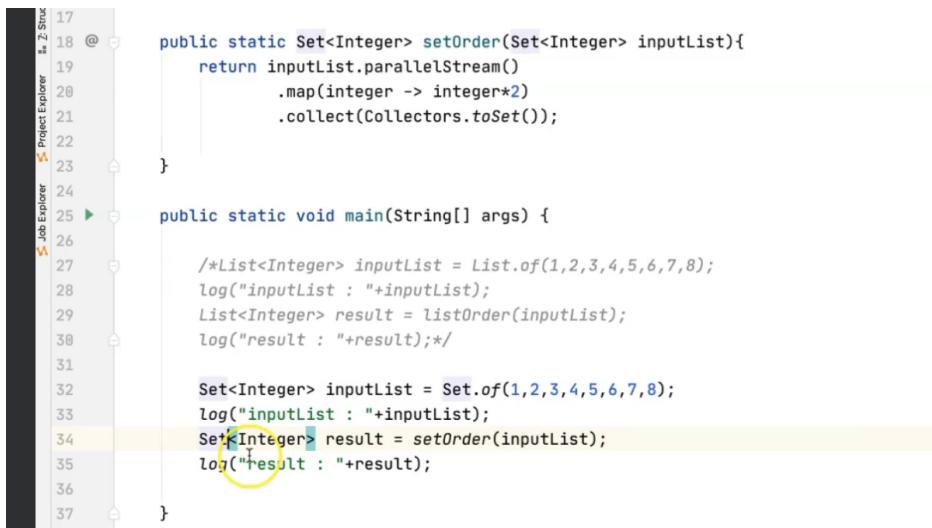
    public static List<Integer> listOrder(List<Integer> inputList){
        return inputList.parallelStream()
            .map(integer -> integer*2)
            .collect(Collectors.toList());
    }

    public static void main(String[] args) {
        List<Integer> inputList = List.of(1,2,3,4,5,6,7,8);
        log("inputList : "+inputList);
        List<Integer> result = listOrder(inputList);
        log("result : "+result);
    }
}
```

Run ParallelStreamResultOrder

```
/Library/Java/JavaVirtualMachines/jdk-11.0.4.jdk/Contents/Home/bin/java ...
[main] - inputList : [1, 2, 3, 4, 5, 6, 7, 8]
[main] - result : [2, 4, 6, 8, 10, 12, 14, 16]

Process finished with exit code 0
```



```
public static Set<Integer> setOrder(Set<Integer> inputList){
    return inputList.parallelStream()
        .map(integer -> integer*2)
        .collect(Collectors.toSet());
}

public static void main(String[] args) {
    /*List<Integer> inputList = List.of(1,2,3,4,5,6,7,8);
    log("inputList : "+inputList);
    List<Integer> result = listOrder(inputList);
    log("result : "+result);*/

    Set<Integer> inputList = Set.of(1,2,3,4,5,6,7,8);
    log("inputList : "+inputList);
    Set<Integer> result = setOrder(inputList);
    log("result : "+result);
}
```

```

Run: ParallelStreamResultOrder
/Library/Java/JavaVirtualMachines/jdk-11.0.4.jdk/Contents/Home/bin/java ...
[main] - inputList : [1, 2, 3, 4, 5, 6, 7, 8]
[main] - result : [16, 2, 4, 6, 8, 10, 12, 14]
Process finished with exit code 0

```

Collect() vs Reduce()

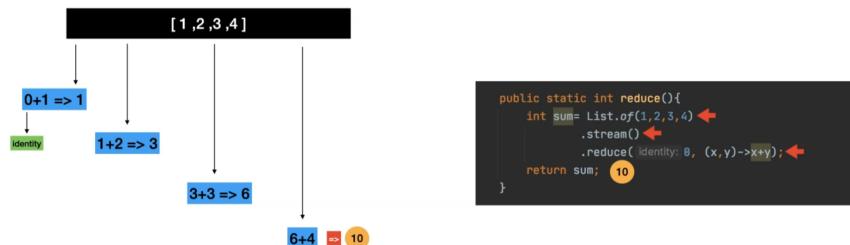
Collect

- Part of Streams API
- Used as a terminal operation in **Streams API**
- Produces a single result
- Result is produced in a mutable fashion
- Feature rich and used for many different use cases
- Example
 - `collect(toList()), collect(toSet())`
 - `collect(summingDouble(Double::doubleValue));`

Reduce

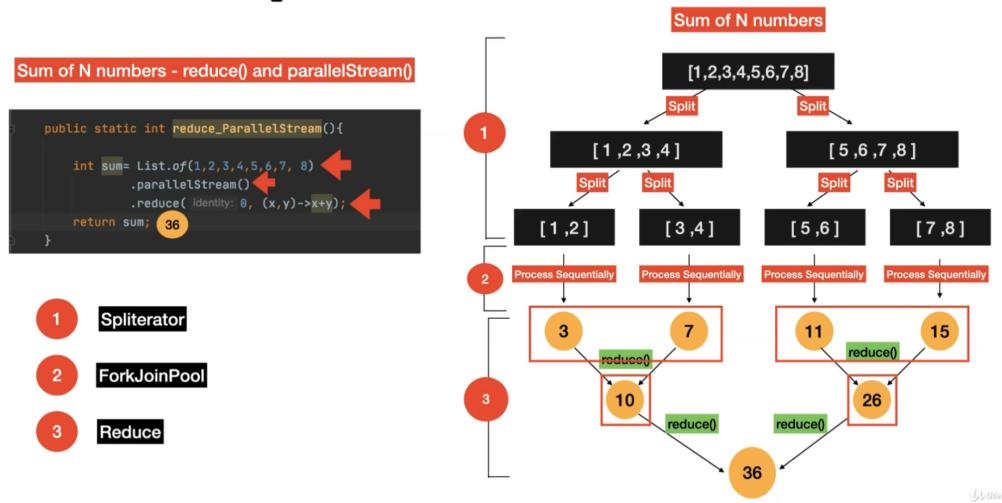
- Part of Streams API
- Used as a terminal operation in **Streams API**
- Produces a single result
- Result is produced in a immutable fashion
- Reduce the computation into a single value
 - Sum, Multiplication
- Example
 - Sum -> `reduce(0.0, (x, y)->x+y)`
 - Multiply -> `reduce(1.0, (x, y)->x * y)`

How reduce() works ?



The `reduce()` function performs an immutable computation throughout in each and every step.

How reduce() with ParallelStream works ?



Calculate final price of checkout out service using sum and reduce methods use either one.

```

private double calculateFinalPrice(Cart cart) {
    //double finalPrice = calculateFinalPrice(cart);
    double finalPrice = calculateFinalPrice_reduce(cart);
    Log(" Checkout Complete and the final price is " + finalPrice);
    return new CheckoutResponse(CheckoutStatus.SUCCESS,finalPrice);
}

private double calculateFinalPrice(Cart cart) {
    return cart.getCartItemList() List<CartItem>
        .parallelStream() Stream<CartItem>
        .map(cartItem -> cartItem.getQuantity() * cartItem.getRate()) Stream<Double>
        .mapToDouble(Double::doubleValue) DoubleStream
        .sum();
}

private double calculateFinalPrice_reduce(Cart cart) {
    return cart.getCartItemList() List<CartItem>
        .parallelStream() Stream<CartItem>
        .map(cartItem -> cartItem.getQuantity() * cartItem.getRate()) Stream<Double>
        .reduce(identity: 0.0, (x,y)->x+y);
}

```

Between collection and reduce collect perform good on string concatenation.

```

parallel-asynchronous-using-java / src / main / java / com / learnjava / parallelstreams / CollectVsReduce.java
11
12     public static String collect() {
13         List<String> list = DataSet.namesList();
14
15         String result = list.
16             parallelStream().
17                 collect(Collectors.joining());
18
19         return result;
20     }
21
22     public static String reduce() {
23
24         List<String> list = DataSet.namesList();
25
26         String result = list.
27             parallelStream().
28                 reduce( identity: "", (s1, s2) -> s1 + s2);
29
30         return result;
31     }

```

Identity in reduce()

- Identity gives you the same value when its used in the computation

- Addition: **Identity = 0**

- $0 + 1 \Rightarrow 1$
- $0 + 20 \Rightarrow 20$

- Multiplication : **Identity = 1**

- $1 * 1 \Rightarrow 1$
- $1 * 20 \Rightarrow 20$

Sum of N numbers - reduce() and parallelStream()

```

public static int reduce_ParallelStream(){
    int sum= List.of(1,2,3,4,5,6,7, 8)
        .parallelStream()
        .reduce( identity: 0, (x,y)->x+y);
    return sum;
}

```

reduce() is recommended for computations that are associative

Identity gives the same value in multiplication we should use 1, in addition 0 and in string empty string.

The screenshot shows an IDE interface with two main panes. The left pane displays a Java test class named `ReduceExampleTest` containing several test methods for parallel streams. The right pane shows the corresponding implementation classes: `ReduceExample.java` and `ReduceExampleTest.java`. The code uses `parallelStream()` and `reduce()` to process lists of integers.

```
parallel-asynchronous-using-java
parallel-asynchronous [D:\Drop\udemyn\parallel-asynchronous-programming-in-Java\course-codebase\parallel-asynchronous-using-Java] - ReduceExampleTest.java [parallel-asynchronous.test]
src | test | java | com | learnjava | parallelstreams | ReduceExampleTest.java | ReduceExampleTest.reduced_parallelStream_emptyList | Event Log

parallel-asynchronous-using-java
parallel-asynchronous [D:\Drop\udemyn\parallel-asynchronous-programming-in-Java\course-codebase\parallel-asynchronous-using-Java] - ReduceExampleTest.java [parallel-asynchronous.test]
src | test | java | com | learnjava | parallelstreams | ReduceExample.java | ReduceExampleTest.java | ReduceExampleTest.reduced_parallelStream_emptyList | Event Log

ReduceExampleTest.java
1 @TEST
2 void reduce_parallelStream_emptyList() {
3
4     //given
5     List<Integer> inputList = new ArrayList<>();
6
7     //when
8     int result = reduceExample.reduce_sum_parallelStream(inputList);
9
10    //then
11    assertEquals(expected: 0, result);
12 }
13
14 @Test
15 void reduce_parallelStream_multiply() {
16     //given
17     List<Integer> inputList = List.of(1,2,3,4);
18
19     //when
20     int result = reduceExample.reduce_multiply_parallelStream(inputList);
21
22    //then
23    assertEquals(expected: 24, result);
24 }
25
26 @Test
27 void reduce_multiply_emptyList() {
28
29     //given
30     List<Integer> inputList = new ArrayList<>();
31
32     //when
33     int result = reduceExample.reduce_multiply_parallelStream(inputList);
34
35    //then
36    assertEquals(expected: 1, result);
37 }
38
39 @Test
40 void reduce_parallelStream_reduce() {
41     //given
42     List<Integer> inputList = new ArrayList<>();
43
44     //when
45     int result = reduceExample.reduce_parallelStream_reduce(inputList);
46
47    //then
48    assertEquals(expected: 0, result);
49 }
50
51 @Test
52 void reduce_parallelStream_reduce_identity() {
53
54     //given
55     List<Integer> inputList = new ArrayList<>();
56
57     //when
58     int result = reduceExample.reduce_parallelStream_reduce_identity(inputList);
59
60    //then
61    assertEquals(expected: 0, result);
62 }

ReduceExample.java
1 package com.learnjava.parallelstreams;
2
3 import java.util.List;
4
5 public class ReduceExample {
6
7     @Public int reduce_sum_parallelStream(List<Integer> inputList) {
8
9         int sum= inputList
10            .parallelStream()
11            //.reduce(1, (x,y)->x+y);
12            .reduce(identity: 0, (x,y)->x+y);
13
14        return sum;
15    }
16
17    @Public int reduce_multiply_parallelStream(List<Integer> inputList) {
18
19        int multiply= inputList
20            .parallelStream()
21            .reduce(identity: 1, (x,y)->x*y);
22
23        return multiply;
24    }
25 }

ReduceExampleTest.java
1 package com.learnjava.parallelstreams;
2
3 import java.util.List;
4
5 public class ReduceExampleTest {
6
7     @Test void reduced_parallelStream_emptyList() {
8
9         //given
10        List<Integer> inputList = new ArrayList<>();
11
12        //when
13        int result = reduceExample.reduce_parallelStream_emptyList();
14
15        //then
16        assertEquals(expected: 0, result);
17    }
18
19    @Test void reduced_parallelStream_reduce() {
20
21        //given
22        List<Integer> inputList = new ArrayList<>();
23
24        //when
25        int result = reduceExample.reduce_parallelStream_reduce();
26
27        //then
28        assertEquals(expected: 0, result);
29    }
30
31    @Test void reduced_parallelStream_reduce_identity() {
32
33        //given
34        List<Integer> inputList = new ArrayList<>();
35
36        //when
37        int result = reduceExample.reduce_parallelStream_reduce_identity();
38
39        //then
40        assertEquals(expected: 0, result);
41    }
42 }
```

Try to change the identity value (right side) and run the test case left. We may see the weared result because when you run in parallel - split execute and combine will happen - for each split wrong identity value will be taken and calculation starts.

Parallel Stream Operations & Poor Performance

- Stream Operations that perform poor
 - Impact of **Boxing** and **UnBoxing** when it comes to parallel Streams
 - **Boxing** -> Converting a Primitive Type to Wrapper class equivalent
 - **1 -> new Integer(1)** 
 - **UnBoxing** -> Converting a Wrapper class to Primitive equivalent
 - **new Integer(1) -> 1** 

And its example and test cases provided below.

The screenshot shows an IDE interface with two panes. The left pane displays the Java code for `ParallelStreamPerformanceTest`. The right pane shows the execution results in the 'Run' tab, specifically the 'Test Result' section.

```

parallel-asynchronous - ParallelStreamPerformanceTest.java [parallel-asynchronous.test]
parallel-asynchronous-using-java src test java com learnjava parallelstreams ParallelStreamPerformanceTest m_using_intstream_parallel

import ...
import static com.learnjava.util.CommonUtil.startTimer;
import static com.learnjava.util.CommonUtil.timeTaken;

public class ParallelStreamPerformance {

    public int sum_using_intstream(int count, boolean isParallel){
        startTimer();
        IntStream intStream = IntStream.rangeClosed(0, count);

        if(isParallel)
            intStream.parallel();

        int sum = intStream
            .sum();
        timeTaken();
        return sum;
    }

    public int sum_using_list(List<Integer> inputList, boolean isParallel){
        startTimer();
        Stream<Integer> inputStream = inputList.stream();

        if(isParallel)
            inputStream.parallel();

        int sum = inputStream
            .mapToInt(Integer::intValue) // unboxing
            .sum();
        timeTaken();
        return sum;
    }

    public int sum_using_iterate(int n, boolean isParallel){
        startTimer();
        Stream<Integer> integerStream = Stream.
            iterate(0, i ->i+1 );
    }
}

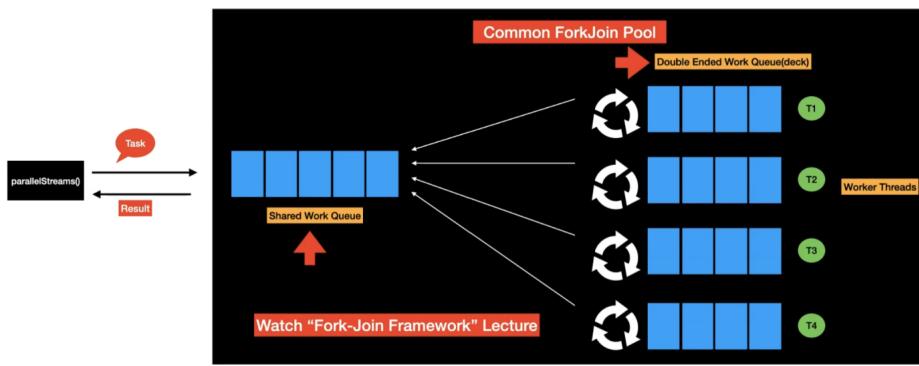
```

Test Results:

- ParallelStreamPerformanceTest
 - sum_using_intstream() 294 ms
 - [main] - Total Time Taken : 31
 - sum : 1784293664
 - sum_using_intstream_parallel() 22 ms
 - [main] - Total Time Taken : 22
 - sum : 1784293664
 - sum_using_iterate() 71 ms
 - [main] - Total Time Taken : 71
 - sum : 1784293664
 - sum_using_iterate_parallel() 43 ms
 - [main] - Total Time Taken : 43
 - sum : 1784293664
 - sum_using_list() 66 ms
 - [main] - Total Time Taken : 66
 - sum : 1784293664
 - sum_using_list_parallel() 20 ms
 - [main] - Total Time Taken : 20
 - sum : 1784293664

Tests passed: 6 (38 minutes ago)

Common ForkJoin Pool



Common ForkJoin Pool

- Common ForkJoin Pool is used by:
 - ParallelStreams
 - CompletableFuture
 - CompletableFuture have options to use a User-defined ThreadPools
- Common ForkJoin Pool is shared by the whole process

Parallelism & Threads in Common ForkJoinPool

- parallelStreams()
 - Runs your code in parallel
 - Improves the performance of the code
- Is there a way to look in to parallelism and threads involved ?
 - Yes

```
27
28
29     @Test
30     void parallelism() {
31         //given
32         //when
33         System.out.println("parallelism :" + ForkJoinPool.getCommonPoolParallelism());
34
35         //then
36     }
37
38     @Test
```

Run: CheckoutServiceTest.parallelism

Test Results

CheckoutServiceTest

parallelism()

parallelism :11

Process finished with exit code 0

Actual cores are 12. => Main thread - 1 = 11.

```
8
9     public class PriceValidatorService {
10
11     @
12         public boolean isCartItemInvalid(CartItem cartItem){
13             int cartId = cartItem.getItemId();
14             log("isCartItemInvalid : " + cartItem);
15             delay( delayMilliseconds: 500);
```

```

CheckoutServiceTest.checkout13_items
  ✓ Tests passed: 1 of 1 test - 1s 107 ms
    Test Result: 1s 107 ms
      ✓ Check: 1s 107 ms
        ✓ che: 1s 107 ms
          [ForkJoinPool.commonPool-worker-19] - isCartItemInvalid : CartItem(itemId=4, itemName=CartItem -4, rate=78.46119062496592, quant
          [ForkJoinPool.commonPool-worker-7] - isCartItemInvalid : CartItem(itemId=5, itemName=CartItem -5, rate=67.88887802258029, quant
          [ForkJoinPool.commonPool-worker-23] - isCartItemInvalid : CartItem(itemId=2, itemName=CartItem -2, rate=78.86347273421539, quant
          [ForkJoinPool.commonPool-worker-3] - isCartItemInvalid : CartItem(itemId=6, itemName=CartItem -6, rate=57.536626454739434, quant
          [ForkJoinPool.commonPool-worker-31] - isCartItemInvalid : CartItem(itemId=11, itemName=CartItem -11, rate=88.56165656871106, quant
          [ForkJoinPool.commonPool-worker-21] - isCartItemInvalid : CartItem(itemId=9, itemName=CartItem -9, rate=77.68628288925395, quant
          [main] - isCartItemInvalid : CartItem(itemId=8, itemName=CartItem -8, rate=58.19176577651548, quantity=8, isExpired=false)
          [ForkJoinPool.commonPool-worker-5] - isCartItemInvalid : CartItem(itemId=12, itemName=CartItem -12, rate=59.86573944947712, quant
          [ForkJoinPool.commonPool-worker-17] - isCartItemInvalid : CartItem(itemId=10, itemName=CartItem -10, rate=87.6261469485215, quant
          [ForkJoinPool.commonPool-worker-27] - isCartItemInvalid : CartItem(itemId=1, itemName=CartItem -1, rate=85.85685384225236, quant
          [ForkJoinPool.commonPool-worker-13] - isCartItemInvalid : CartItem(itemId=3, itemName=CartItem -3, rate=108.95361149431073, quant
          [ForkJoinPool.commonPool-worker-9] - isCartItemInvalid : CartItem(itemId=7, itemName=CartItem -7, rate=74.61233977178599, quant
          [ForkJoinPool.commonPool-worker-19] - isCartItemInvalid : CartItem(itemId=13, itemName=CartItem -13, rate=88.32985285310899, quant
        [main] - Total Time Taken : 107 ms
  
```

Modifying Default parallelism in Parallel Streams

Modifying Default parallelism

→ `System.setProperty("java.util.concurrent.ForkJoinPool.common.parallelism", "100");`

OR

→ `-Djava.util.concurrent.ForkJoinPool.common.parallelism=100`

```

CheckoutServiceTest.modify_parallelism
  ✓ Tests passed: 1 of 1 test - 640 ms
    Test Result: 640 ms
      ✓ Check: 640 ms
        ✓ modify: 640 ms
          [ForkJoinPool.commonPool-worker-75] - isCartItemInvalid : CartItem(itemId=32, itemName=CartItem -32, rate=75.93835088585404, quant
          [ForkJoinPool.commonPool-worker-123] - isCartItemInvalid : CartItem(itemId=81, itemName=CartItem -81, rate=58.22475674832434, quant
          [ForkJoinPool.commonPool-worker-43] - isCartItemInvalid : CartItem(itemId=39, itemName=CartItem -39, rate=52.78559728178621, quant
          [ForkJoinPool.commonPool-worker-63] - isCartItemInvalid : CartItem(itemId=27, itemName=CartItem -27, rate=99.71229376136739, quant
          [ForkJoinPool.commonPool-worker-93] - isCartItemInvalid : CartItem(itemId=19, itemName=CartItem -19, rate=99.89656987742071, quant
          [ForkJoinPool.commonPool-worker-161] - isCartItemInvalid : CartItem(itemId=96, itemName=CartItem -96, rate=63.569635683915984, quant
          [ForkJoinPool.commonPool-worker-191] - isCartItemInvalid : CartItem(itemId=1, itemName=CartItem -1, rate=51.87679732871836, quant
          [ForkJoinPool.commonPool-worker-183] - isCartItemInvalid : CartItem(itemId=6, itemName=CartItem -6, rate=53.86651978275834, quant
          [ForkJoinPool.commonPool-worker-91] - isCartItemInvalid : CartItem(itemId=85, itemName=CartItem -85, rate=50.56381468103207, quant
          [ForkJoinPool.commonPool-worker-21] - isCartItemInvalid : CartItem(itemId=22, itemName=CartItem -22, rate=64.43348142663231, quant
          [ForkJoinPool.commonPool-worker-99] - isCartItemInvalid : CartItem(itemId=95, itemName=CartItem -95, rate=85.95951025837977, quant
          [ForkJoinPool.commonPool-worker-201] - isCartItemInvalid : CartItem(itemId=54, itemName=CartItem -54, rate=88.78357161742179, quant
          [ForkJoinPool.commonPool-worker-151] - isCartItemInvalid : CartItem(itemId=48, itemName=CartItem -48, rate=56.75388184454375, quant
          [ForkJoinPool.commonPool-worker-77] - isCartItemInvalid : CartItem(itemId=3, itemName=CartItem -3, rate=52.347718955889485, quant
          [ForkJoinPool.commonPool-worker-95] - isCartItemInvalid : CartItem(itemId=30, itemName=CartItem -30, rate=84.05613128736896, quant
        [main] - Total Time Taken : 640 ms
  
```

Process finished with exit code 0

It runs much faster.

Parallel Streams - When to use them ?

- Parallel Streams do a lot compared to sequential(default) Streams
- Parallel Streams
 - Split ←
 - Execute
 - Combine ←

Parallel Streams - When to use them ?

- Computation takes a longer time to complete
- Lots of data
- More cores in your machine

Always compare the performance between sequential and parallel streams

Parallel Streams - When not to use them ?

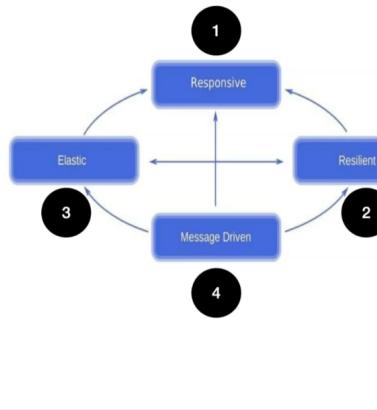
- Parallel Streams
 - Split ←
 - Execute
 - Combine ←
- Data set is small
- Auto Boxing and Unboxing doesn't perform better
- Stream API operators -> `iterate()`, `limit()`

CompletableFuture

- Introduced in **Java 8**
- CompletableFuture is an **Asynchronous Reactive Functional Programming API**
- Asynchronous Computations in a functional Style
- CompletableFutures API is created to solve the limitations of Future API

CompletableFuture and Reactive Programming

- **Responsive:**
 - Fundamentally Asynchronous
 - Call returns immediately and the response will be sent when its available
- **Resilient:**
 - Exception or error won't crash the app or code
- **Elastic:**
 - Asynchronous Computations normally run in a pool of threads
 - No of threads can go up or down based on the need
- **Message Driven:**
 - Asynchronous computations interact with each other through messages in an event-driven style



We can categorize all the methods of CompletableFuture into 3 categories as shown below

CompletableFuture API

- **Factory Methods**
 - Initiate asynchronous computation
- **Completion Stage Methods**
 - Chain asynchronous computation
- **Exception Methods**
 - Handle Exceptions in an Asynchronous Computation

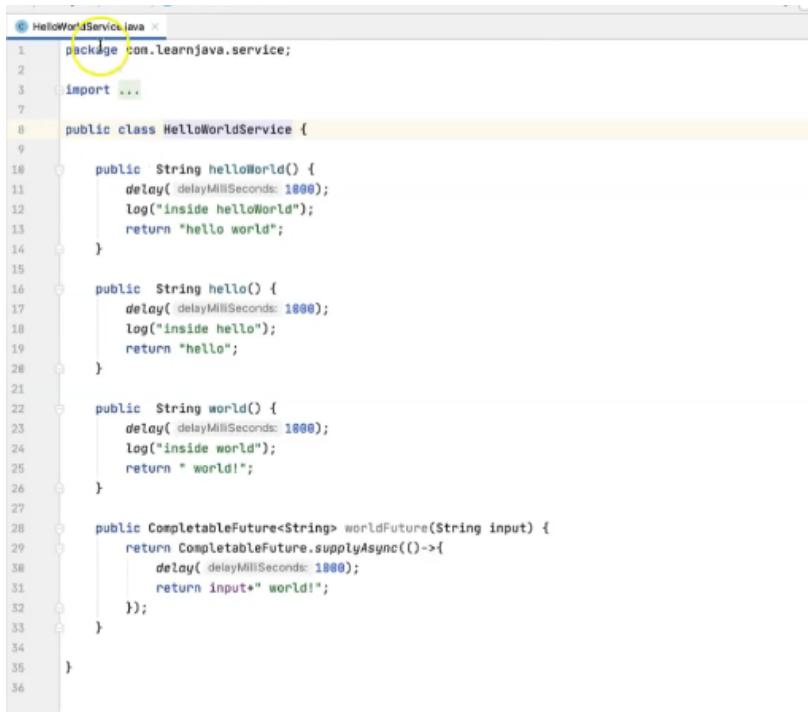
CompletableFuture

supplyAsync()

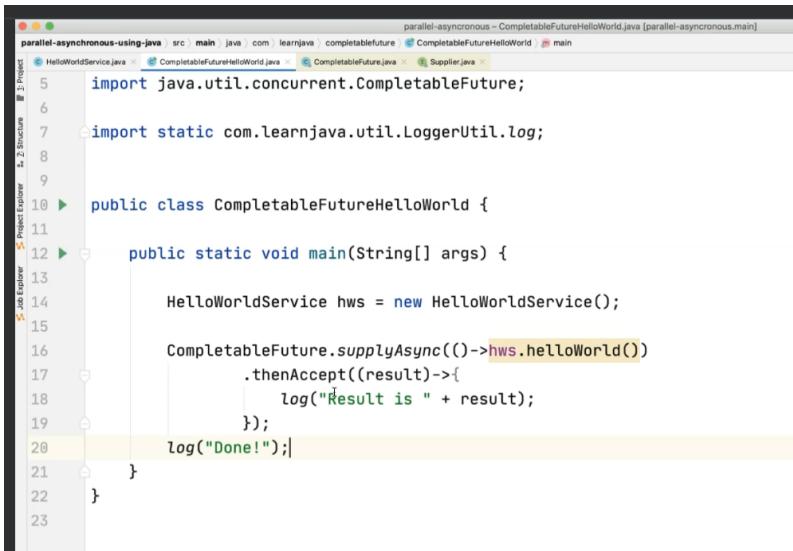
- FactoryMethod
- Initiate Asynchronous computation
- Input is **Supplier** Functional Interface
- Returns CompletableFuture<T>()

thenAccept()

- CompletionStage Method
- Chain Asynchronous Computation
- Input is **Consumer** Functional Interface
- Consumes the result of the previous
- Returns CompletableFuture<Void>



```
❶ HelloWorldService.java x
❷ package com.learnjava.service;
❸ import ...
❹ public class HelloWorldService {
❺     public String helloWorld() {
❻         delay( delayMillisSeconds: 1000 );
❼         log("inside helloWorld");
➋         return "hello world";
⌽     }
⌑
⌒     public String hello() {
⌓         delay( delayMillisSeconds: 1000 );
⌔         log("inside hello");
⌕         return "hello";
⌖     }
⌗
⌘     public String world() {
⌙         delay( delayMillisSeconds: 1000 );
⌚         log("inside world");
⌛         return " world!";
⌜     }
⌝
⌞     public CompletableFuture<String> worldFuture(String input) {
⌟         return CompletableFuture.supplyAsync(()->{
⌡             delay( delayMillisSeconds: 1000 );
⌢             return input+" world!";
⌣         });
⌤     }
⌥
⌦ }
⌧ }
```



```
import java.util.concurrent.CompletableFuture;
import static com.learnjava.util.LoggerUtil.log;

public class CompletableFutureHelloWorld {
    public static void main(String[] args) {
        HelloWorldService hws = new HelloWorldService();

        CompletableFuture.supplyAsync(() -> hws.helloWorld())
            .thenAccept((result) -> {
                log("Result is " + result);
            });
        log("Done!");
    }
}
```

theAccept takes the consumer as an input.



```
HelloWorldService hws = new HelloWorldService();

CompletableFuture.supplyAsync(() -> hws.helloWorld())
    .thenAccept((result) -> {
        log("Result is " + result);
});
log("Done!");
}
```

Run: CompletableFutureHelloWorld

```
/Library/Java/JavaVirtualMachines/jdk-11.0.4.jdk/Contents/Home/bin/java
[main] - Done!
Process finished with exit code 0
```

It is not printed at the result (at line number 18) because it won't wait for the supplier as supplier is taking 1 second . introduce some delay as shown below to get a response.

```

parallel-asynchronous - CompletableFutureHelloWorld.java [parallel-asynchronous.main]
parallel-asynchronous-using-java / src / main / java / com / learnjava / completablefuture / CompletableFutureHelloWorld / main
HelloWorldService.java  CompletableFutureHelloWorld.java  CompletableFuture.java  Supplier.java

12
13 public static void main(String[] args) {
14
15     HelloWorldService hws = new HelloWorldService();
16
17     CompletableFuture.supplyAsync(()->hws.helloWorld())
18         .thenAccept((result)->{
19             log("Result is " + result);
20         });
21     log("Done!");
22     delay(delayMilliseconds: 2000);
23 }

```

Run: CompletableFutureHelloWorld

```

/Library/Java/JavaVirtualMachines/jdk-11.0.4.jdk/Contents/Home/bin/java -javaagent:/Applications/IntelliJ IDEA CE
[main] - Done!
[ForkJoinPool.commonPool-worker-19] - inside helloWorld
[ForkJoinPool.commonPool-worker-19] - Result is hello world

Process finished with exit code 0

```

Other option is use of join method which blocks until whole computation is completed.

```

parallel-asynchronous - CompletableFutureHelloWorld.java [parallel-asynchronous.main]
parallel-asynchronous-using-java / src / main / java / com / learnjava / completablefuture / CompletableFutureHelloWorld / main
HelloWorldService.java  CompletableFutureHelloWorld.java  CompletableFuture.java  Supplier.java

12
13 public static void main(String[] args) {
14
15     HelloWorldService hws = new HelloWorldService();
16
17     CompletableFuture.supplyAsync(() -> hws.helloWorld())
18         .thenAccept((result) -> {
19             log("Result is " + result);
20         })
21         .join();
22     log("Done!");
23     //delay(2000);
}

```

Run: CompletableFutureHelloWorld

```

/Library/Java/JavaVirtualMachines/jdk-11.0.4.jdk/Contents/Home/bin/java -javaagent:/Applications/IntelliJ IDEA CE
[ForkJoinPool.commonPool-worker-19] - inside helloWorld
[ForkJoinPool.commonPool-worker-19] - Result is hello world
[main] - Done!

Process finished with exit code 0

```

thenApply()

- Completion Stage method
- Transform the data from one form to another
- Input is **Function** Functional Interface
- Returns CompletableFuture<T>

Example convert to uppercase.

```

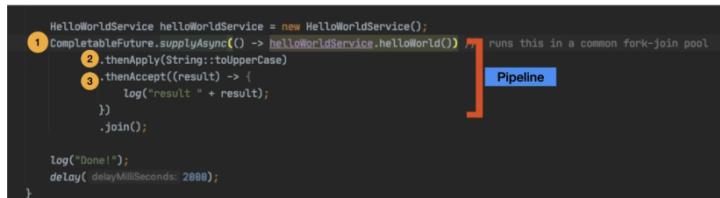
10 >     public class CompletableFutureHelloWorld {
11
12 >         public static void main(String[] args) {
13
14             HelloWorldService hws = new HelloWorldService();
15
16             CompletableFuture.supplyAsync(() -> hws.helloWorld())
17                 .thenApply((result)->result.toUpperCase())
18                 .thenAccept(result) -> {
19                     log("Result is : " + result);
20                 }
21                 .join();
22             log("Done!");
23         }
24     }

```

Debug: CompletableFutureHelloWorld

Process finished with exit code 0

CompletableFuture



How to perform unit testing.

```

15 >     @Test
16 >     void helloWorld() {
17 >         //given
18
19 >         //when
20 >         CompletableFuture<String> completableFuture = cfhw.helloWorld();
21
22 >         //then
23 >         completableFuture
24 >             .thenAccept(s -> {
25 >                 assertEquals(expected: "HELLO WORLD", s);
26 >             })
27 >             .join();
28
29

```

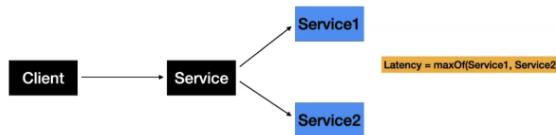
Run: CompletableFutureHelloWorldTest.helloWorld

Process finished with exit code 0

Combining independent Async Tasks using “thenCombine”

thenCombine()

- This is a Completion Stage Method
- Used to Combine Independent Completable Futures



- Takes two arguments
 - CompletionStage , BiFunction
- Returns a CompletableFuture

Traditional Coding

```
24
25     public String helloWorld_approach1() {
26
27         String hello = hws.hello();
28         String world = hws.world();
29
30         return hello + world;
31     }
```

How to do same using CompletableFuture

```
32
33     public String helloWorld_multiple_async_calls(){
34
35         CompletableFuture<String> hello = CompletableFuture.supplyAsync(()->hws.hello());
36         CompletableFuture<String> world = CompletableFuture.supplyAsync(()->hws.world());
37
38         return hello
39             .thenCombine(world, (h, w) -> h+w) // first, second
40             .thenApply(String::toUpperCase)
41             .join();
42     }
```

```

HelloWorldService.java
1 package com.example;
2
3 import java.util.concurrent.CompletableFuture;
4 import java.util.concurrent.ExecutionException;
5 import java.util.concurrent.TimeUnit;
6 import java.util.concurrent.TimeoutException;
7
8 public class HelloWorldService {
9     public String helloWorld() {
10         return "Hello World!";
11     }
12 }

```

```

CompletableFutureHelloWorld.java
1 package com.example;
2
3 import java.util.concurrent.CompletableFuture;
4 import java.util.concurrent.ExecutionException;
5 import java.util.concurrent.TimeUnit;
6 import java.util.concurrent.TimeoutException;
7
8 public class CompletableFutureHelloWorld {
9     public String helloWorld() {
10         return "Hello World!";
11     }
12 }

```

```

HelloWorldTest.java
1 package com.example;
2
3 import org.junit.Test;
4 import org.junit.runner.RunWith;
5 import org.junit.runners.JUnit4;
6
7 import static org.junit.Assert.assertEquals;
8
9 @RunWith(JUnit4.class)
10 public class HelloWorldTest {
11     @Test
12     void helloworld_multiple_async_calls() {
13         //given
14         String helloWorld = cfhw.helloworld_multiple_async_calls();
15
16         //when
17         assertEquals("HELLO WORLD!", helloWorld);
18     }
19 }

```

```

CompletableFutureHelloWorldTest.java
1 package com.example;
2
3 import org.junit.Test;
4 import org.junit.runner.RunWith;
5 import org.junit.runners.JUnit4;
6
7 import static org.junit.Assert.assertEquals;
8
9 @RunWith(JUnit4.class)
10 public class CompletableFutureHelloWorldTest {
11     @Test
12     void helloworld_multiple_async_calls() {
13         CompletableFuture<String> hello = CompletableFuture.supplyAsync(() -> hws.hello());
14         CompletableFuture<String> world = CompletableFuture.supplyAsync(() -> hws.world());
15
16         CompletableFuture<String> hiCompletableFuture = hello.thenCombine(world, (h, w) -> h+w) // first, second
17             .thenApply(String::toUpperCase)
18             .join();
19
20         assertEquals("HELLO WORLD! HI COMPLETABLEFUTURE!", hiCompletableFuture.get());
21     }
22 }

```

Another example

```

public String helloworld_3_async_calls(){
    startTimer();

    CompletableFuture<String> hello = CompletableFuture.supplyAsync(() -> hws.hello());
    CompletableFuture<String> world = CompletableFuture.supplyAsync(() -> hws.world());
    CompletableFuture<String> hiCompletableFuture = CompletableFuture.supplyAsync(() ->{
        delay(1000);
        return " Hi CompletableFuture!";
    });

    String hw= hello
        .thenCombine(world, (h, w) -> h+w) // first, second
        .thenCombine(hiCompletableFuture, (previous,current) -> previous+current)
        .thenApply(String::toUpperCase)
        .join();

    timeTaken();
    return hw;
}

```

Test case

```

//when
String helloWorld = cfhw.helloworld_3_async_calls();

//then
assertEquals("HELLO WORLD! HI COMPLETABLEFUTURE!", helloWorld);
}

```

thenCompose()

- Completion Stage method
- Transform the data from one form to another
- Input is **Function** Functional Interface
- Deals with functions that return CompletableFuture
- thenApply deals with Function that returns a value
- Returns CompletableFuture<T>

```

public CompletableFuture<String> worldFuture(String input)
{
    return CompletableFuture.supplyAsync(() ->{
        delay(1000);
        return input + " world!";
    });
}

```

```

68     public CompletableFuture<String> helloWorld_thenCompose(){
69
70         return CompletableFuture.supplyAsync(hws::hello)
71             .thenCompose((previous)-> hws.worldFuture(previous))
72                 .thenApply(String::toUpperCase)
73                     /*.join()*/;
74     }
75

```

The screenshot shows an IDE interface with two code snippets side-by-side. The left snippet is a test method:

```

53     @Test
54     void helloWorld_thenCompose() {
55         //given
56         startTimer();
57         //when
58         CompletableFuture<String> completableFuture = cfhw.helloWor
59
60         //then
61         completableFuture
62             .thenAccept(s -> {
63                 assertEquals(expected: "HELLO WORLD!", s);
64             })
65             .join();
66
67         timeTaken(); // The cursor is here
68     }
69 }
70

```

The right snippet is the implementation of the `helloWorld_thenCompose` method:

```

69
70
71
72
73
74
75
76
77     public CompletableFuture<String> helloWorld_thenCompose(){
78         return CompletableFuture.supplyAsync(hws::hello)
79             .thenCompose((previous)-> hws.worldFuture(previous))
80                 .thenApply(String::toUpperCase)
81                     /*.join()*/;
82     }
83
84
85
86
87
88
89

```

A yellow highlight bar spans from the end of the first snippet's `timeTaken()` call to the start of the second snippet's `helloWorld_thenCompose` method, indicating the flow of execution.

thenCompose takes 2 seconds. This is expected why because theCompose its input task to complete first then only its block will execute.

The screenshot shows the IDE's run tab with the following details:

- Run:** CompletetureHelloWorldTest.helloWo... (Test Result 2s 56ms)
- Tests passed:** 1 of 1 test - 2s 56 ms
- Compiler:** 2s 56 ms (Library/Java/JavaVirtualMachines/jdk-11.0.4.jdk/Contents/Home/bin/java ...)
- Output:**
 - [ForkJoinPool.commonPool-worker-19] - inside hello
 - [main] - Total Time Taken : 2053