Concurrency

synchronized

synchronized on class

```
  String synchronizedOnClass() {
    //this is a synchronized block on the class
    synchronized (SynchronizedMethodAndBlock.class) {
      return message();
    }
  }
```

Threads

Runnable

Invalid Runnable

```
Runnable r = ()->5; //does not compile
```

Start thread

```
Thread t = new Thread(() -> System.out.println("hello"));
t.start();
System.out.println("completed");
```

Basic Threads

Thread interrupt

`interrupt()` only impacts a thread in a **WAITING** or **TIMED_WAITING** state. Calling `interrupt()` on a thread in a **NEW** or **RUNNABLE** state has no impact. mInterrupting a thread that is not alive not have any effect.

Parallel Stream

The `forEachOrdered()` method will process the elements in the order in which they are stored in the stream.

Reduce

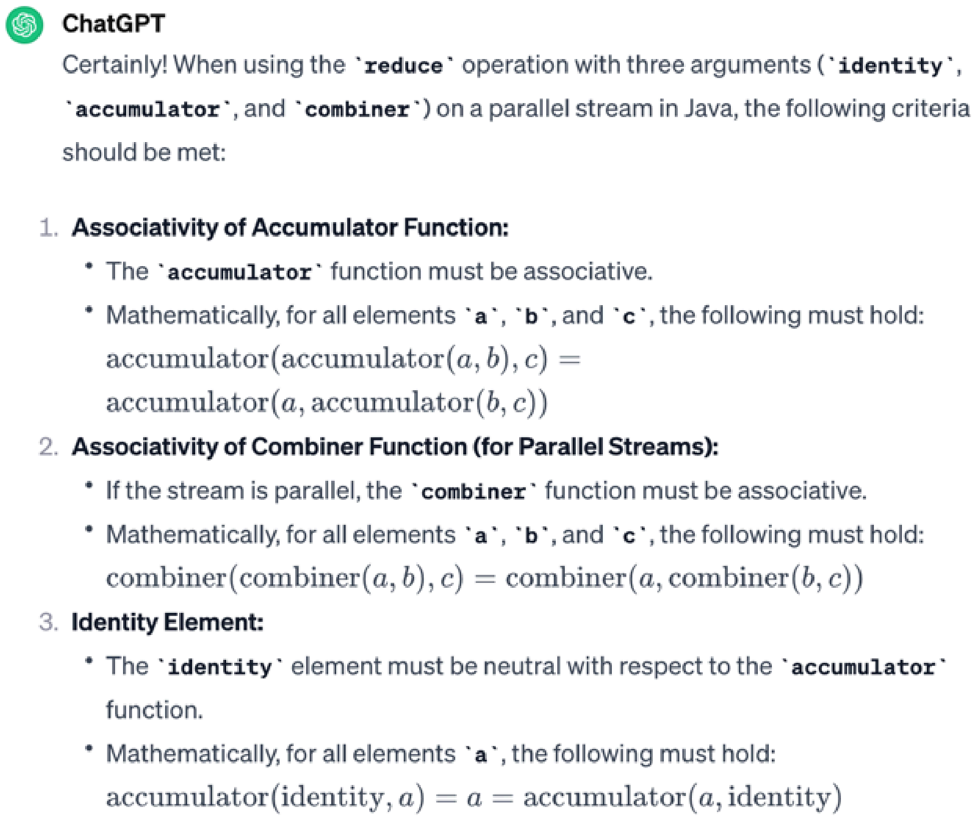
```
Stream<T> stream

<U> U reduce(U identity,
    BiFunction<U,? super T,U>accumulator,
    BinaryOperator<U> combiner)
```

Rules

- identity: same type as the return type
- accumulator: 1st parameter as the return type, 2nd parameter as the source stream
- combiner: 1st and 2nd parameter as the return type

Performs a reduction on the elements of this stream, using the provided identity, accumulation and combining functions. The accumulator and combiner must be:

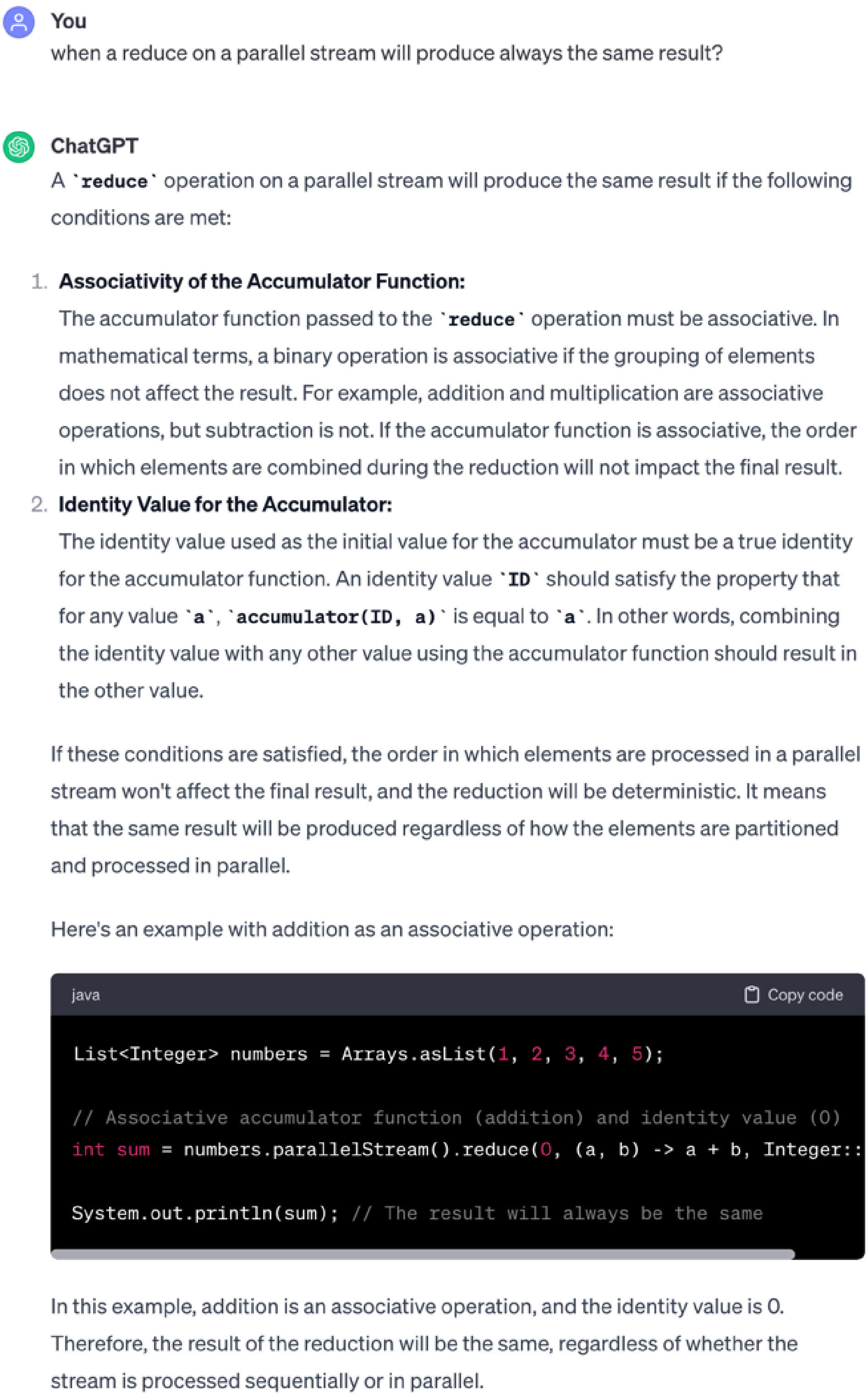
- associative;
- non-interfering;
- stateless.



ChatGPT

Certainly! When using the `reduce` operation with three arguments (`identity`, `accumulator`, and `combiner`) on a parallel stream in Java, the following criteria should be met:

1. **Associativity of Accumulator Function:**
    - The `accumulator` function must be associative.
    - Mathematically, for all elements `a`, `b`, and `c`, the following must hold: $\text{accumulator}(\text{accumulator}(a, b), c) = \text{accumulator}(a, \text{accumulator}(b, c))$
2. **Associativity of Combiner Function (for Parallel Streams):**
    - If the stream is parallel, the `combiner` function must be associative.
    - Mathematically, for all elements `a`, `b`, and `c`, the following must hold: $\text{combiner}(\text{combiner}(a, b), c) = \text{combiner}(a, \text{combiner}(b, c))$
3. **Identity Element:**
    - The `identity` element must be neutral with respect to the `accumulator` function.
    - Mathematically, for all elements `a`, the following must hold: $\text{accumulator}(\text{identity}, a) = a = \text{accumulator}(a, \text{identity})$



You

when a reduce on a parallel stream will produce always the same result?

ChatGPT

A `reduce` operation on a parallel stream will produce the same result if the following conditions are met:

1. **Associativity of the Accumulator Function:**
   The accumulator function passed to the `reduce` operation must be associative. In mathematical terms, a binary operation is associative if the grouping of elements does not affect the result. For example, addition and multiplication are associative operations, but subtraction is not. If the accumulator function is associative, the order in which elements are combined during the reduction will not impact the final result.
2. **Identity Value for the Accumulator:**
   The identity value used as the initial value for the accumulator must be a true identity for the accumulator function. An identity value `ID` should satisfy the property that for any value `a`, `accumulator(ID, a)` is equal to `a`. In other words, combining the identity value with any other value using the accumulator function should result in the other value.

If these conditions are satisfied, the order in which elements are processed in a parallel stream won't affect the final result, and the reduction will be deterministic. It means that the same result will be produced regardless of how the elements are partitioned and processed in parallel.

Here's an example with addition as an associative operation:

```java
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

// Associative accumulator function (addition) and identity value (0)
int sum = numbers.parallelStream().reduce(0, (a, b) -> a + b, Integer::

System.out.println(sum); // The result will always be the same
```

In this example, addition is an associative operation, and the identity value is 0. Therefore, the result of the reduction will be the same, regardless of whether the stream is processed sequentially or in parallel.

Parallel vs Sequential reduce

Sequential

To obtain an equivalent sequential stream of an existing parallel stream.

```
List<Integer> sequentialResult = numbers.parallelStream()
    .sequential()  // This makes the stream sequential
    .collect(Collectors.toList());
```

Methods of Executors

```
    ExecutorService executorService = Executors.newSingleThreadExec
    ExecutorService cachedExecutorService = Executors.newCachedThre
    ExecutorService fixedExecutorService = Executors.newFixedThread
    //scheduled methods
    ScheduledExecutorService scheduledExecutorService = Executors.n
    ScheduledExecutorService singleScheduledExecutorService = Execu
```

ExecutorService

Methods

```
Future<?> submit(Runnable task)

<T> Future<T> submit(Callable<T> task)

void execute(Runnable command) //from the parent Executor
```

Calling get on Future<?> returns null.

InvokeAny

```
List<Callable<String>> list = buildActions();
String message;

//NOTE: it only accepts Collection<Callable<T>>
//note that it returns not a future but the data
message = executorService.invokeAny(list);
```

Future

```
V get(long timeout, TimeUnit unit) throws InterruptedException, Exe
```

Mind to catch the checked exception when you call `Future.get(...)`

```
try {
    result = future.get(1, TimeUnit.SECONDS);
    //mind the three checked exceptions to be caught
    } catch (InterruptedException | ExecutionException | TimeoutExce
    throw new RuntimeException(e);
}
```

Shutdown

```
void executor(){
  ExecutorService executorService = Executors.newSingleThreadExecut
  try{
      executorService.submit(printInventory);
    }finally{
      //if you do not call this the method
      //the code will run but never terminate,
      executorService.shutdown();
    }
}
```

ScheduledExecutorService

```
public interface ScheduledExecutorService extends ExecutorService {
```

```
ScheduledFuture<?> schedule(Runnable command, long delay, TimeUnit
<V> ScheduledFuture<V> schedule(Callable<V> callable, long delay, T
SheduledFuture<?> scheduleAtFixedRate(Runnable command, long initia
ScheduledFuture<?> scheduleWithFixedDelay(Runnable command, long in
```

- `scheduleAtFixedRate()`can result in the same action being executed by two threads at the same time.
- `scheduleAtFixedRate()` takes 4 args

Concurrent Collections

Sorted Concurrent Collections

If you see `SkipList` as part of a concurrent class name, it means it is sorted in some way.

- `ConcurrentSkipListSet`
- `ConcurrentSkipListMap`

Requirements for Parallel Reduction with collect()

- The stream is parallel.
- The parameter of the collect() operation has the `Characteristics.CONCURRENT` characteristic.
- Either the stream is unordered or the collector has the characteristic `Characteristics.UNORDERED`.

```
<R> R collect(Supplier<R> supplier,
 BiConsumer<R,? super T> accumulator,
 BiConsumer<R,R> combiner)
```

Example

Check Characteristic

CyclicBarrier

```
public int await() throws InterruptedException, BrokenBarrierExcept
```

Mind that `InterruptedException` and `BrokenBarrierException` are both checked exceptions.
CyclicBarrier examples

Lock

tryLock

Acquires the lock if it is free within the given waiting time:

```
 if (lock.tryLock(1, TimeUnit.MINUTES)) {..}
```

Returns immediately

```
 if (lock.tryLock() {..}
```

tryLock

parallel stream

`findFirst()` method always returns the first element on an ordered stream, regardless if it is serial or parallel!

Identifying Threading Problems

There are three types of *liveness* issues with which you should be familiar: deadlock, starvation, and livelock.

Deadlock

Deadlock occurs when two or more threads are blocked forever.
Deadlock 1
Deadlock 2

Starvation

Starvation occurs when a single thread is perpetually denied access to a shared resource.

Livelock

Livelock is a form of starvation where two or more threads are active but conceptually blocked forever.