Correction Java Concurrency - 0

D -

```
OCP
null
```

get() method throws 2 checked exceptions: InterruptedException and ExecutionException. And future.get() returns Object and it can be type-casted to Integer, so no compilation error.

run() method of MyThread prints ‘OCP’ on to the console and doesn’t return anything, hence get() method of Future object returns null. null can be easily assigned to Integer.

Hence ‘System.out.println(tmp);’ prints null on to the console.

Correction Java Concurrency - 1

B - It will print numbers from 1 to 10 in ascending order

IntStream.rangeClosed(1, 10) returns a sequential ordered IntStream but parallel() method converts it to a parallel stream.

forEachOrdered() will processes the elements of the stream in the order specified by its source (Encounter order), regardless of whether the stream is sequential or parallel, hence given code prints 1 to 10 in ascending order.

Correction Java Concurrency - 2

A - HELLO and DONE will be printed but printing order is not fixed.

‘es.submit(new Printer());’ is asynchronous call, hence 2 threads(main and thread from pool) always run in asynchronous mode.

Thread execution depends upon underlying OS/JVM’s thread scheduling mechanism.

HELLO and DONE will be printed but their order cannot be predicted.

Correction Java Concurrency - 3

A - Program will print 5 in any time greater than or equal to 3 secs.

There is a fixed thread pool of 10 threads, this means 10 threads can run in parallel.

invokeAll() method causes the current thread (main) to wait, until all the Callable instances finish their execution.

This means ‘System.out.println(futures.size());’ will execute once all the 5 tasks are complete.

Important point to note here is that all the Callable instances are executed in parallel. Even if all the tasks start at one instant, there will always be some overhead for sleeping thread to become running.

In most of the cases it will be in nanoseconds/milliseconds. Hence overall time will be slightly greater than 3 secs.

Correction Java Concurrency - 4

C - new Player(cb);

‘new CyclicBarrier(2, match);’ means 2 threads must call await() so that Cyclic Barrier is tripped and barrier action is executed, this means run() method of thread referred by match is invoked and this will print expected output on to the console.

‘new Player(cb);’ assigns passed CyclicBarrier instance to ‘cb’ property of the Player class and it also invokes start() method so that this Player thread becomes Runnable. run() method is invoked later which invokes cb.await() method.

One more ‘cb.await();’ call is needed to get the expected output.

‘p1.start();’ => Thread referred by p1 is already started in the constructor, calling start() again throws IllegalThreadStateException.

‘new Player(cb);’ => Just like previous statement, it will invoke the await() method on cb and CyclicBarrier will be tripped. This is correct option.

‘new Player(cb).start();’ => ‘new Player(cb)’ will invoke the await() method on cb and CyclicBarrier will be tripped. ‘Match is starting…’ will be printed on to the console. But calling start() method again throws IllegalThreadStateException and program is not terminated successfully.

‘cb.await();’ => await() method throws 2 checked exceptions: InterruptedException and BrokenBarrierException and these are not handled in the main method. Hence this will cause compilation error. If these 2 exceptions are handled inside main, then this could be one of the correct options.

‘new Player();’ => No-argument constructor of Player class doesn’t assign value to ‘cb’ and doesn’t invoke start() method as well. Hence, it will not have any affect.

Correction Java Concurrency - 5

D - It will print three digits 321 but order can be different

AtomicInteger’s getAndDecrement() method will first retrieve the value and then decrement and it happens atomically.

So during the execution of getAndDecrement() method, another thread cannot execute getAndDecrement() method. This means output will have 3 digits 3, 2 and 1.

But execution of threads depend on OS/JVM implementation, hence order of output can be different.

Correction Java Concurrency - 6

A - es.submit(new Printer());

D - es.execute(new Printer());

ExecutorService interface has following methods:

Future<?> submit(Runnable task);

Future submit(Runnable task, T result);

Future submit(Callable task);

Hence, ‘es.submit(new Printer());’ is one of the correct options.

ExecutorService interface extends Executor interface and it has ‘void execute(Runnable command);’ method. Hence, ‘es.execute(new Printer());’ is the 2nd correct option.

Correction Java Concurrency - 7

C - RecursiveTask

E - Recursive Task

F - RecursiveTask

RecursiveTask is a generic class which extends from ForkJoinTask class.

RecursiveTask declares compute() method as: ‘protected abstract V compute();’

In the given code overriding method returns Long, so classes from which class Task can extend are: RecursiveTask, RecursiveTask [co-variant return type in overriding method] and RecursiveTask [co-variant return type in overriding method].

RecursiveAction is a non-generic class which extends from ForkJoinTask class.

RecursiveAction declared compute() method as: ‘protected abstract void compute();’

In the given code overriding method returns Long, hence RecursiveAction can’t be used as super class of Task.

Correction Java Concurrency - 8

C -

```
99
98
```

‘Executors.newSingleThreadExecutor();’ creates an Executor that uses a single worker thread.

‘new MyCallable(100);’ creates MyCallable object and initializes its property i to 100.

‘es.submit(callable)’ invokes the call method and returns a Future object containing Integer value 99. get() method prints this value on to the console.

‘es.submit(callable)’ is invoked again for the same MyCallable instance, call method returns a Future object containing Integer value 98. get() method prints this value on to the console.

Correction Java Concurrency - 9

A -

```
[Melon, Apple, Banana, Mango]
[Apple, Banana]
```

‘new CopyOnWriteArrayList<>(list1);’ creates a thread-safe list containing the elements of list1. list1 and list2 are not linked, hence changes made to one list doesn’t affect other list.

CopyOnWriteArrayList allows add/set/remove while iterating through the list. On every modification, a fresh copy of underlying array is created, leaving the iterator object unchanged.

‘Melon’ and ‘Mango’ are deleted from list2.

list1 refers to [Melon, Apple, Banana, Mango] and list2 refers to [Apple, Banana].

Correction Java Concurrency - 10

A - It can print any number between 0 and 1000

As ‘list.add(i);’ is invoked inside synchronized method, hence adding to list is thread-safe.

shutdown() doesn’t wait for previously submitted tasks to complete execution and as all submitted tasks execute in parallel hence list can contain any number of elements between 1 and 1000.

It will be a rare case when none of the tasks were completed and shutdown() invoked before that, but still that’s a chance and if it happens then list will have no elements.

Correction Java Concurrency - 11

D - livelock

In deadlock, threads’ state do not change but in livelock threads’ state change constantly. In both cases, process hangs.

Correction Java Concurrency - 12

C - “Match is Starting…” is never printed on to the console and program waits indefinitely.

ExecutorService can handle 1 thread, but 2 threads are submitted. The first thread calls await() and wait endlessly.

CyclicBarrier needs 2 threads to call await() method but as this is not going to happen, hence the program waits endlessly.

‘Match is starting…’ is never printed on to the console and program waits indefinitely.

NOTE: To trip the CyclicBarrier, replace ‘Executors.newFixedThreadPool(1);’ with ‘Executors.newFixedThreadPool(2);’.

Correction Java Concurrency - 13

E - None of the other options

Given program will print 3 digits but all 3 digits can be different or same as variable i is not accessed in synchronized manner.

Correction Java Concurrency - 14

B - It will always print 51

First element that matches the filter is 51.

In this case, base stream is sequential ordered IntStream (it has specific Encounter Order), hence findFirst() method will always return 51 regardless of whether the stream is sequential or parallel.

Correction Java Concurrency - 15

D-

```
Melon
Apple
Banana
Mango
```

removeIf method accepts Predicate, hence no compilation error.

Enhanced for loop uses an internal iterator and as CopyOnWriteArrayList is used, add/set/remove operations while iterating doesn't cause any exception.

In first iteration, removeIf method removes 'Melon' and 'Mango' from the list. On every modification, a fresh copy of underlying array is created, leaving the iterator object unchanged. 'Melon' is printed on to he console.

In 2nd iteration, removeIf method doesn't remove anything as list doesn't contain any element starting with 'M'. But iterator still has 4 elements. 2nd iteration prints 'Apple' on to the console. And so on.

Correction Java Concurrency - 16

B - It will print numbers form 1 to 10 but not in any specific order

IntStream.rangeClosed(1, 10) returns a sequential ordered IntStream but parallel() method converts it to a parallel stream.

Hence, forEach method doesn’t guarantee the order for printing numbers.

Correction Java Concurrency - 17

C - It will print 15 on to the console

RecursiveAction class has an abstract method ‘void compute()’ and Adder class correctly overrides it.

new Adder(1, 5); instantiate an Adder object with from = 1, to = 5 and total = 0.

5 - 1 = 4, which is less than or equal to 4 hence else block will not get executed. compute() method will simply add the numbers from 1 to 5 and will update the total field of Adder object created at Line 34.

So when ‘System.out.println(adder.total);’ is executed, it prints 15 on to the console.

If Line 34 is replaced with ‘Adder adder = new Adder(1, 6);’, then else block of compute() method will create more Adder objects and total filed of those Adder objects will be updated and not the Adder object created at Line 34. So, in this case output will be 0.

Best solution would be to add below code as the last statement in the else block so that total field of Adder object created at Line 34 is updated successfully:

total = first.total + second.total;

It will give the expected result in all the cases.

Correction Java Concurrency - 18

B - Compilation error

get() method throws 2 checked exceptions: InterruptedException and ExecutionException.

As the given code neither handles these exceptions using catch block nor declares these exceptions in throws clause, hence call to get() method causes compilation failure.

Correction Java Concurrency - 19

C - HELLO

submit method of ExecutorService interface is overloaded: submit(Runnable) and submit(Callable). Both Runnable and Callable are Functional interfaces.

Given lambda expression ‘() -> “HELLO”’, it accepts no parameter and returns a String, hence it matches with the call() method implementation of Callable interface.

f.get() would return ‘HELLO’.

Notice, get() method throws 2 checked exceptions InterruptedException and ExecutionException and main method declares to throw these exceptions, hence no compilation error.

Correction Java Concurrency - 20

B -

```
PRINT
null
10
```

Method reference ‘Test::print’ is for the run() method implementation of Runnable and ‘Test::get’ is for the call() method implementation of Callable.

Future<?> is valid return type for both the method calls. get() method throws 2 checked exceptions: InterruptedException and ExecutionException, hence given code compiles fine.

get() method waits for task completion, hence ‘PRINT’ will be printed first.

future1.get() returns null and future2.get() returns 10.

Correction Java Concurrency - 21

A - false

Parallel streams internally use fork/join framework only, so there is always an overhead of splitting the tasks and joining the results.

Parallel streams improves performance for streams with large number of elements, easily splittable into independent operations and computations are complex.

Correction Java Concurrency - 22

A - Yes

All streams in Java implements BaseStream interface and this interface has parallel() and sequential() methods.

Hence all streams can either be parallel or sequential.

Correction Java Concurrency - 23

C - Output cannot be predicted

list.parallelStream() returns a parallel stream.

Method reference ‘RES::append’ is same as lambda expression ‘s -> RES.append(s)’. NOTE: In the lambda expression as static variable RES is used hence given code suffers from race condition.

Output cannot be predicted in this case.

Correction Java Concurrency - 24

B - Compilation error

ExecutorService interface extends Executor interface and it has ‘void execute(Runnable command);’. Runnable is a Functional interface which has single abstract method ‘public abstract void run();’.

Given lambda expression, ‘() -> “HELLO”’ returns a String and it doesn’t match with the implementation of run() method whose return type is void. Hence it causes compilation error.

Return type of execute method is void, hence another reason for compilation error is that result of ex.execute(…) cannot be assigned to Future.

Correction Java Concurrency - 25

B - The program doesn’t terminate but prints following:

```
CALL
null
```

Callable is of Void type and call() method returns null.

‘es.submit(new Caller(“Call”));’ creates a Caller object and invokes the call method. This method prints ‘CALL’ on to the console and returns null.

‘System.out.println(future.get());’ prints ‘null’ on to the console.

As ‘es.shutdown();’ is not invoked, hence program doesn’t terminate.

Correction Java Concurrency - 26

A - ForkJoinTask

B - RecursiveAction

C - RecursiveTask

ForkJoinPool class declares the invoke() method as: public T invoke(ForkJoinTask task) {...}

Instance of ForkJoinTask class can be easily passed to the invoke method.

As RecursiveAction and RecursiveTask extend from ForkJoinTask, hence their instances can also be passed to the invoke method.

Correction Java Concurrency - 27

C - It will print 0 on to the console.

RecursiveAction class has an abstract method ‘void compute()’ and Adder class correctly overrides it.

new Adder(1, 20); instantiate an Adder object with from = 1, to = 20 and total = 0.

20 - 1 is not less than or equal to 4 hence else block will create multiple Adder objects, and compute() method will be invoked on these objects. More Adder objects will be created further.

Statement ‘total+=sum;’ will update the total filed of these objects but not the adder object created at Line 34. So when ‘System.out.println(adder.total);’ is executed, it prints 0 on to the console.

To get the expected output (which is sum of numbers from 1 to 20), add below code as the last statement in the else block so that total field of Adder object created at Line 34 is updated successfully:

total = first.total + second.total;

Correction Java Concurrency - 28

A - It will print 15 on to the console

stream –> {1, 2, 3, 4, 5}.

stream.parallel() returns a parallel stream.

To understand, 'reduce((x, y) -> x + y)' is equivalent to:

```
boolean foundAny = false;
int result = null;
for (int element : this stream) {
 if (!foundAny) {
     foundAny = true;
     result = element;
 }
 else
     result = accumulator.applyAsInt(result, element);
}
return foundAny ? OptionalInt.of(result) : OptionalInt.empty();
```

result will be initialized to 1st element of the stream and output will be the result of '1 + 2 + 3 + 4 + 5', which is 15.

The whole computation may run in parallel, but parallelism doesn't impact final result. In this case as there are only 5 numbers, hence it is an overhead to use parallelism.

reduce((x, y) -> x + y) returns OptionalInt and it has getAsInt() method.

Correction Java Concurrency - 29

B - System.out.println(ai.addAndGet(1) + “:” + ai);

C - System.out.println(ai.incrementAndGet() + “:” + ai.get());

AtomicInteger overrides toString() method hence when ai is passed to println method, its value is printed.

‘ai.addAndGet(1)’ first adds 1 (10+1) and returns 11. ‘ai’ also prints 11.

‘ai.getAndAdd(1)’ first returns 10 and then adds 1.

‘ai.getAndIncrement()’ first returns 10 and then adds 1.

‘ai.incrementAndGet()’ increments the value by 1 (10+1) and returns 11.

‘ai.incrementAndGet(1)’ causes compilation failure.

‘ai.get()’ returns the current value of ai.

Correction Java Concurrency - 30

C - Output cannot be predicted

Line 13 is changing the state of list object and hence it should be avoided in parallel stream. You can never predict the order in which elements will be added to the stream.

Line 13 and Line 15 doesn't run in synchronized manner, hence as the result, output of Line 17 may be different from that of Line 15.

On my machine below is the output of various executions:

Execution 1:

```
5427163 5412736
```

Execution 2:

```
5476231 5476123
```

Execution 3:

```
5476231 5476231
```

Correction Java Concurrency - 31

B - Output cannot be predicted

reduce method in Stream class is declared as: T reduce(T identity, BinaryOperator accumulator)

By checking the reduce method, ‘reduce(““, String::concat)’, we can say that:

Identity is String type, accumulator is BinaryOperator type.

Though you may always get false but result cannot be predicted as identity value (“_“) used in reduce method is not following an important rule.

For each element ‘t’ of the stream, accumulator.apply(identity, t) is equal to t.

‘String::concat’ is equivalent to lambda expression ‘(s1, s2) -> s1.concat(s2);’.

For 1st element of the stream, “A” accumulator.apply(“_“,”A”) results in “_A”, which is not equal to “A” and hence rule is not followed.

s1 will always refer to “_AEIOU” but s2 may refer to various possible string objects depending upon how parallel stream is processed.

s2 may refer to “_A_E_I_O_U” or “_AE_I_OU” or “_AEIOU”. So output cannot be predicted.

Correction Java Concurrency - 32

A - It will always print true

reduce method in Stream class is declared as: U reduce(U identity, BiFunction<U,? super T,U> accumulator, BinaryOperator combiner)

By checking the reduce method 'reduce("", (i, s) -> i + s, (s1, s2) -> s1 + s2)', we can say that:

Identity is String type, accumulator is BiFunction<String, ? super Integer, String> type, combiner is BinaryOperator type.

To get consistent output, there are requirements for reduce method arguments:

1. The identity value must be an identity for the combiner function. This means that for all u, combiner(identity, u) is equal to u.

   As u is of String type, let's say u = "X", combiner("","X") = "X". Hence, u is equal to combiner("","X"). First rule is obeyed.

2. The combiner function must be compatible with the accumulator function; for all u and t, the following must hold:

   combiner.apply(u, accumulator.apply(identity, t)) == accumulator.apply(u, t).

   Let's consider, u = "Y", t is element of Stream, say t = 1, identity = "".

   combiner.apply(u, accumulator.apply(identity, t))

   = combiner.apply("Y", accumulator.apply("", 1))

   = combiner.apply("Y", "1")

   = "Y1"

   and

   accumulator.apply(u, t)

   = accumulator.apply("Y", 1)

   = "Y1"

   Hence, combiner.apply(u, accumulator.apply(identity, t)) == accumulator.apply(u, t). 2nd rule is also followed.

3. The accumulator operator must be associative and stateless. Operator + is associative and lambda expression is stateless. 3rd rule is followed.

4. The combiner operator must be associative and stateless. Operator + is associative and lambda expression is stateless. 4th rule is followed.

As all the rules are followed in this case, hence str1 refers to “12345678910” and str2 refers to “12345678910”

Correction Java Concurrency - 33

B - stream().parallel()

D - parallelStream( )

Collection interface has a default method stream() to return a sequential() stream for the currently executing Collection.

Collection interface has a default method parallelStream() to return a parallel stream for the currently executing Collection.

Stream class has parallel() method to convert to parallel stream and sequential() method to convert to sequential stream.

isParallel() method of Stream class returns true for parallel Stream.

list is a Collection. So,

list.stream() returns a sequential stream and list.stream().parallel() returns a parallel stream. list.stream().parallel().isParallel() returns true.

stream() returns a sequential stream and list.stream().isParallel() returns false.

list.parallel() causes compilation error.

list.parallelStream() returns a parallel stream.
list.parallelStream().isParallel() returns true.

Correction Java Concurrency - 34

D -

```
9
null
```

Method submit is overloaded to accept both Callable and Runnable: Future submit(Callable task); and Future<?> submit(Runnable task);

Both returns a Future object. call() method of MyCallable returns 9 and get() method returns this value.

run() method of MyThread doesn’t return anything, hence get() method of Future object returns null.

Correction Java Concurrency - 35

B - return second.compute() + first.join();

After invoking fork() on 1st subtask, it is necessary to invoke join() on 1st subtask and compute() on 2nd subtask. Hence ‘return first.join();’ and ‘return second.compute();’ are not valid options.

The order of execution of calling join() and compute() on divided subtasks is important in a fork/join framework.

For the statement ‘return first.join() + second.compute();’ as join() is called so each task waits for its completion before starting execution of a new thread. Even though you will get the correct output but it is not utilizing the fork/join framework efficiently.

If you are interested in finding out the time difference then I will suggest to modify main(String[]) method as below and run the program for both the statements separately.

```java
public static void main(String[] args) {
    LocalTime start = LocalTime.now();
    ForkJoinPool pool = new ForkJoinPool(THREADS);
    long sum = pool.invoke(new AdderTask(1, LIMIT));
    System.out.printf("sum of the number from %d to %d is %d %n",
        1, LIMIT, sum);
    LocalTime end = LocalTime.now();
    System.out.println(ChronoUnit.NANOS.between(start, end));
}
```

Correction Java Concurrency - 36

D - HELLO

ExecutorService interface extends Executor interface and it has ‘void execute(Runnable command);’.

Runnable is a Functional interface which has single abstract method ‘public abstract void run();’.

Given lambda expression, ‘() -> System.out.println(“HELLO”)’ accepts no parameter and doesn’t return anything, hence it matches with the implementation of run() method.

‘HELLO’ is printed on to the console.

Correction Java Concurrency - 37

C - It will always print JAVA on to the console

reduce method in Stream class is declared as: ‘Optional reduce(BinaryOperator accumulator);’

‘String::concat’ is equivalent to lambda expression ‘(s1, s2) -> s1.concat(s2);’.

By checking the reduce method, ‘reduce(String::concat)’ we can say that:

accumulator is BinaryOperator type.

To get consistent output, accumulator must be associative and stateless. concat is associative as (s1.concat(s2)).concat(s3) equals to s1.concat(s2.concat(s3)). Given method reference syntax is stateless as well.

Hence, reduce will give the same result for both sequential and parallel stream.

As per Javadoc, given reduce method is equivalent to:

```
boolean foundAny = false;
T result = null;
for (T element : this stream) {
 if (!foundAny) {
     foundAny = true;
     result = element;
 }
 else
     result = accumulator.apply(result, element);
}
return foundAny ? Optional.of(result) : Optional.empty();
```

This means, output will be JAVA.

Correction Java Concurrency - 38

D - Book[9781976704033:25.98]

books –> [Book[9781976704031:9.99], Book[9781976704032:15.99]].

books.stream() –> {Book[9781976704031:9.99], Book[9781976704032:15.99]}. It is sequential stream.

To understand, first reduce() method can be somewhat written as:

```
Book book = new Book("9781976704033", 0.0);
for(Book element : stream) {
    book = accumulator.apply(book, element);
}
return book;
```

Above code is just for understanding purpose, you can't iterate a stream using given loop.

apply(book, element) invokes the code of lambda expression:

```
(b1, b2) -> {
    b1.price = b1.price + b2.price;
    return new Book(b1.isbn, b1.price);
}
```

This means, price of Book object referred by book will be the addition of 9.99 and 15.99, which is 25.98 and its isbn will remain 9781976704033.

b –> Book[9781976704033:25.98].

Above book is added to the books list.

books –> [Book[9781976704031:9.99], Book[9781976704032:15.99], Book[9781976704033:25.98]].

books.parallelStream().reduce((x, y) -> x.price > y.price ? x : y) returns Optional, the Book object inside Optional has highest price.

Hence, output is: Book[9781976704033:25.98].

Correction Java Concurrency - 39

D - Invoke compute() on 2nd subtask and then join() on 1st subtask

After invoking fork() on 1st subtask, it is necessary to invoke join() on 1st subtask and compute() on 2nd subtask.

The order of execution of calling join() and compute() on divided subtasks is important in a fork/join framework.

First invoke compute() on 2nd subtask and then join() on 1st subtask.

Correction Java Concurrency - 40

A - It will always print true

reduce method in Stream class is declared as: T reduce(T identity, BinaryOperator accumulator)

By checking the reduce method, ‘reduce(““, String::concat)’, we can say that:

Identity is String type, accumulator is BinaryOperator type.

‘String::concat’ is equivalent to lambda expression ‘(s1, s2) -> s1.concat(s2);’.

To get consistent output, there are requirements for reduce method arguments:

1. For each element ‘t’ of the stream, accumulator.apply(identity, t) is equal to t.

   For 1st element of the stream, “A” accumulator.apply(““,”A”) results in “A”, which is equal to “A” and hence 1st rule is followed.

2. The accumulator operator (concat) in this case must be associative and stateless.

   concat is associative as (s1.concat(s2)).concat(s3) equals to s1.concat(s2.concat(s3)). Given method reference syntax is stateless as well.

As both the rules are followed, hence reduce will give the same result for both sequential and parallel stream.

Correction Java Concurrency - 41

E - Output of Line 15 can be predicted

Line 13 is changing the state of list object and hence it should be avoided in parallel stream. You can never predict the order in which elements will be added to the stream.

Line 13 and Line 15 doesn’t run in synchronized manner, hence as the result, output of Line 17 may be different from that of Line 15.

forEachOrdered() will processes the elements of the stream in the order specified by its source, regardless of whether the stream is sequential or parallel.

As forEachOrdered() method is used at Line 15, hence Line 15 will always print ‘1234567’ on to the console.

On my machine below is the output of various executions:

Execution 1:

```
1234567 1352764
```

Execution 2:

```
1234567 6514327
```

Execution 3:

```
1234567 1732645
```

Correction Java Concurrency - 42

C - RecursiveTask

E - RecursiveAction

There are no classes in concurrent package with the names ‘Recursion’, ‘RecursionAction’ and ‘RecursionTask’.

Both RecursiveAction and RecursiveTask can be used to define a recursive task.

RecursiveTask is used to define tasks which returns a value whereas RecursiveAction is used to define tasks that don’t return a value.