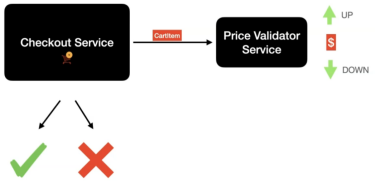
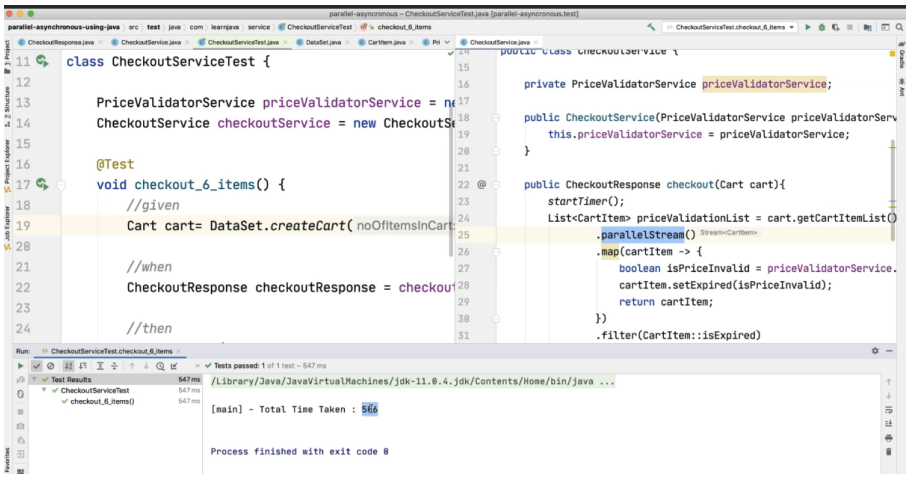
Overview of the Retail Checkout Service

Checkout Service(BackEnd)




```java
public CheckoutResponse checkout(Cart cart){
    startTimer();
    List<CartItem> priceValidationList = cart.getCartItemList() List<CartItem>
            .stream() Stream<CartItem>
            .map(cartItem -> {
                boolean isPriceInvalid = priceValidatorService.isCartItemInvalid(cartItem);
                cartItem.setExpired(isPriceInvalid);
                return cartItem;
            })
            .filter(CartItem::isExpired)
            .collect(Collectors.toList());

    if(priceValidationList.size()>0){
        return new CheckoutResponse(CheckoutStatus.FAILURE, priceValidationList);
    }

    timeTaken();
     return new CheckoutResponse(CheckoutStatus.SUCCESS);
}

}
```

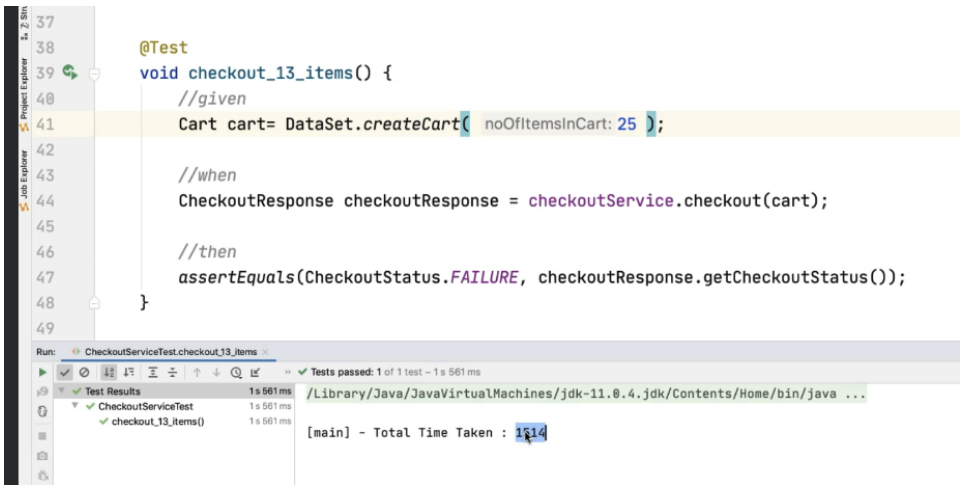
```java
class CheckoutServiceTest {

    PriceValidatorService priceValidatorService = ne
    CheckoutService checkoutService = new CheckoutSe

    @Test
    void checkout_6_items() {
        //given
        Cart cart= DataSet.createCart( noOfItemsInCart:

        //when
        CheckoutResponse checkoutResponse = checkou

        //then
        assertEquals(CheckoutStatus.SUCCESS, checkou
    }
}
```



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

```java
    @Test
    void no_Of_cores() {
        //given

        //when
        System.out.println("no of cores :" + Runtime.getRuntime().availableProcessors());

        //then
    }

    @Test
```

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



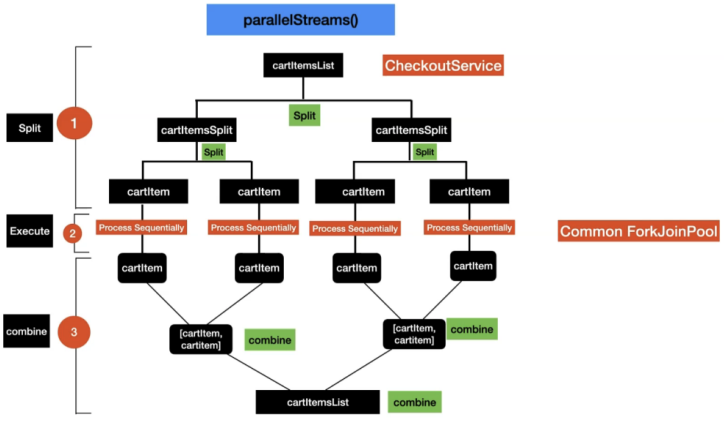
ParallelStreams - How it works ?

- **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

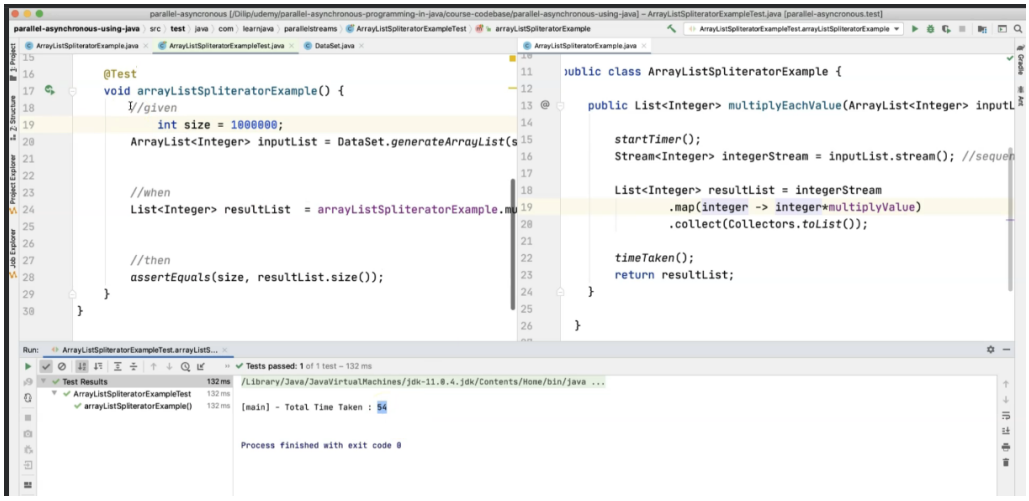
- Data source is split in to multiple chunks by the Spliterator
- Each and every collection has a different Spliterator Implementation
- Performance differ based on the implementation

Multiply each value in the collection by a user passed value

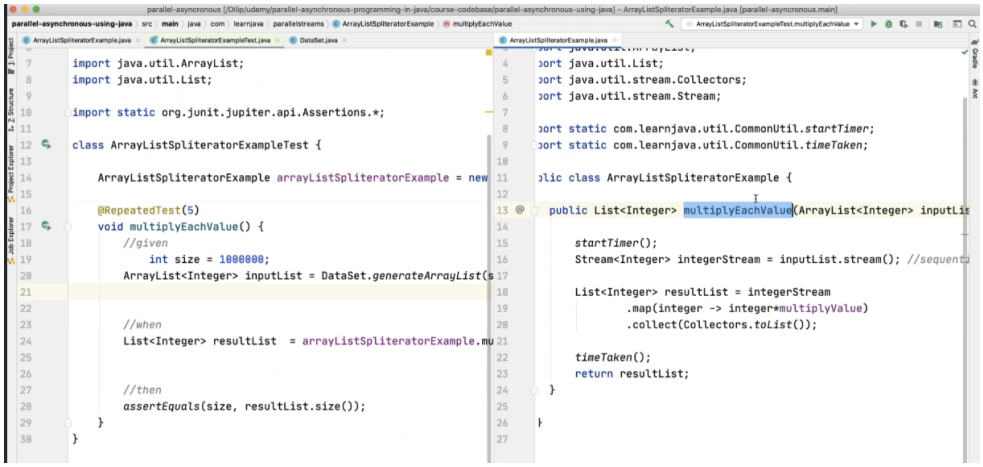
[1, 2, 3, 4] -> [2, 4, 6, 8]

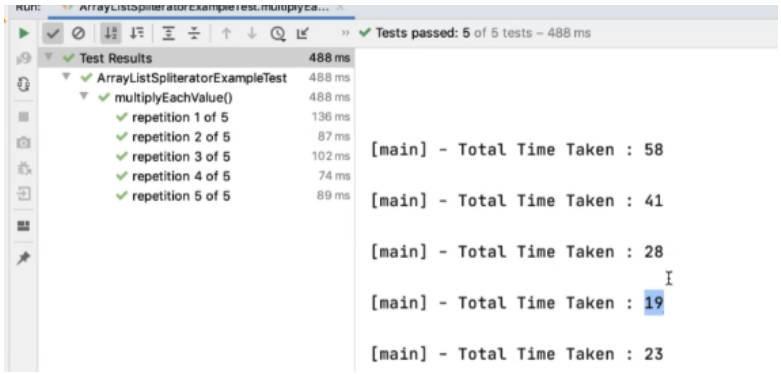
Value * 2

For one lack record it took 54 mins.

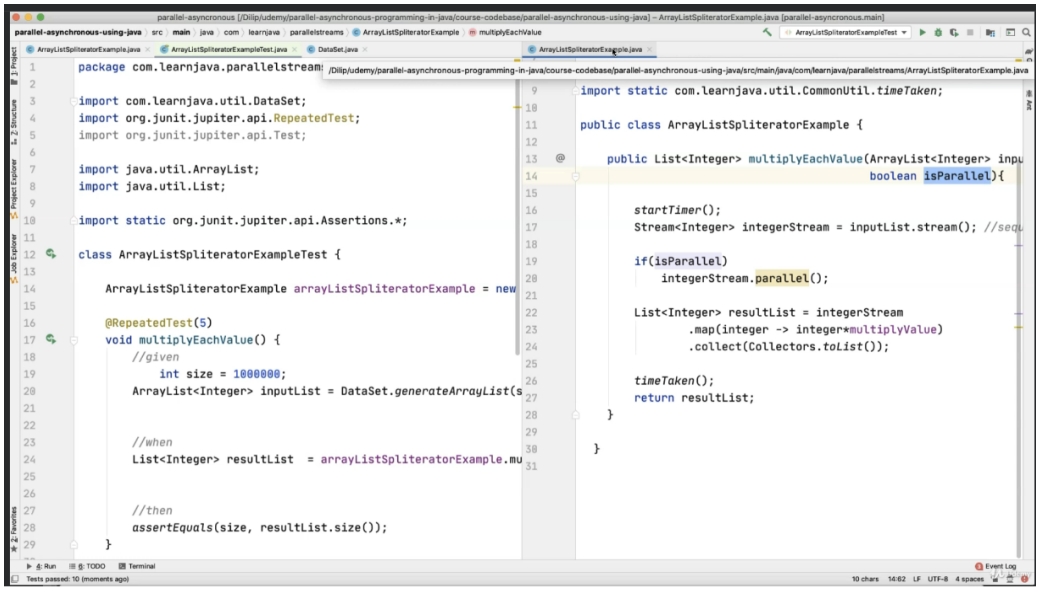


Repeat the test using RepeatedAnnotation as shown below due to JVM and chache it will reduce the performance time as shown below.



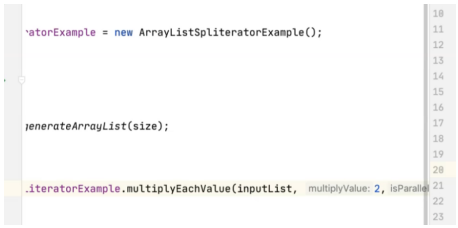


Convert sequential to parallel. By default streams are sequential



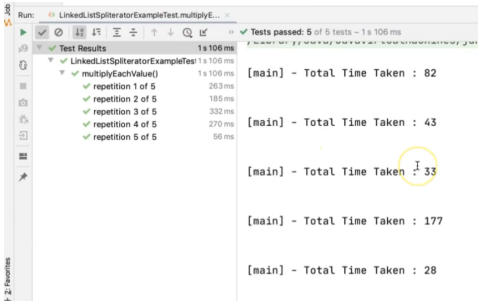
```
package com.learnjava.parallelstreams;

import com.learnjava.util.DataSet;
import org.junit.jupiter.api.RepeatedTest;
import org.junit.jupiter.api.Test;

import java.util.ArrayList;
import java.util.List;

import static org.junit.jupiter.api.Assertions.*;

class ArrayListSpliteratorExampleTest {

    ArrayListSpliteratorExample arrayListSpliteratorExample = new

    @RepeatedTest(5)
    void multiplyEachValue() {
        //given
            int size = 1000000;
        ArrayList<Integer> inputList = DataSet.generateArrayList(s


        //when
        List<Integer> resultList  = arrayListSpliteratorExample.mu


        //then
        assertEquals(size, resultList.size());
    }
```

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

public class ArrayListSpliteratorExample {

    public List<Integer> multiplyEachValue(ArrayList<Integer> inpu
                                           boolean isParallel){

        startTimer();
        Stream<Integer> integerStream = inputList.stream(); //sequ

        if(isParallel)
            integerStream.parallel();

        List<Integer> resultList = integerStream
                .map(integer -> integer*multiplyValue)
                .collect(Collectors.toList());

        timeTaken();
        return resultList;
    }

}
```



```
atorExample = new ArrayListSpliteratorExample();

enerateArrayList(size);

.iteratorExample.multiplyEachValue(inputList,  multiplyValue: 2, isParalle
```

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



```
Run:  LinkedListSpliteratorExampleTest.multiplyE...
Tests passed: 5 of 5 tests – 1 s 106 ms
Test Results                                1 s 106 ms
  LinkedListSpliteratorExampleTest          1 s 106 ms
    multiplyEachValue()                     1 s 106 ms
      repetition 1 of 5                     263 ms
      repetition 2 of 5                     185 ms
      repetition 3 of 5                     332 ms
      repetition 4 of 5                     270 ms
      repetition 5 of 5                     56 ms

[main] - Total Time Taken : 82

[main] - Total Time Taken : 43

[main] - Total Time Taken : 33

[main] - Total Time Taken : 177

[main] - Total Time Taken : 28
```

LikedList 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()

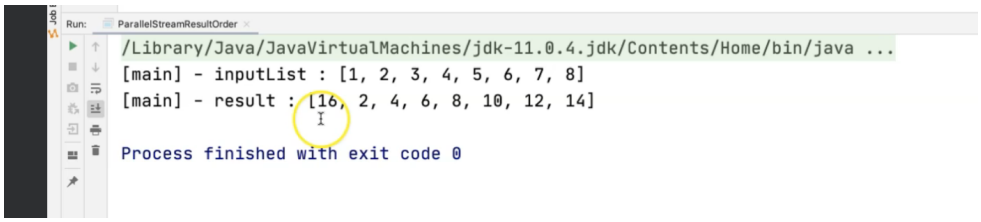
Parallel Streams - Final Computation Result Order

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

```java
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);

    }
}
```

```
/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
```

```java
    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);

    }
```



Collect( ) vs Reduce( )

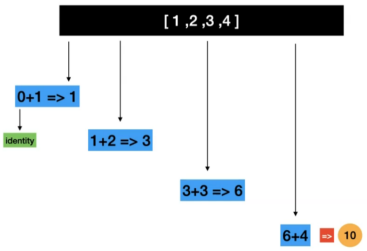
Collect

- 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

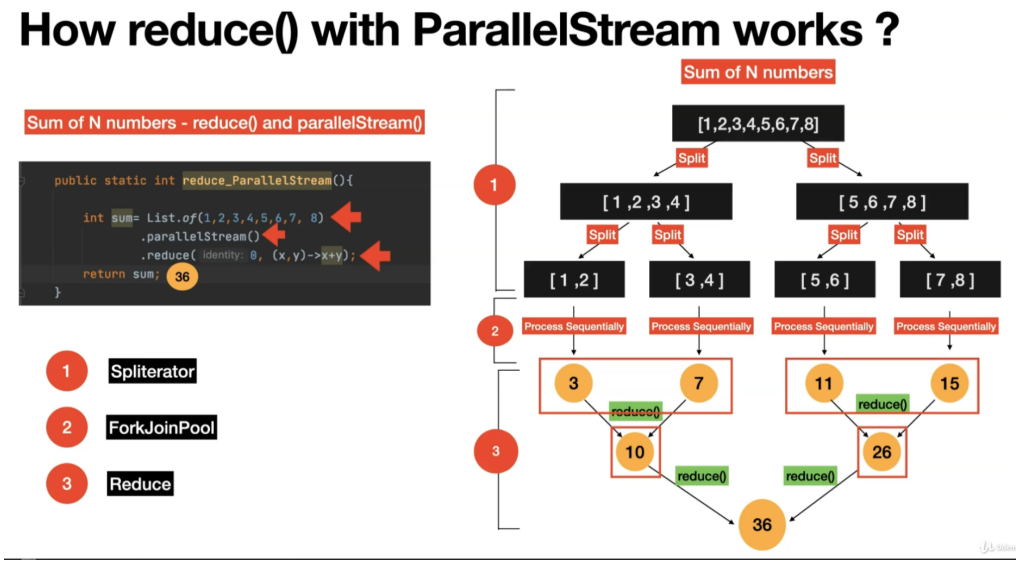
- Part of Streams API
- Used as a terminal operation in **Streams API**
- Produces a single result
- Result is produced in a immutable fashion
- Reduce the computation into a single value
  - Sum, Multiplication
  - Example
    - Sum -> reduce(0.0, **(x , y)->x+y**)
    - Multiply -> reduce(1.0, **(x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

How reduce() works ?

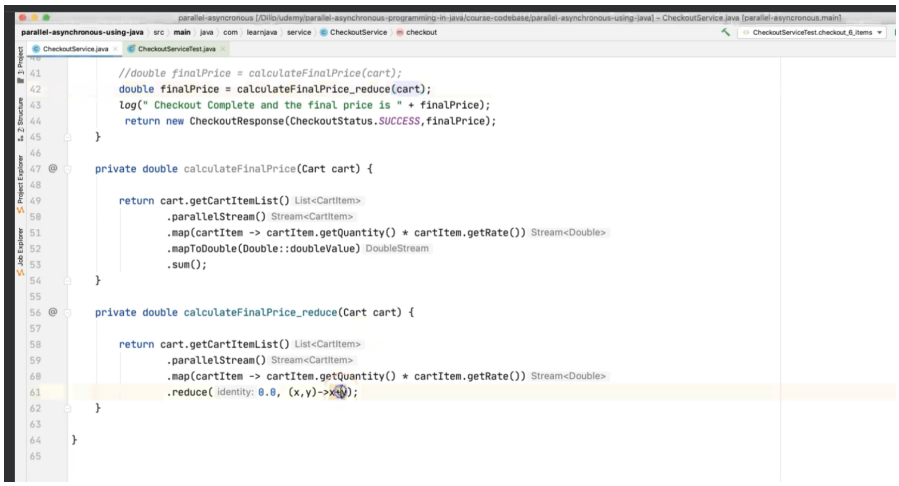



```
public static int reduce(){
    int sum= List.of(1,2,3,4)
            .stream()
            .reduce( identity: 0, (x,y)->x+y);
    return sum;  10
}
```

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

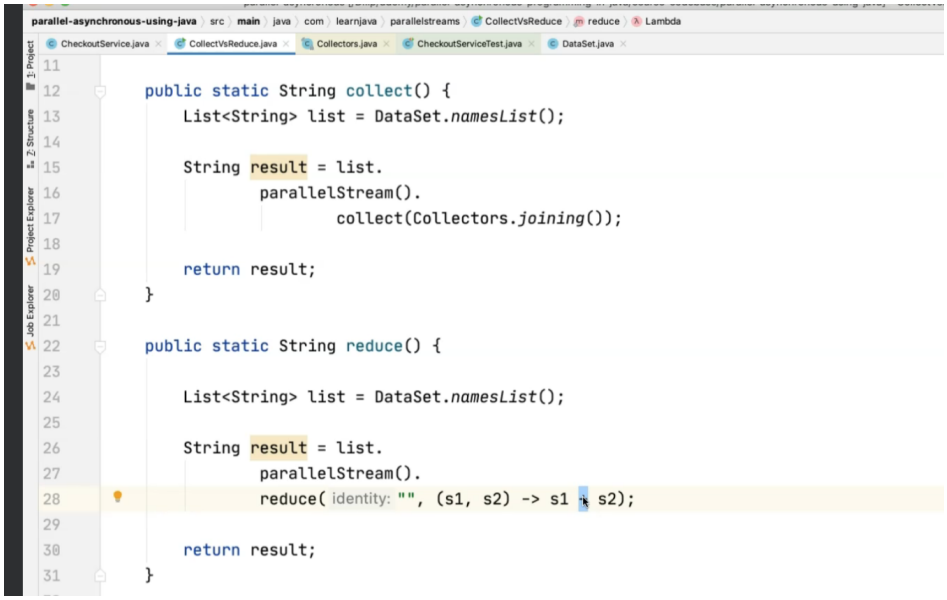


Calculate final price of checkout out service using sum and reduce methods use either one.



```
        //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.



Identity in reduce()

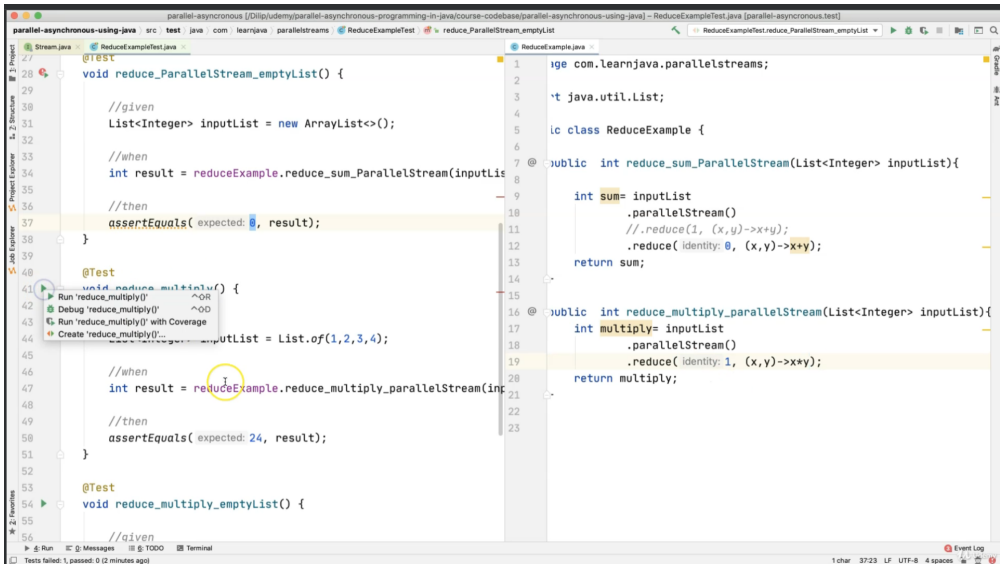
- Identity gives you the same value when its used in the computation
  - Addition: **Identity = 0**
    - 0 + 1 => 1
    - 0 + 20 => 20
  - Multiplication : **Identity = 1**
    - 1 * 1 => **1**
    - 1 * 20 => **20**

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.

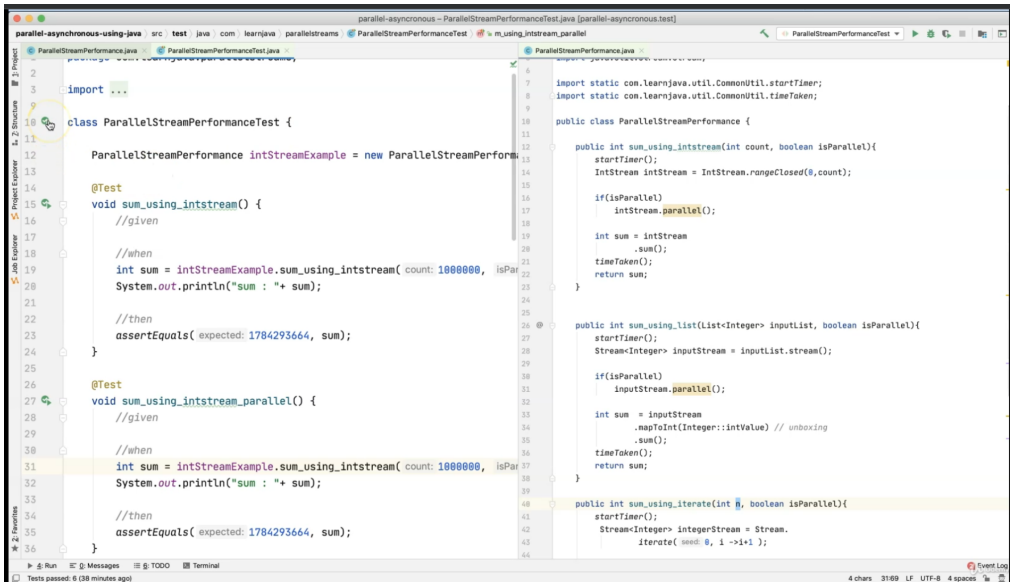


Try to change the identity value (right side) and run the test case left. We may see the weared result because when you run in parallel - split execute and combine will happen - for each split wrong identity value will be taken and calculation starts.

Parallel Stream Operations & Poor Performance

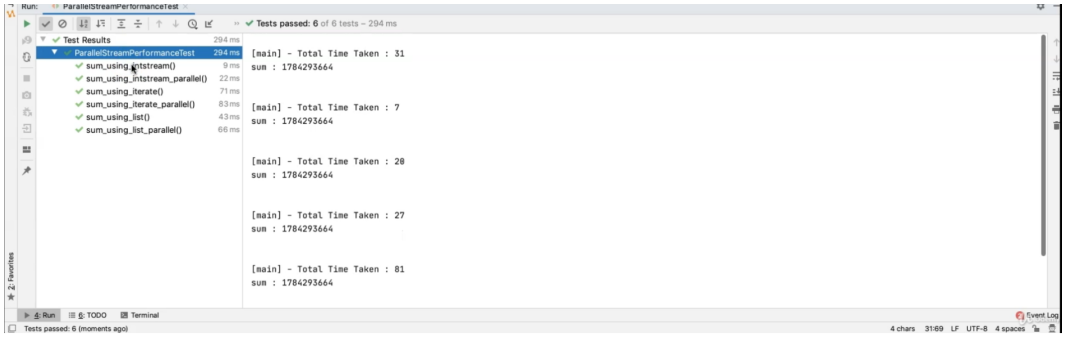
- Stream Operations that perform poor
- Impact of **Boxing** and **UnBoxing** when it comes to parallel Streams
  - **Boxing** -> Converting a Primitive Type to Wrapper class equivalent
    - **1** -> **new Integer(1)** ⬅
  - **UnBoxing** -> Converting a Wrapper class to Primitive equivalent
    - **new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

And its example and test cases provided below.



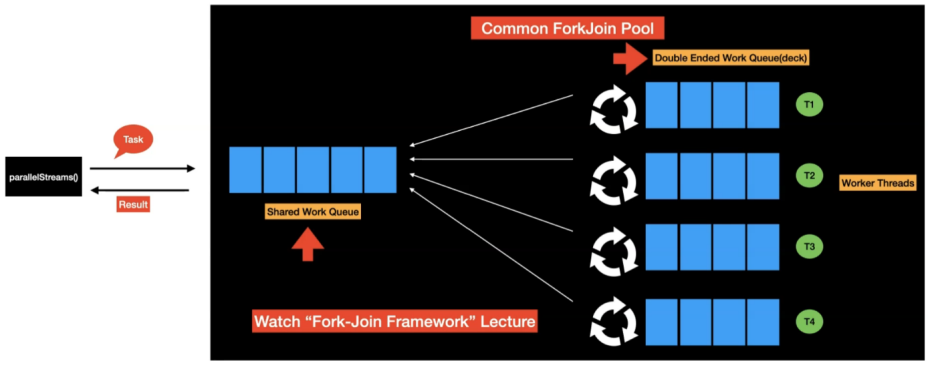
```
import ...

class ParallelStreamPerformanceTest {

    ParallelStreamPerformance intStreamExample = new ParallelStreamPerforma

    @Test
    void sum_using_intstream() {
        //given

        //when
        int sum = intStreamExample.sum_using_intstream( count: 1000000,  isPar
        System.out.println("sum : "+ sum);

        //then
        assertEquals( expected: 1784293664, sum);
    }

    @Test
    void sum_using_intstream_parallel() {
        //given

        //when
        int sum = intStreamExample.sum_using_intstream( count: 1000000,  isPar
        System.out.println("sum : "+ sum);

        //then
        assertEquals( expected: 1784293664, sum);
    }
```

```
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( seed: 0, i ->i+1 );
```



```
Test Results                                294 ms
  ParallelStreamPerformanceTest             294 ms
    sum_using_intstream()                     9 ms
    sum_using_intstream_parallel()           22 ms
    sum_using_iterate()                      71 ms
    sum_using_iterate_parallel()             83 ms
    sum_using_list()                         43 ms
    sum_using_list_parallel()                66 ms

[main] - Total Time Taken : 31
sum : 1784293664

[main] - Total Time Taken : 7
sum : 1784293664

[main] - Total Time Taken : 20
sum : 1784293664

[main] - Total Time Taken : 27
sum : 1784293664

[main] - Total Time Taken : 81
sum : 1784293664
```

Common ForkJoin Pool



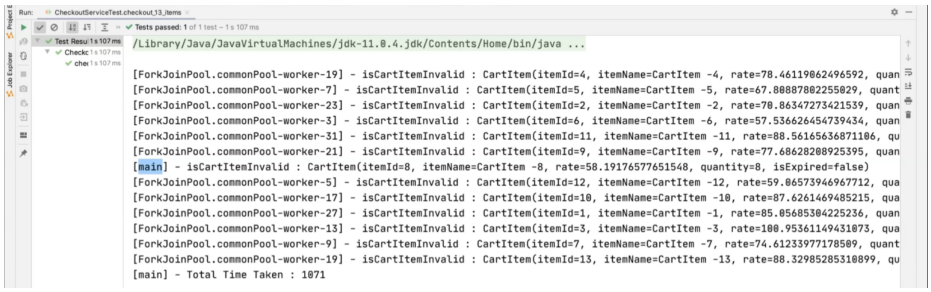
Common ForkJoin Pool

- Common ForkJoin Pool is used by:
  - ParallelStreams
  - CompletableFuture
    - Completable Future 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**

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



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
```

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 ?

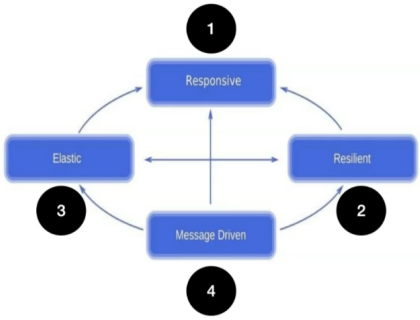
- Parallel Streams
  - Split ⬅
  - Execute
  - Combine ⬅
- Data set is small
- Auto Boxing and Unboxing doesn't perform better
- Stream API operators -> **iterate**(), **limit**()

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 through messages in a event-driven style



We can categorize all the method 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

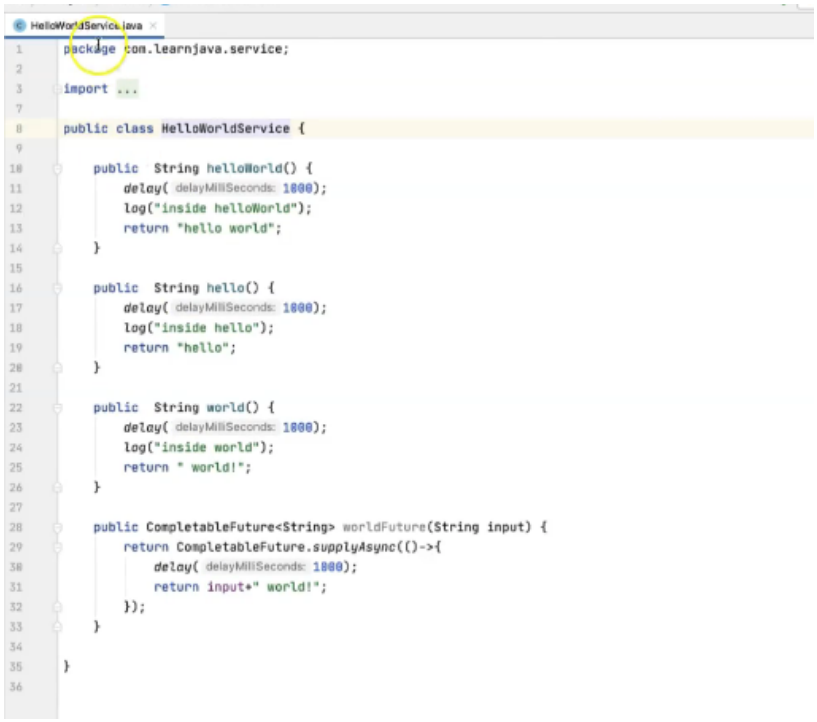
CompletableFuture

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>



```java
package com.learnjava.service;

import ...

public class HelloWorldService {

    public  String helloWorld() {
        delay( delayMilliSeconds: 1000);
        log("inside helloWorld");
        return "hello world";
    }

    public  String hello() {
        delay( delayMilliSeconds: 1000);
        log("inside hello");
        return "hello";
    }

    public  String world() {
        delay( delayMilliSeconds: 1000);
        log("inside world");
        return " world!";
    }

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

}
```

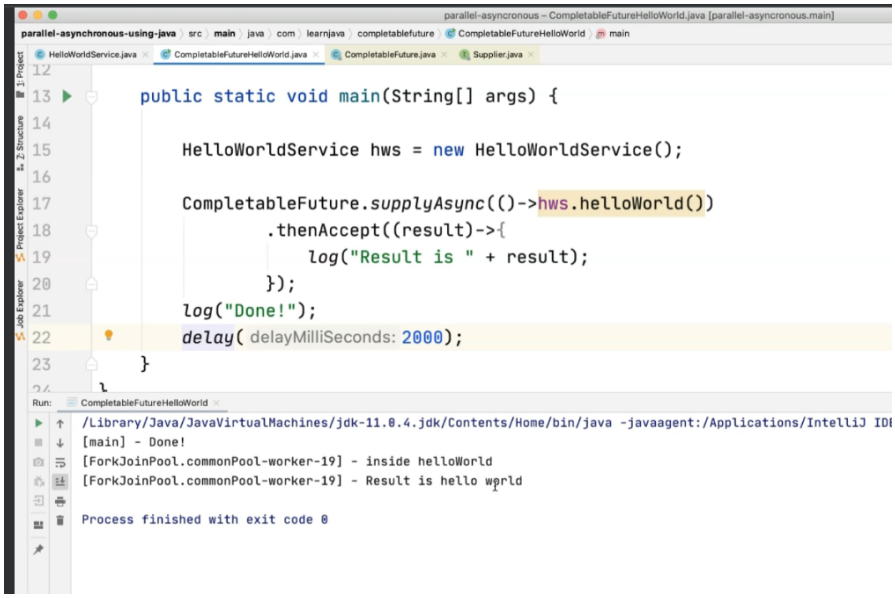
```java
import java.util.concurrent.CompletableFuture;

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.

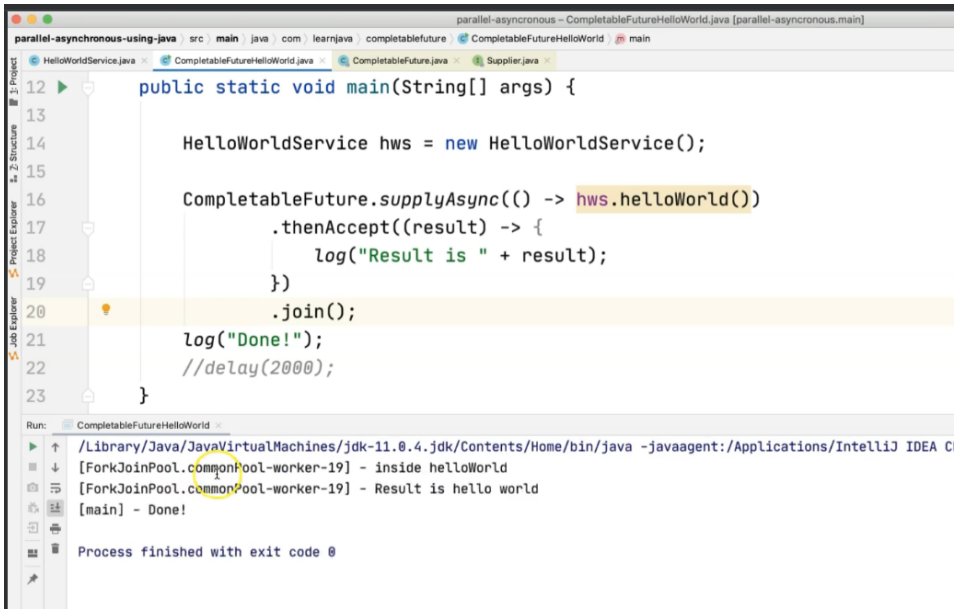
```java
        HelloWorldService hws = new HelloWorldService();

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

```
/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.



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



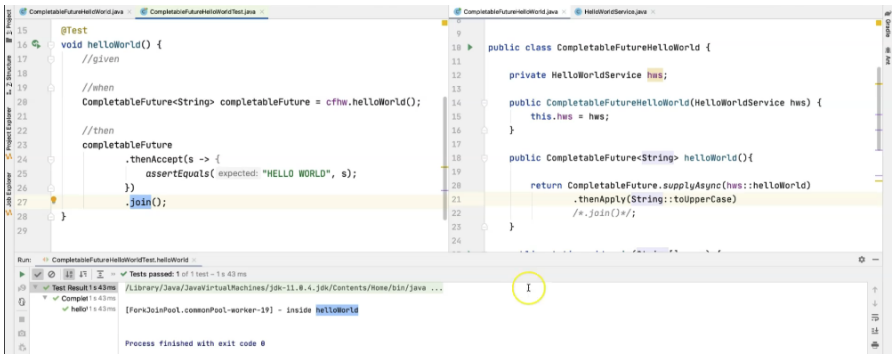
thenApply()

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

Example convert to uppercase.

CompletableFuture

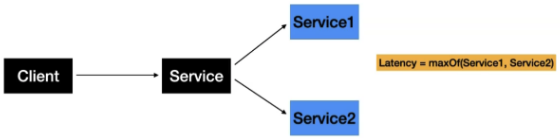
How to perform unit testing.



Combing independent Async Tasks using “thenCombine"

thenCombine()

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



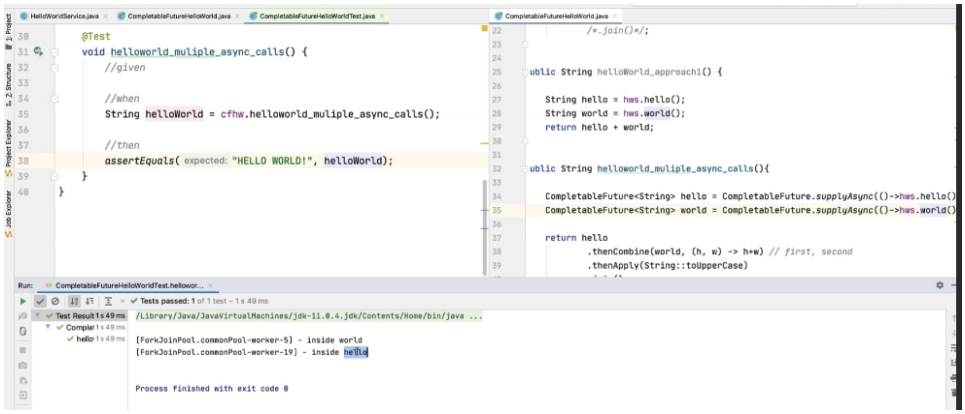
- Takes two arguments
  - CompletionStage , BiFunction
- Returns a CompletableFuture

Traditional Coding

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

How to do same using CompletableFuture

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



Another example

Test case

thenCompose()

- Completion Stage method
- Transform the data from one form to another
- Input is **Function** Functional Interface

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

- Deals with functions that return CompletableFuture
  - thenApply deals with Function that returns a value
- Returns CompletableFuture<T>

```java
public CompletableFuture<String> helloWorld_thenCompose(){

    return CompletableFuture.supplyAsync(hws::hello)
            .thenCompose((previous)-> hws.worldFuture(previous))
            .thenApply(String::toUpperCase)
            /*.join()*/;
}
```

```java
@Test
void helloWorld_thenCompose() {
    //given
     startTimer();
    //when
    CompletableFuture<String> completableFuture = cfhw.helloWor

    //then
    completableFuture
            .thenAccept(s -> {
                assertEquals( expected: "HELLO WORLD!", s);
            })
            .join();

    timeTaken();
}
}
```

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