Working with Kafka Advanced Consumers

Kafka Consumers Advanced

Working with consumers in Java

Details and advanced topics

Objectives Advanced Kafka Producers

- Using auto commit / Turning auto commit off
- Managing a custom partition and offsets
  - ConsumerRebalanceListener
- Manual Partition Assignment (***assign***() vs. ***subscribe***())
- Consumer Groups, aliveness
  - ***poll***() and ***session.timeout.ms***
- Consumers and message delivery semantics
  - at most once, at least once, exactly once
- Consumer threading models
  - thread per consumer, consumer with many threads (both)

Java Consumer Examples Overview

- Rewind Partition using ***ConsumerRebalanceListener*** and ***consumer.seekX***() full Java example
- At-Least-Once Delivery Semantics full Java Example
- At-Most-Once Delivery Semantics full Java Example
- Exactly-Once Delivery Semantics full Java Example
    - full example storing topic, partition, offset in RDBMS with JDBC
    - Uses ***ConsumerRebalanceListener*** to restore to correct location in log partitions based off of database records
    - Uses transactions, rollbacks if commitSync fails
- Threading model Java Examples
    - Thread per Consumer
    - Consumer with multiple threads (***TopicPartition*** management of ***commitSync***())
- Implement full “priority queue” behavior using Partitioner and manual partition assignment
    - Manual Partition Assignment (***assign***() vs. ***subscribe***())

KafkaConsumer

- A Consumer client
  - consumes records from Kafka cluster
- Automatically handles Kafka broker failure
  - adapts as topic partitions leadership moves in Kafka cluster
- Works with Kafka broker to form consumers groups and load balance consumers
- Consumer maintains connections to Kafka brokers in cluster
- Use ***close()*** method to not leak resources
- NOT thread-safe

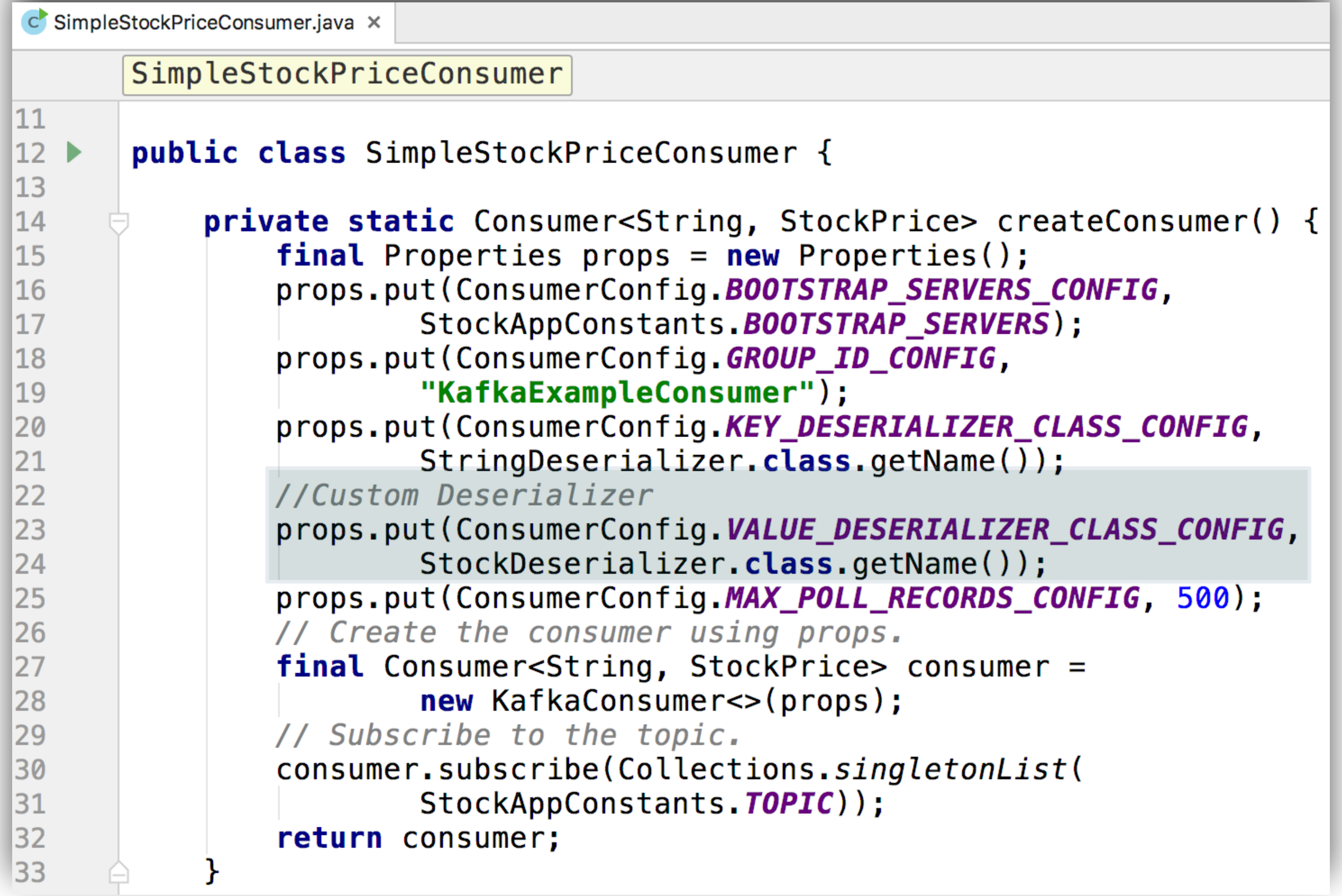
StockPrice App to demo Advanced Producer

- ***StockPrice*** - holds a stock price has a name, dollar, and cents
- ***StockPriceConsumer*** **-** Kafka ***Consumer*** that consumes ***StockPrices***
- ***StockAppConstants*** - holds topic and broker list
- ***StockPriceDeserializer*** **-** can deserialize a ***StockPrice*** from ***byte[]***

Kafka Consumer Example

- ***StockPriceConsumer*** to consume StockPrices and display batch lengths for ***poll***()
- Shows using poll(), and basic configuration (review)
- Also Shows how to use a Kafka Deserializer
- How to configure the ***Deserializer*** in Consumer config
- All the other examples build on this and this builds on Advanced Producer StockPrice example

Consumer: createConsumer / Consumer Config

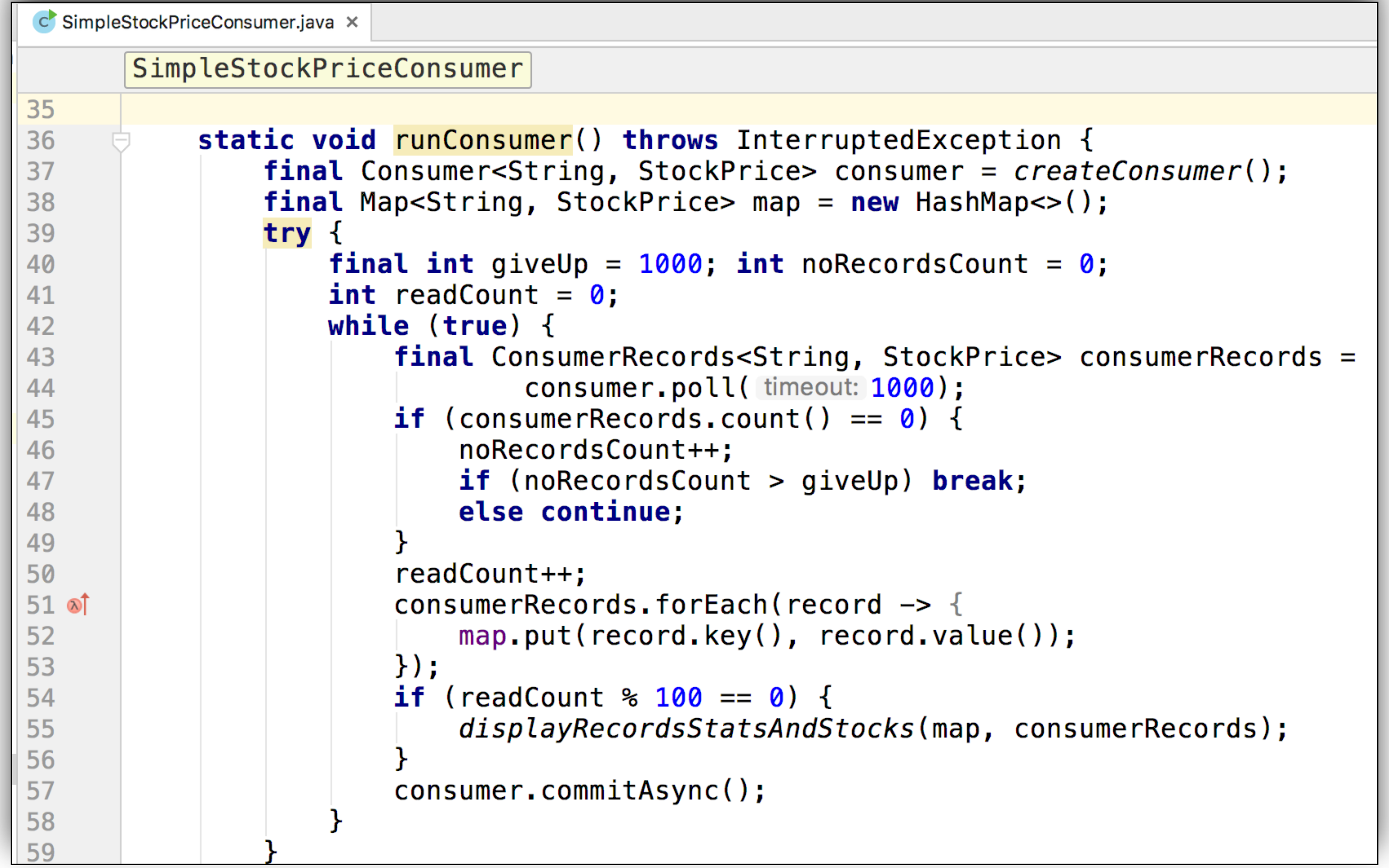

SimpleStockPriceConsumer.java

SimpleStockPriceConsumer

```java
public class SimpleStockPriceConsumer {

    private static Consumer<String, StockPrice> createConsumer() {
        final Properties props = new Properties();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
                StockAppConstants.BOOTSTRAP_SERVERS);
        props.put(ConsumerConfig.GROUP_ID_CONFIG,
                "KafkaExampleConsumer");
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
                StringDeserializer.class.getName());
        //Custom Deserializer
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
                StockDeserializer.class.getName());
        props.put(ConsumerConfig.MAX_POLL_RECORDS_CONFIG, 500);
        // Create the consumer using props.
        final Consumer<String, StockPrice> consumer =
                new KafkaConsumer<>(props);
        // Subscribe to the topic.
        consumer.subscribe(Collections.singletonList(
                StockAppConstants.TOPIC));
        return consumer;
    }
```

- Similar to other Consumer examples so far
- Subscribes to ***stock-prices*** topic
- Has custom serializer

SimpleStockPriceConsumer.runConsumer



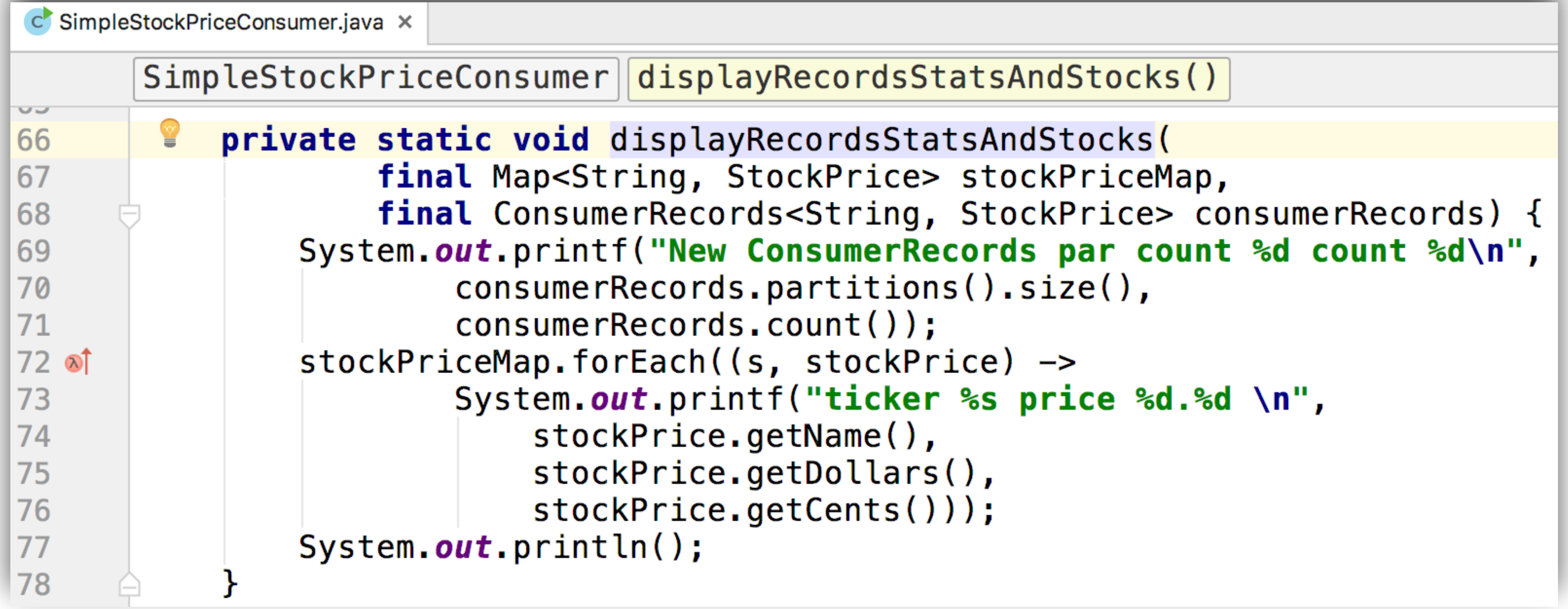
SimpleStockPriceConsumer.java ×

SimpleStockPriceConsumer

```java

    static void runConsumer() throws InterruptedException {
        final Consumer<String, StockPrice> consumer = createConsumer();
        final Map<String, StockPrice> map = new HashMap<>();
        try {
            final int giveUp = 1000; int noRecordsCount = 0;
            int readCount = 0;
            while (true) {
                final ConsumerRecords<String, StockPrice> consumerRecords =
                        consumer.poll( timeout: 1000);
                if (consumerRecords.count() == 0) {
                    noRecordsCount++;
                    if (noRecordsCount > giveUp) break;
                    else continue;
                }
                readCount++;
                consumerRecords.forEach(record -> {
                    map.put(record.key(), record.value());
                });
                if (readCount % 100 == 0) {
                    displayRecordsStatsAndStocks(map, consumerRecords);
                }
                consumer.commitAsync();
            }
        }
```

- Drains topic; Creates map of current stocks; Calls ***displayRecordsStatsAndStocks()***

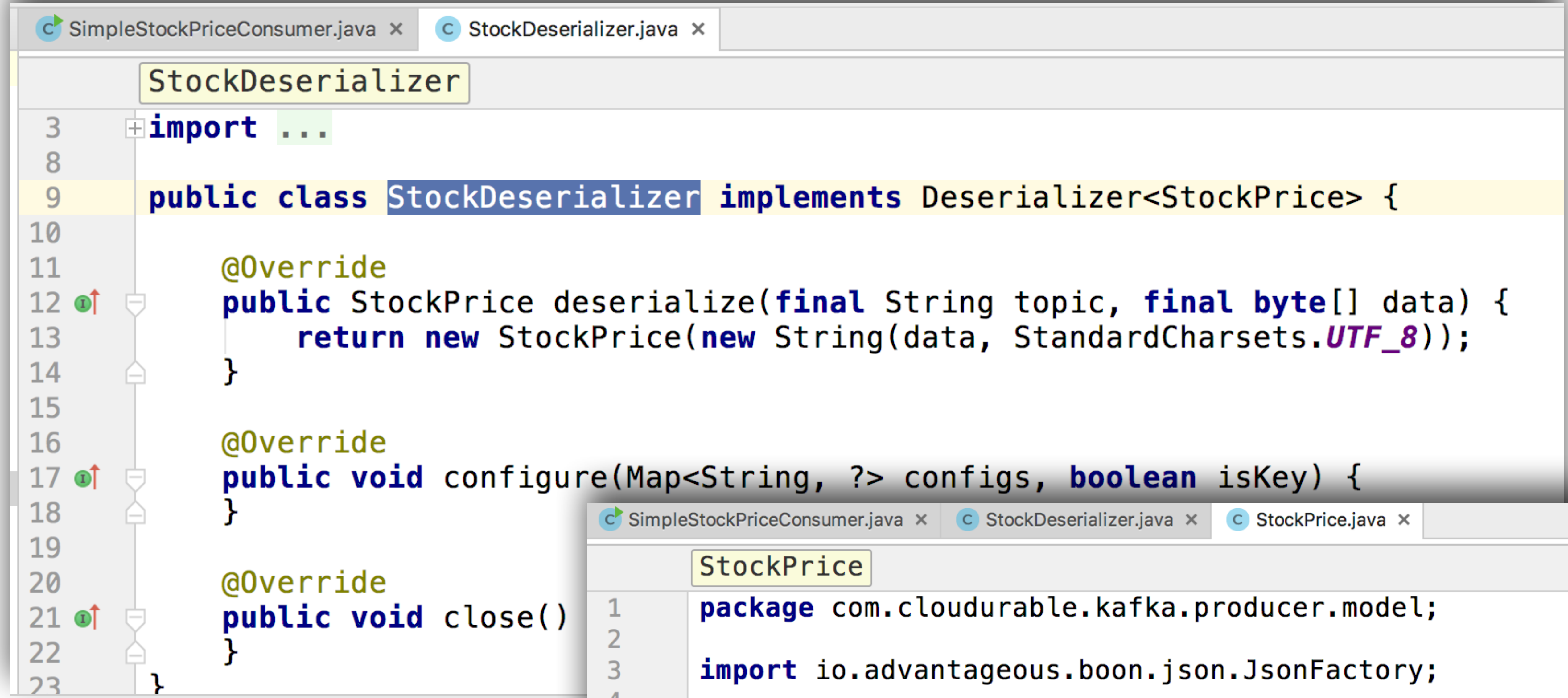
Using ConsumerRecords : SimpleStockPriceConsumer.display



```java
    private static void displayRecordsStatsAndStocks(
            final Map<String, StockPrice> stockPriceMap,
            final ConsumerRecords<String, StockPrice> consumerRecords) {
        System.out.printf("New ConsumerRecords par count %d count %d\n",
                consumerRecords.partitions().size(),
                consumerRecords.count());
        stockPriceMap.forEach((s, stockPrice) ->
                System.out.printf("ticker %s price %d.%d \n",
                    stockPrice.getName(),
                    stockPrice.getDollars(),
                    stockPrice.getCents()));
        System.out.println();
    }
```

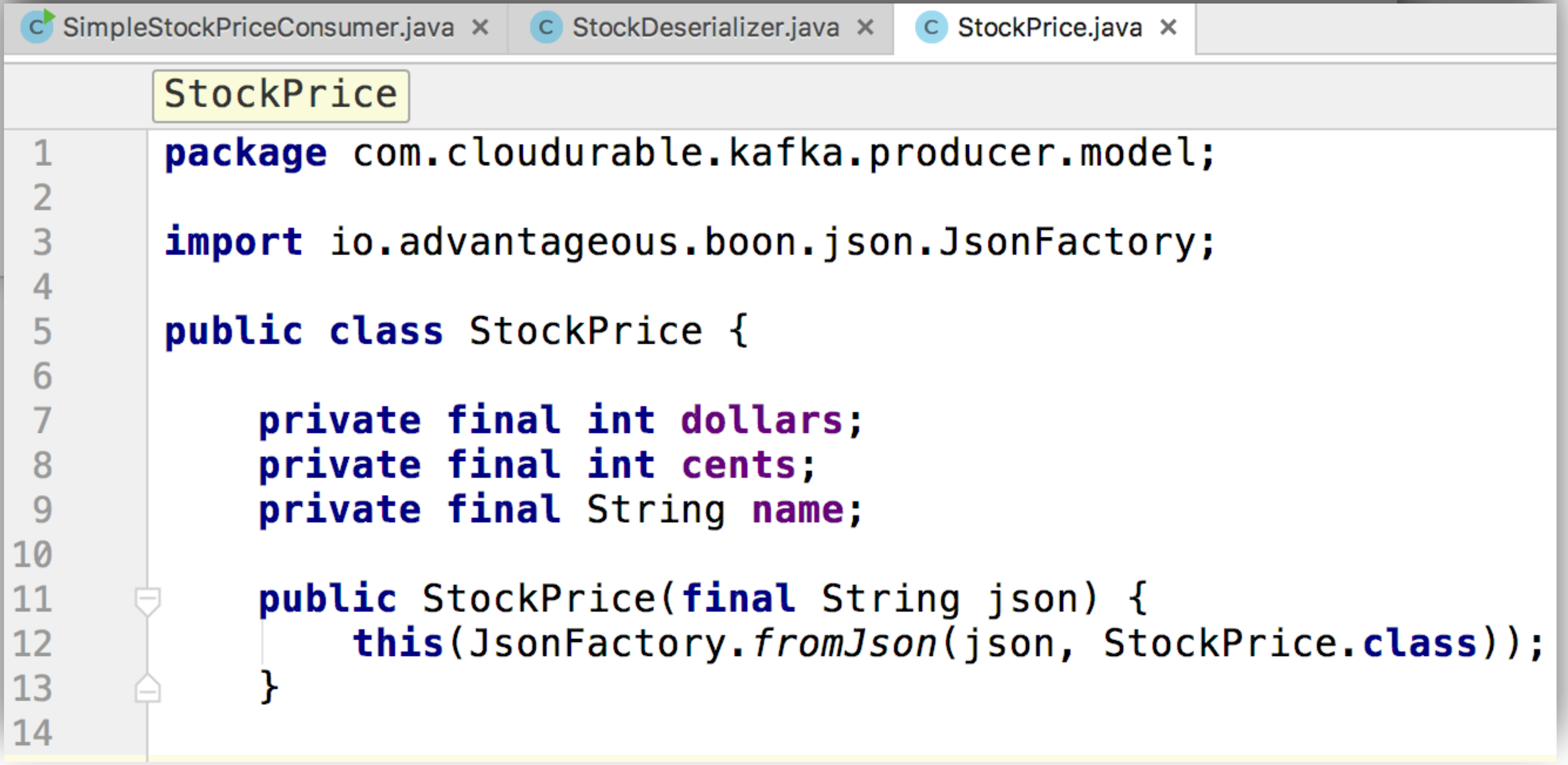
- Prints out size of each partition read and total record count
- Prints out each stock at its current price

Consumer Deserializer: StockDeserializer


SimpleStockPriceConsumer.java × StockDeserializer.java ×

StockDeserializer

```java
import ...

public class StockDeserializer implements Deserializer<StockPrice> {

    @Override
    public StockPrice deserialize(final String topic, final byte[] data) {
        return new StockPrice(new String(data, StandardCharsets.UTF_8));
    }

    @Override
    public void configure(Map<String, ?> configs, boolean isKey) {
    }

    @Override
    public void close()
    }
}
```


SimpleStockPriceConsumer.java × StockDeserializer.java × StockPrice.java ×

StockPrice

```java
package com.cloudurable.kafka.producer.model;

import io.advantageous.boon.json.JsonFactory;

public class StockPrice {

    private final int dollars;
    private final int cents;
    private final String name;

    public StockPrice(final String json) {
        this(JsonFactory.fromJson(json, StockPrice.class));
    }

```

KafkaConsumer: Offsets and Consumer Position

- Consumer position is offset and partition of last record per partition consuming from
  - offset for each record in a partition as a unique identifier record location in partition
- Consumer position gives offset of next record that it consume (next highest)
  - position advances automatically for each call to ***poll***(..)
- Consumer committed position is last offset that has been stored to broker
  - If consumer fails, it picks up at last committed position
- Consumer can auto commit offsets (***enable.auto.commit***) periodically (***auto.commit.interval.ms***) or do commit explicitly using ***commitSync()*** and ***commitAsync()***

KafkaConsumer: Consumer Groups

- Consumers organized into consumer groups (Consumer instances with same ***group.id***)
  - Pool of consumers divide work of consuming and processing records
  - Processes or threads running on same box or distributed for scalability/fault tolerance
- Kafka shares topic partitions among all consumers in a consumer group
  - each partition is assigned to exactly one consumer in consumer group
  - Example topic has six partitions, and a consumer group has two consumer processes, each process gets consume three partitions
- Failover and Group rebalancing
  - if a consumer fails, Kafka reassigned partitions from failed consumer to to other consumers in same consumer group
  - if new consumer joins, Kafka moves partitions from existing consumers to new one

Consumer Groups and Topic Subscriptions

- Consumer group form ***single logical subscriber*** made up of multiple consumers
- Kafka is a multi-subscriber system, Kafka supports N number of ***consumer groups*** for a given topic without duplicating data
- To get something like a MOM queue all consumers would be in single consumer group
  - Load balancing like a MOM queue
- Unlike a MOM, you can have multiple such consumer groups, and price of subscribers does not incur duplicate data
- To get something like MOM pub-sub each process would have its own consumer group

KafkaConsumer: Partition Reassignment

- Consumer partition reassignment in a consumer group happens automatically
- Consumers are notified via ***ConsumerRebalanceListener***
    - Triggers consumers to finish necessary clean up
- Consumer can use API to assign specific partitions using ***assign***(Collection)
    - disables dynamic partition assignment and consumer group coordination
- Dead client may see CommitFailedException thrown from a call to commitSync()
    - Only active members of consumer group can commit offsets.

Controlling Consumers Position

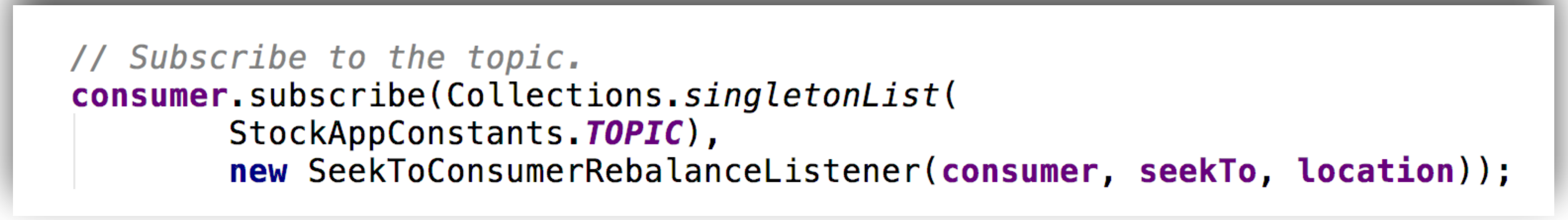
- You can control consumer position
  - moving consumer forward or backwards - consumer can re-consume older records or skip to most recent records
- Use ***consumer.seek***(TopicPartition, long) to specify new position
  - ***consumer.seekToBeginning(Collection)*** and ***seekToEnd(Collection) respectively)***
- Use Case Time-sensitive record processing: Skip to most recent records
- Use Case Bug Fix: Reset position before bug fix and replay log from there
- Use Case Restore State for Restart or Recovery: Consumer initialize position on start-up to whatever is contained in local store and replay missed parts (cache warm-up or replacement in case of failure assumes Kafka retains sufficient history or you are using log compaction)

Storing Offsets Outside: Managing Offsets

- For the consumer o manage its own offset you just need to do the following:
  - Set ***enable.auto.commit=false***
  - Use offset provided with each ConsumerRecord to save your position (partition/offset)
  - On restart restore consumer position using ***kafkaConsumer.seek***(TopicPartition, long).
- Usage like this simplest when the partition assignment is also done manually using ***assign***() instead of ***subscribe***()

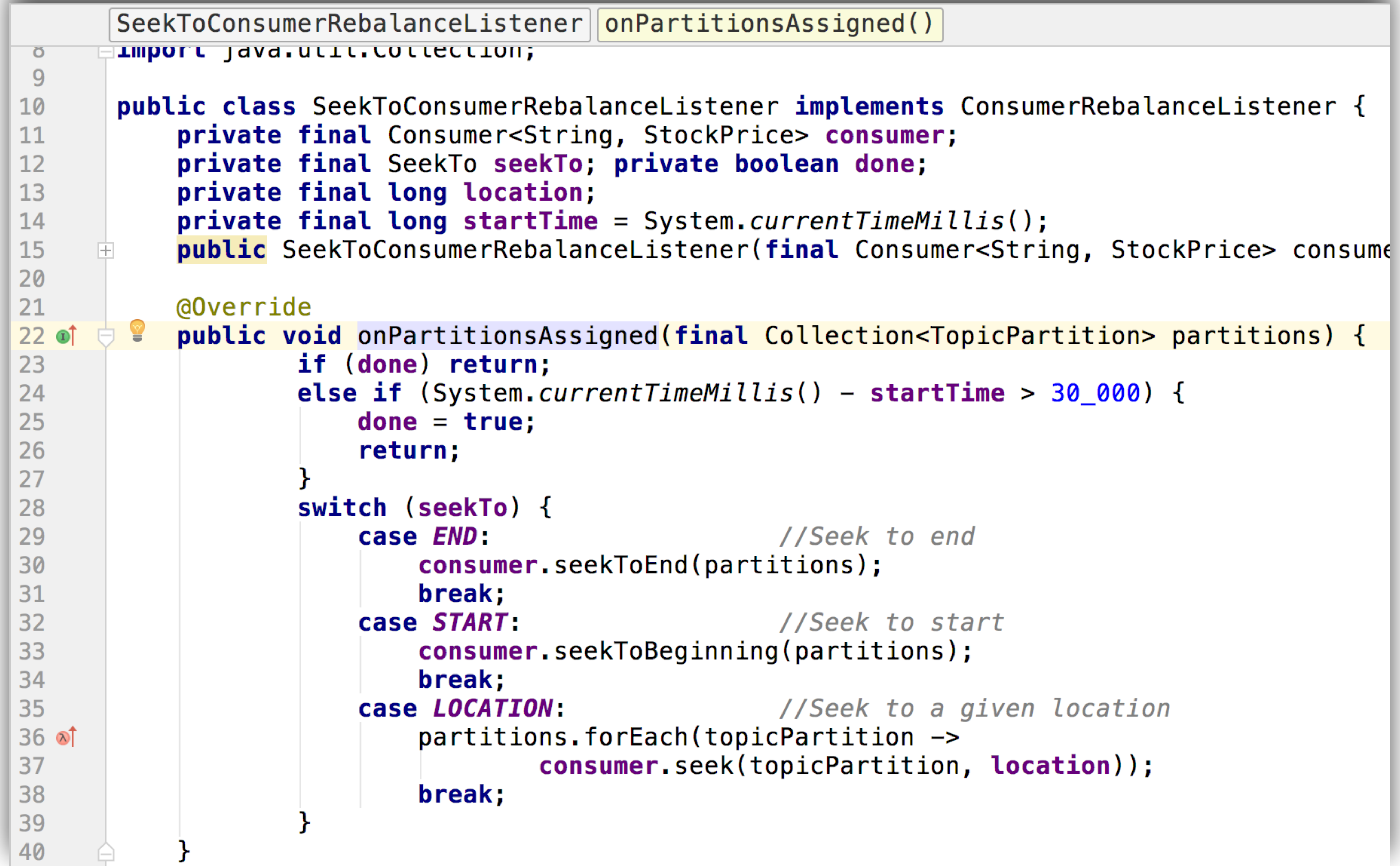
Storing Offsets Outside: Managing Offsets

- If using automatic partition assignment, you must handle cases where partition assignments change
  - Pass *ConsumerRebalanceListener* instance in call to ***kafkaConsumer.subscribe***(Collection, ConsumerRebalanceListener) and ***kafkaConsumer.subscribe***(Pattern, ConsumerRebalanceListener).
  - when partitions taken from consumer, commit its offset for partitions by implementing *ConsumerRebalanceListener.onPartitionsRevoked(Collection)*
  - When partitions are assigned to consumer, look up offset for new partitions and correctly initialize consumer to that position by implementing *ConsumerRebalanceListener.onPartitionsAssigned(Collection)*



```
// Subscribe to the topic.
consumer.subscribe(Collections.singletonList(
        StockAppConstants.TOPIC),
        new SeekToConsumerRebalanceListener(consumer, seekTo, location));
```

Controlling Consumers Position Example



SeekToConsumerRebalanceListener   onPartitionsAssigned()

```java
import java.util.Collection;

public class SeekToConsumerRebalanceListener implements ConsumerRebalanceListener {
    private final Consumer<String, StockPrice> consumer;
    private final SeekTo seekTo; private boolean done;
    private final long location;
    private final long startTime = System.currentTimeMillis();
    public SeekToConsumerRebalanceListener(final Consumer<String, StockPrice> consume

    @Override
    public void onPartitionsAssigned(final Collection<TopicPartition> partitions) {
            if (done) return;
            else if (System.currentTimeMillis() - startTime > 30_000) {
                done = true;
                return;
            }
            switch (seekTo) {
                case END:                       //Seek to end
                    consumer.seekToEnd(partitions);
                    break;
                case START:                     //Seek to start
                    consumer.seekToBeginning(partitions);
                    break;
                case LOCATION:                  //Seek to a given location
                    partitions.forEach(topicPartition ->
                            consumer.seek(topicPartition, location));
                    break;
            }
    }
```

KafkaConsumer: Consumer Alive Detection

- Consumers join consumer group after subscribe and then ***poll***() is called
- Automatically, consumer sends periodic heartbeats to Kafka brokers server
- If consumer crashes or unable to send heartbeats for a duration of ***session.timeout.ms***, then consumer is deemed dead and its partitions are reassigned

KafkaConsumer: Manual Partition Assignment

- Instead of subscribing to the topic using subscribe, you can call ***assign(Collection)*** with the full topic partition list

```
String topic = “log-replication“;

TopicPartition part0 = new TopicPartition(topic, 0);

TopicPartition part1 = new TopicPartition(topic, 1);

consumer.assign(Arrays.asList(part0, part1));
```

- Using consumer as before with ***poll()***
- Manual partition assignment negates use of group coordination, and auto consumer fail over - Each consumer acts independently even if in a consumer group (use unique group id to avoid confusion)
- You have to use ***assign()*** or ***subscribe()*** but not both

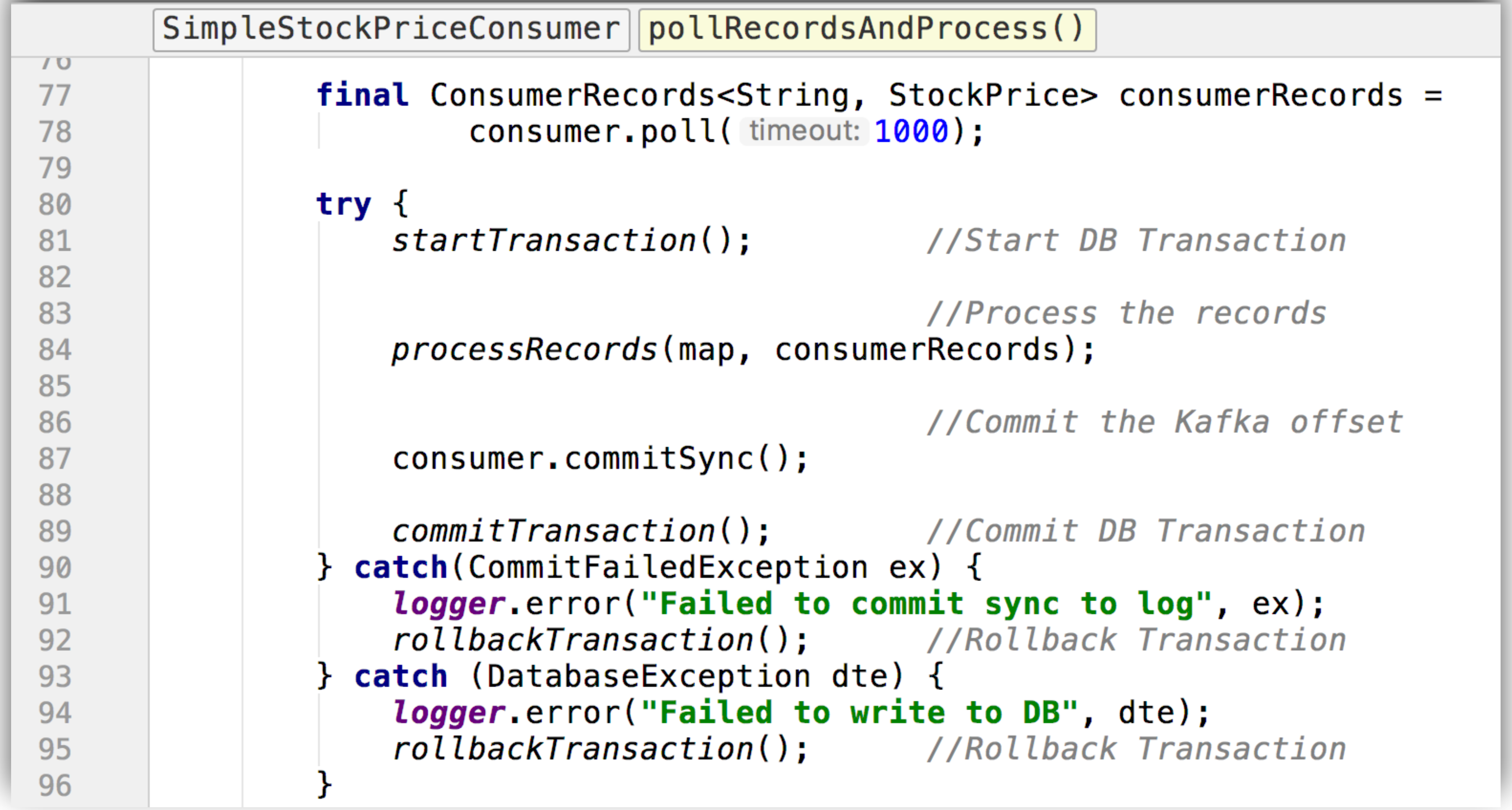
KafkaConsumer: Consumer Alive if Polling

- Calling ***poll()*** marks consumer as alive
  - If consumer continues to call poll(), then consumer is alive and in consumer group and gets messages for partitions assigned (has to call before every ***max.poll.interval.ms interval***)
  - Not calling ***poll()***, even if consumer is sending heartbeats, consumer is still considered dead
- Processing of records from ***poll*** has to be faster than ***max.poll.interval.ms*** interval or your consumer could be marked dead!
- ***max.poll.records*** is used to limit total records returned from a poll call - easier to predict max time to process records on each poll interval

Message Delivery Semantics

- ***At most once***
    - Messages may be lost but are never redelivered
- ***At least once***
    - Messages are never lost but may be redelivered
- ***Exactly once***
    - this is what people actually want, each message is delivered once and only once

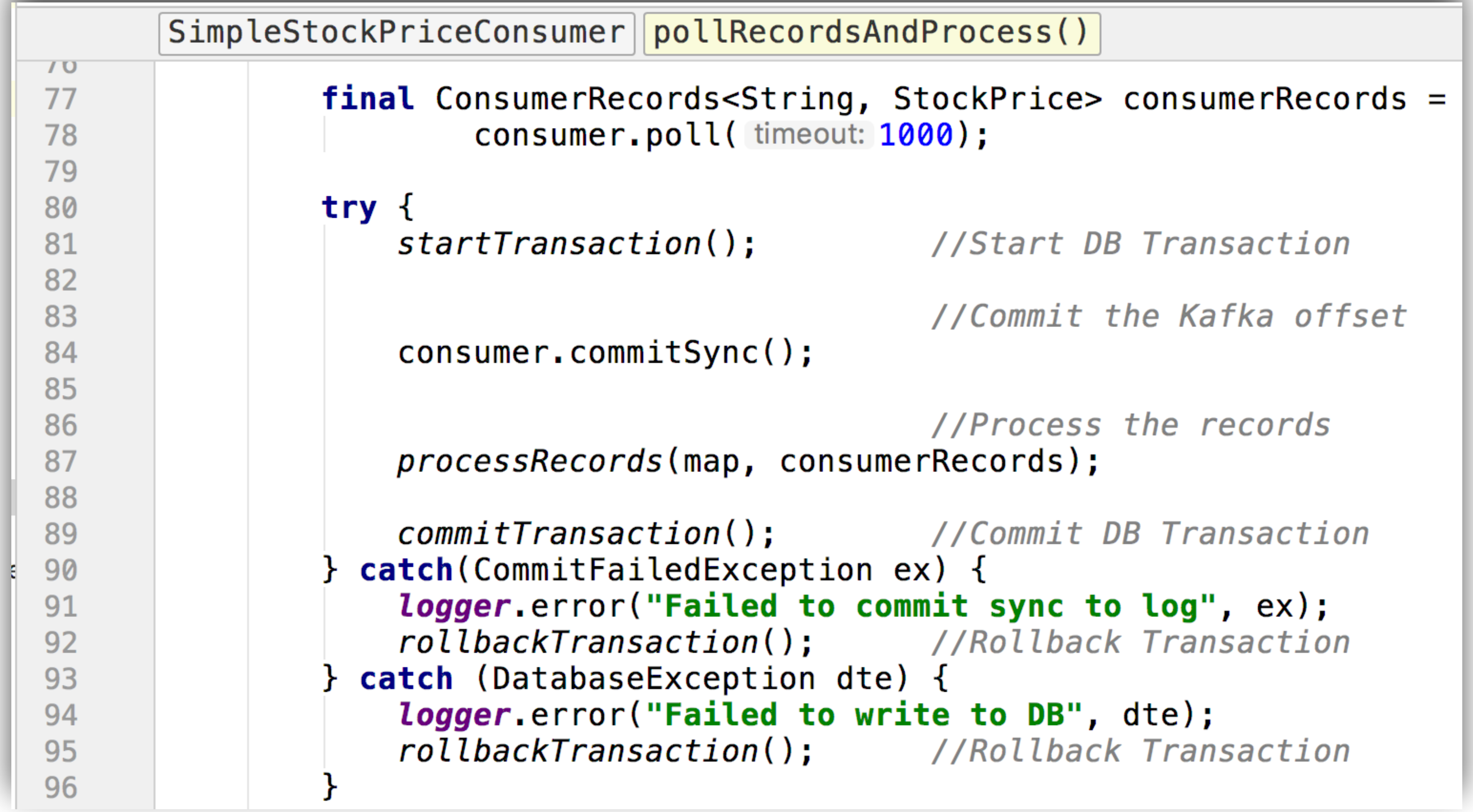
"At-Least-Once" - Delivery Semantics



SimpleStockPriceConsumer pollRecordsAndProcess()

```
        final ConsumerRecords<String, StockPrice> consumerRecords =
                consumer.poll( timeout: 1000);

        try {
            startTransaction();           //Start DB Transaction

                                          //Process the records
            processRecords(map, consumerRecords);

                                          //Commit the Kafka offset
            consumer.commitSync();

            commitTransaction();          //Commit DB Transaction
        } catch(CommitFailedException ex) {
            logger.error("Failed to commit sync to log", ex);
            rollbackTransaction();        //Rollback Transaction
        } catch (DatabaseException dte) {
            logger.error("Failed to write to DB", dte);
            rollbackTransaction();        //Rollback Transaction
        }
```

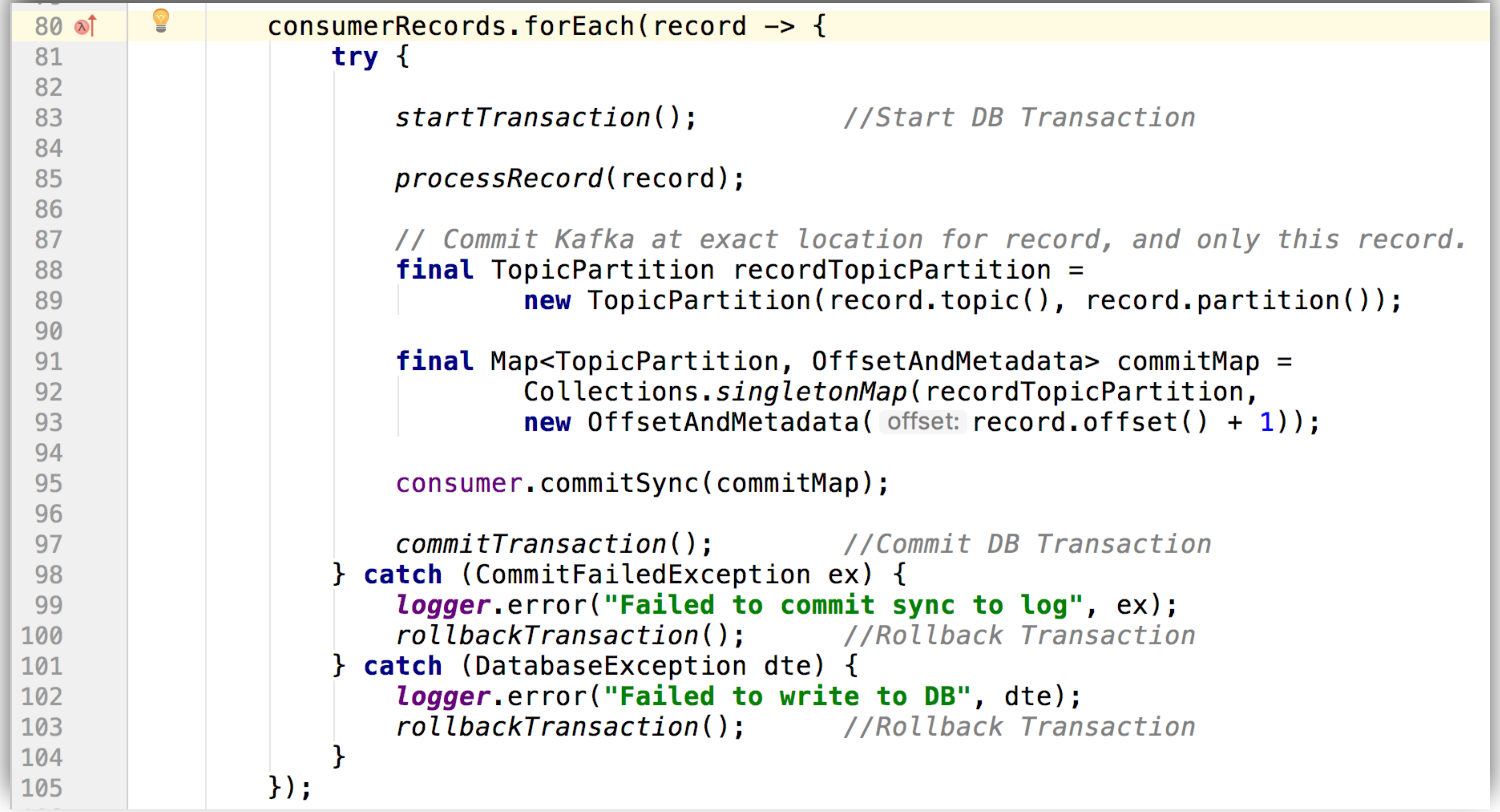
"At-Most-Once" - Delivery Semantics



SimpleStockPriceConsumer | pollRecordsAndProcess()

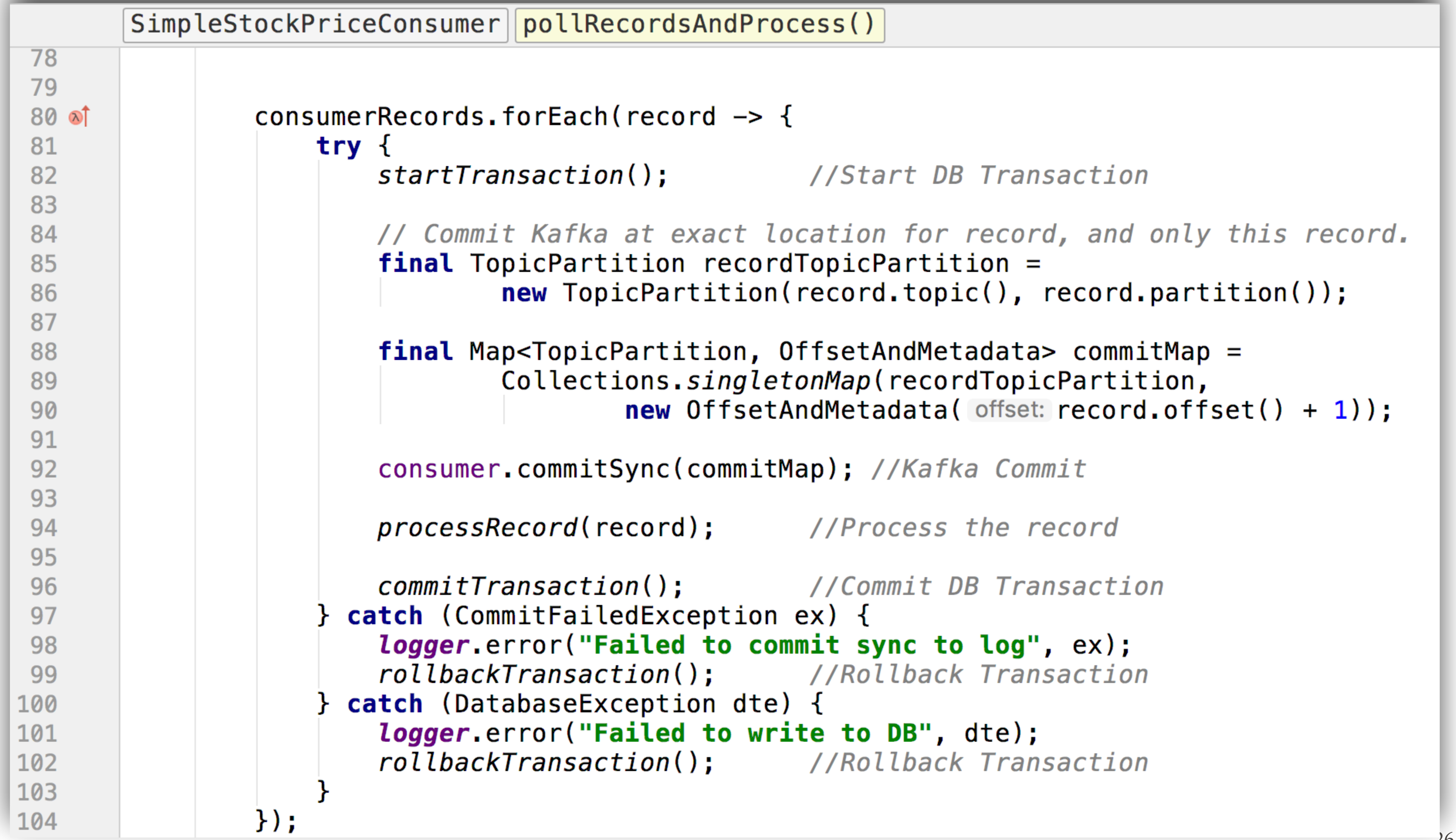
```
        final ConsumerRecords<String, StockPrice> consumerRecords =
                consumer.poll( timeout: 1000);

        try {
            startTransaction();          //Start DB Transaction

                                         //Commit the Kafka offset
            consumer.commitSync();

                                         //Process the records
            processRecords(map, consumerRecords);

            commitTransaction();         //Commit DB Transaction
        } catch(CommitFailedException ex) {
            logger.error("Failed to commit sync to log", ex);
            rollbackTransaction();       //Rollback Transaction
        } catch (DatabaseException dte) {
            logger.error("Failed to write to DB", dte);
            rollbackTransaction();       //Rollback Transaction
        }
```

Fine Grained "At-Most-Once"



```
    consumerRecords.forEach(record -> {
        try {

            startTransaction();         //Start DB Transaction

            processRecord(record);

            // Commit Kafka at exact location for record, and only this record.
            final TopicPartition recordTopicPartition =
                    new TopicPartition(record.topic(), record.partition());

            final Map<TopicPartition, OffsetAndMetadata> commitMap =
                    Collections.singletonMap(recordTopicPartition,
                    new OffsetAndMetadata( offset: record.offset() + 1));

            consumer.commitSync(commitMap);

            commitTransaction();        //Commit DB Transaction
        } catch (CommitFailedException ex) {
            logger.error("Failed to commit sync to log", ex);
            rollbackTransaction();      //Rollback Transaction
        } catch (DatabaseException dte) {
            logger.error("Failed to write to DB", dte);
            rollbackTransaction();      //Rollback Transaction
        }
    });
```

Fine Grained “At-Least-Once”

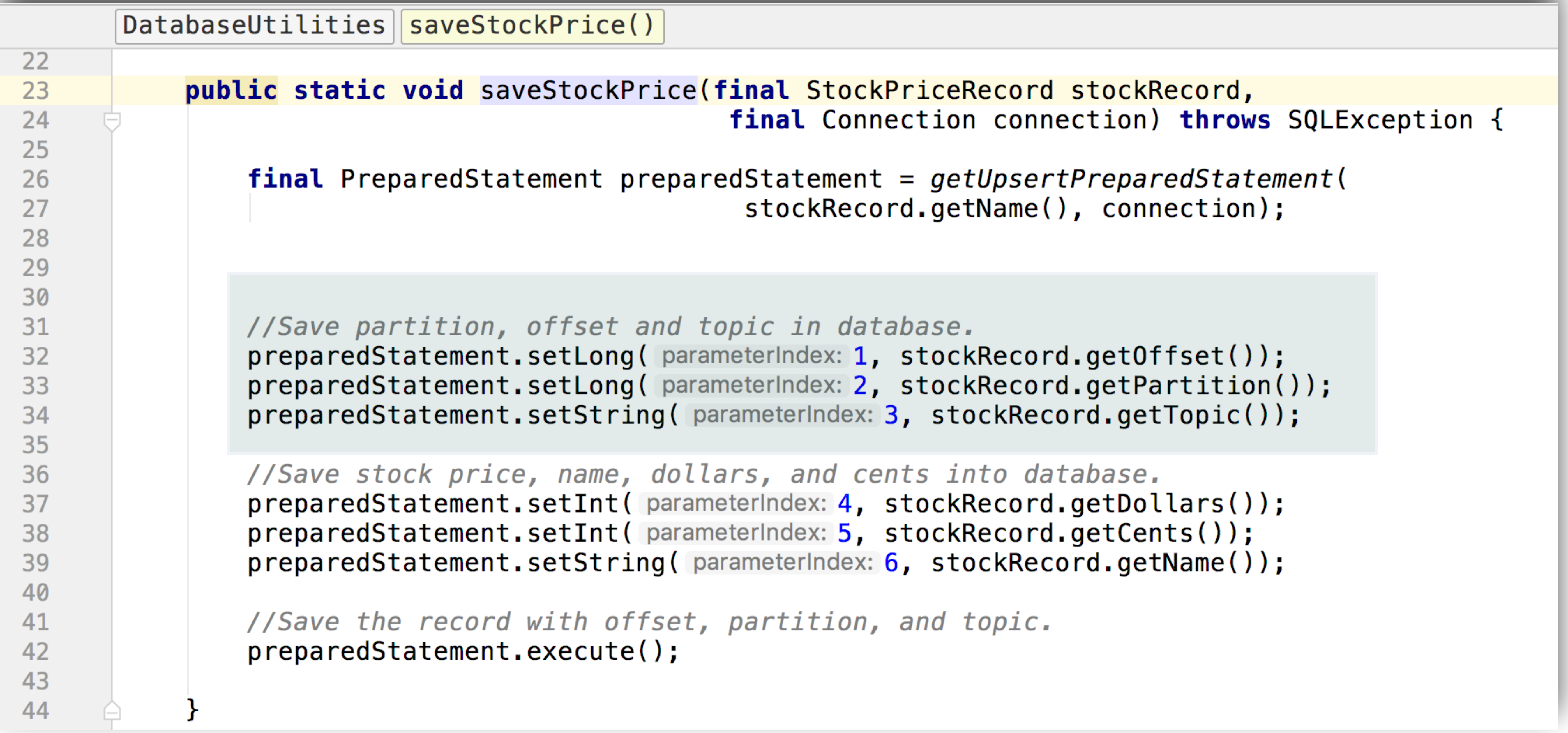

SimpleStockPriceConsumer | pollRecordsAndProcess()

```java


consumerRecords.forEach(record -> {
    try {
        startTransaction();         //Start DB Transaction

        // Commit Kafka at exact location for record, and only this record.
        final TopicPartition recordTopicPartition =
                new TopicPartition(record.topic(), record.partition());

        final Map<TopicPartition, OffsetAndMetadata> commitMap =
                Collections.singletonMap(recordTopicPartition,
                        new OffsetAndMetadata( offset: record.offset() + 1));

        consumer.commitSync(commitMap); //Kafka Commit

        processRecord(record);      //Process the record

        commitTransaction();        //Commit DB Transaction
    } catch (CommitFailedException ex) {
        logger.error("Failed to commit sync to log", ex);
        rollbackTransaction();      //Rollback Transaction
    } catch (DatabaseException dte) {
        logger.error("Failed to write to DB", dte);
        rollbackTransaction();      //Rollback Transaction
    }
});
```

Consumer: Exactly Once, Saving Offset

- Consumer do not have to use Kafka's built-in offset storage
- Consumers can choose to store offsets with processed record output to make it "exactly once" message consumption
- If Consumer output of record consumption is stored in RDBMS then storing offset in database allows committing both process record output and location (partition/offset of record) in a single transaction implementing "exactly once" messaging.
- Typically to achieve "exactly once" you store record location with output of record

Saving Topic, Offset, Partition in DB

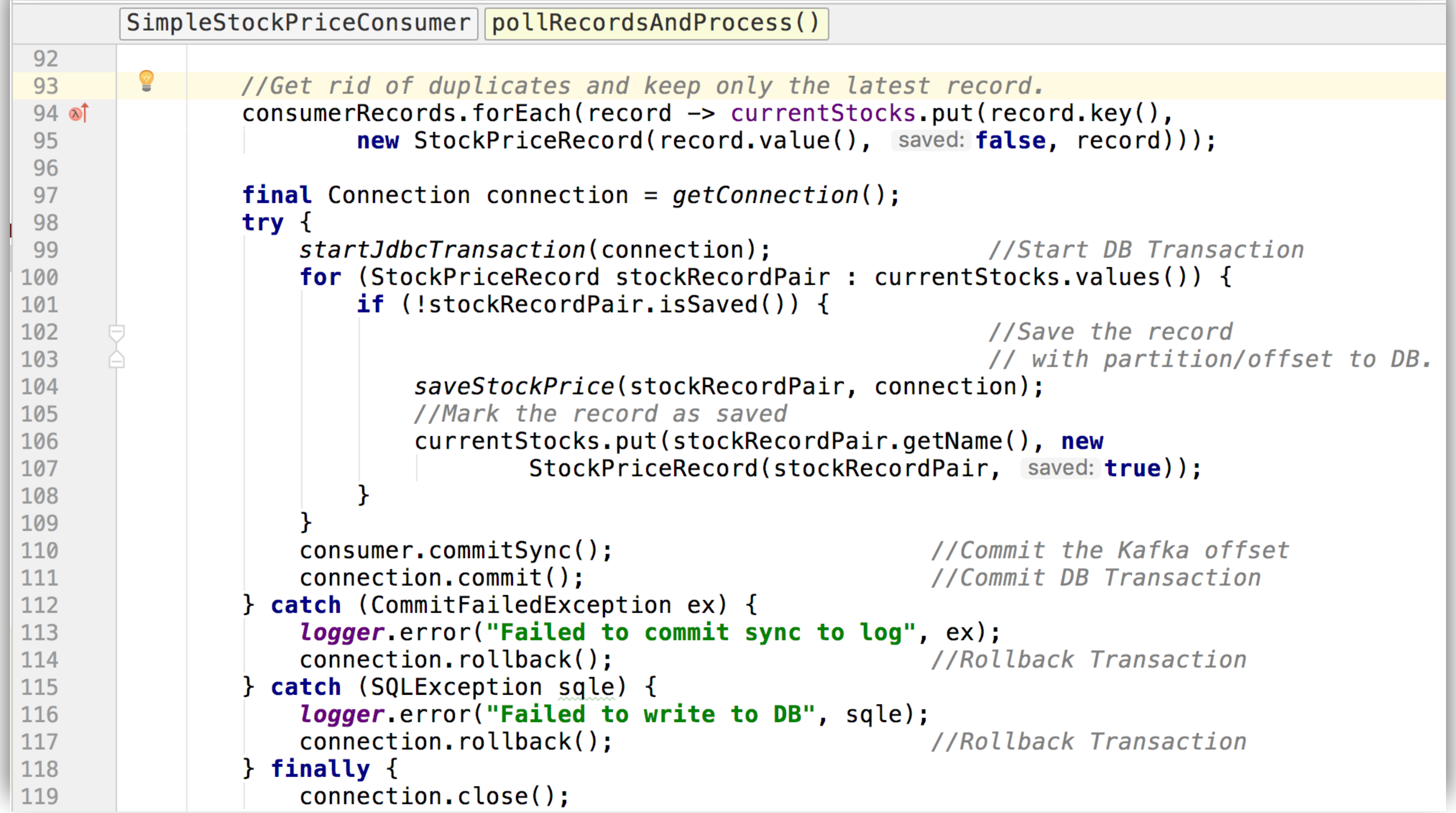


DatabaseUtilities saveStockPrice()

```
    public static void saveStockPrice(final StockPriceRecord stockRecord,
                                      final Connection connection) throws SQLException {

        final PreparedStatement preparedStatement = getUpsertPreparedStatement(
                                        stockRecord.getName(), connection);


        //Save partition, offset and topic in database.
        preparedStatement.setLong( parameterIndex: 1, stockRecord.getOffset());
        preparedStatement.setLong( parameterIndex: 2, stockRecord.getPartition());
        preparedStatement.setString( parameterIndex: 3, stockRecord.getTopic());

        //Save stock price, name, dollars, and cents into database.
        preparedStatement.setInt( parameterIndex: 4, stockRecord.getDollars());
        preparedStatement.setInt( parameterIndex: 5, stockRecord.getCents());
        preparedStatement.setString( parameterIndex: 6, stockRecord.getName());

        //Save the record with offset, partition, and topic.
        preparedStatement.execute();

    }
```

To get exactly once, you need to safe the offset and partition with the output of the consumer process.

"Exactly-Once" - Delivery Semantics



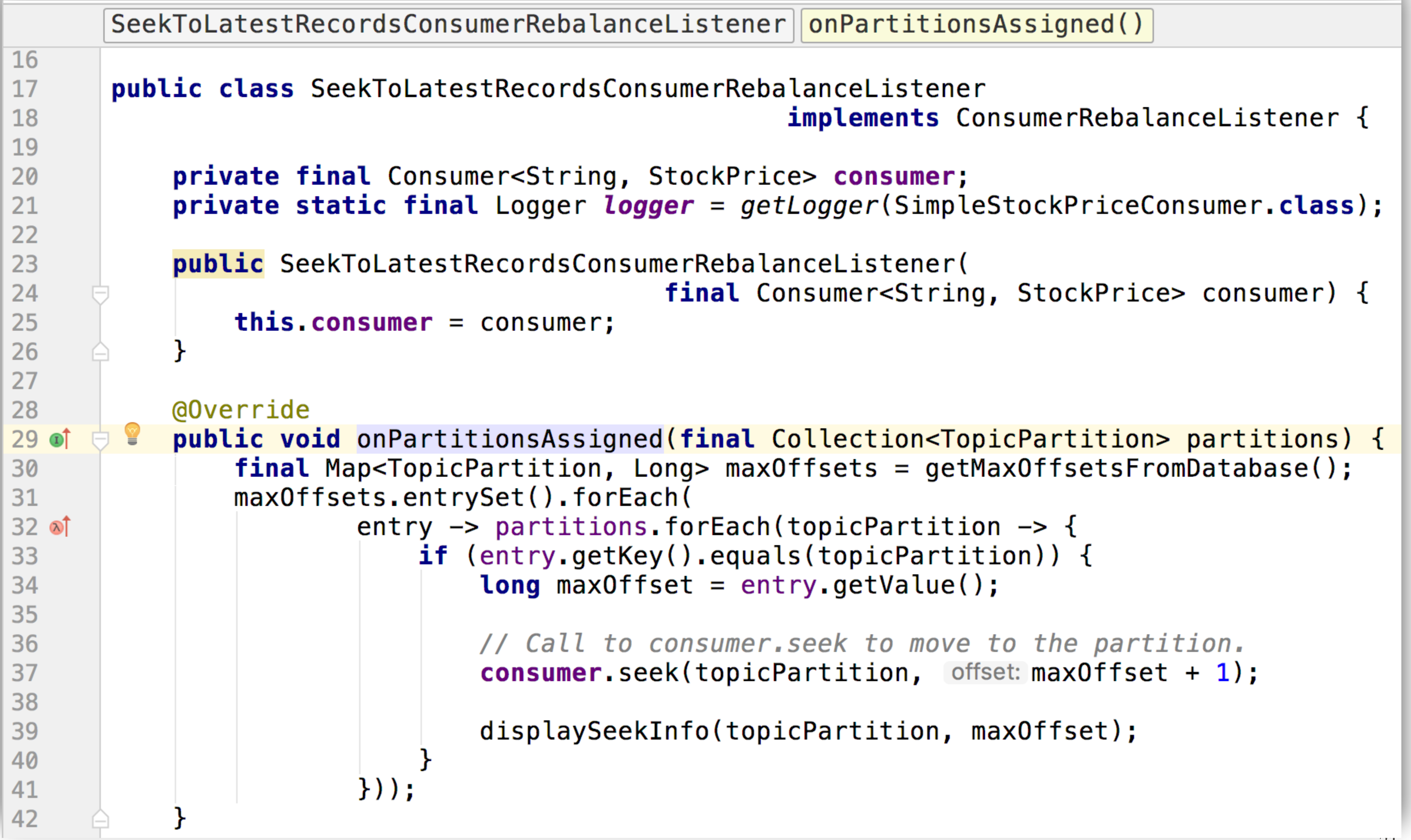
SimpleStockPriceConsumer | pollRecordsAndProcess()

```java

    //Get rid of duplicates and keep only the latest record.
    consumerRecords.forEach(record -> currentStocks.put(record.key(),
            new StockPriceRecord(record.value(),  saved: false, record)));

    final Connection connection = getConnection();
    try {
        startJdbcTransaction(connection);                   //Start DB Transaction
        for (StockPriceRecord stockRecordPair : currentStocks.values()) {
            if (!stockRecordPair.isSaved()) {
                                                            //Save the record
                                                            // with partition/offset to DB.
                saveStockPrice(stockRecordPair, connection);
                //Mark the record as saved
                currentStocks.put(stockRecordPair.getName(), new
                        StockPriceRecord(stockRecordPair,  saved: true));
            }
        }
        consumer.commitSync();                          //Commit the Kafka offset
        connection.commit();                            //Commit DB Transaction
    } catch (CommitFailedException ex) {
        logger.error("Failed to commit sync to log", ex);
        connection.rollback();                          //Rollback Transaction
    } catch (SQLException sqle) {
        logger.error("Failed to write to DB", sqle);
        connection.rollback();                          //Rollback Transaction
    } finally {
        connection.close();
```

Move Offsets past saved Records

- If implementing ***"exactly once"*** message semantics, then you have to manage offset positioning
  - Pass ***ConsumerRebalanceListener*** instance in call to ***kafkaConsumer.subscribe***(Collection, ConsumerRebalanceListener) and ***kafkaConsumer.subscribe***(Pattern, ConsumerRebalanceListener).
  - when partitions taken from consumer, commit its offset for partitions by implementing ***ConsumerRebalanceListener.onPartitionsRevoked(Collection)***
  - When partitions are assigned to consumer, look up offset for new partitions and correctly initialize consumer to that position by implementing ***ConsumerRebalanceListener.onPartitionsAssigned(Collection)***

Exactly Once - Move Offsets past saved Records



SeekToLatestRecordsConsumerRebalanceListener | onPartitionsAssigned()

```java
public class SeekToLatestRecordsConsumerRebalanceListener
                                                    implements ConsumerRebalanceListener {

    private final Consumer<String, StockPrice> consumer;
    private static final Logger logger = getLogger(SimpleStockPriceConsumer.class);

    public SeekToLatestRecordsConsumerRebalanceListener(
                                            final Consumer<String, StockPrice> consumer) {
        this.consumer = consumer;
    }

    @Override
    public void onPartitionsAssigned(final Collection<TopicPartition> partitions) {
        final Map<TopicPartition, Long> maxOffsets = getMaxOffsetsFromDatabase();
        maxOffsets.entrySet().forEach(
                entry -> partitions.forEach(topicPartition -> {
                    if (entry.getKey().equals(topicPartition)) {
                        long maxOffset = entry.getValue();

                        // Call to consumer.seek to move to the partition.
                        consumer.seek(topicPartition, offset: maxOffset + 1);

                        displaySeekInfo(topicPartition, maxOffset);
                    }
                }));
    }
```

KafkaConsumer: Consumption Flow Control

- You can control consumption of topics using by using ***consumer.pause***(Collection) and ***consumer.resume***(Collection)
  - This pauses or resumes consumption on specified assigned partitions for future ***consumer.poll***(long) calls
- Use cases where consumers may want to first focus on fetching from some subset of assigned partitions at full speed, and only start fetching other partitions when these partitions have few or no data to consume
  - Priority queue like behavior from traditional MOM
- Other cases is stream processing if preforming a join and one topic stream is getting behind another.

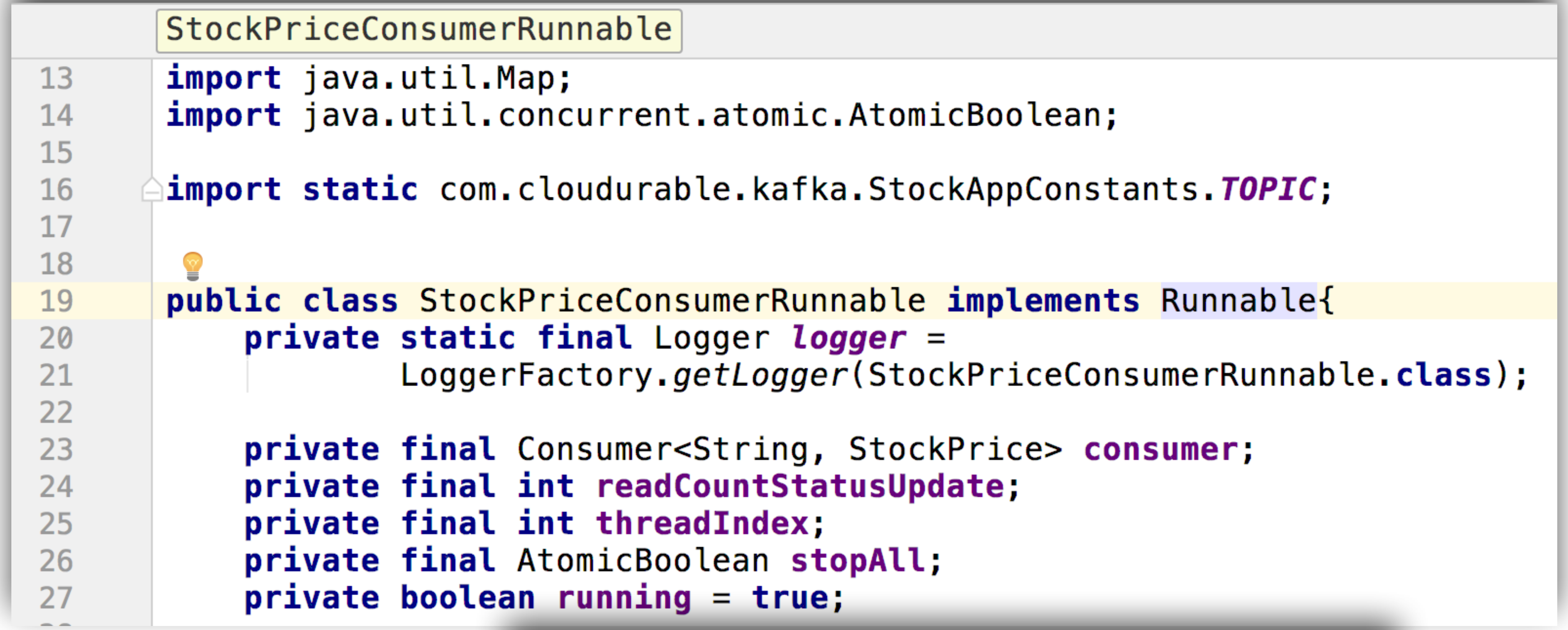
KafkaConsumer: Multi-threaded Processing

- Kafka consumer is NOT thread-safe
- All network I/O happens in thread of the application making call
- Only exception thread safe method is ***consumer.wakeup()***
  - *forces* WakeupException to be thrown from thread blocking on operation
  - Use Case to shutdown consumer from another thread

KafkaConsumer: One Consumer Per Thread

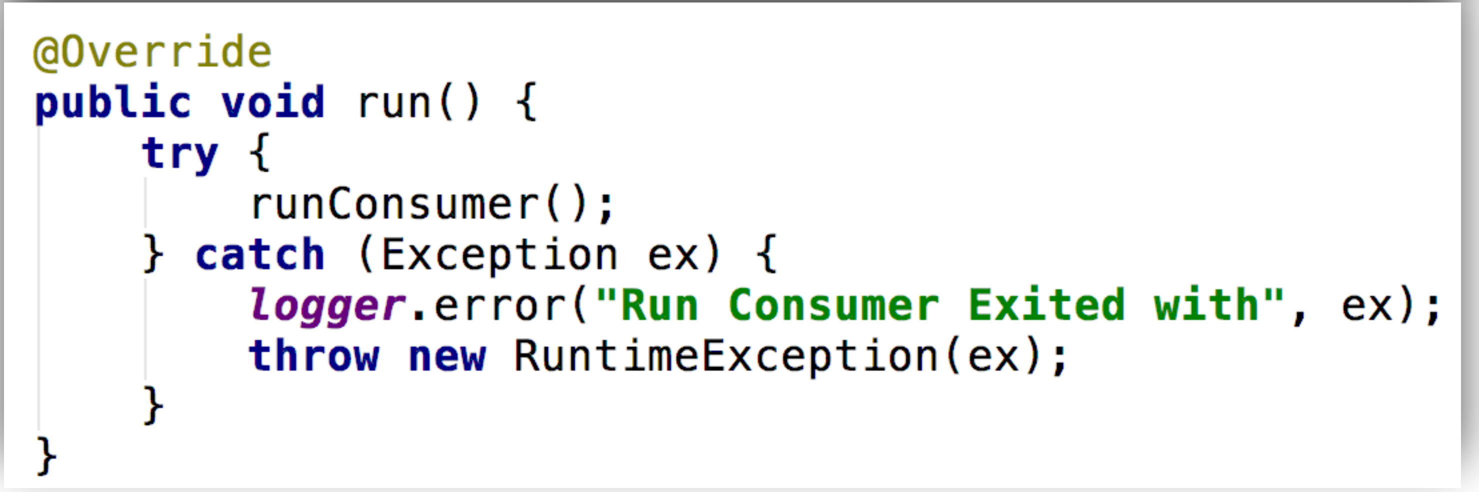
- Pro
  - Easiest to implement
  - Requires no inter-thread co-ordination
  - In-order processing on a per-partition basis easy to implement
    - Process in-order that your receive them
- Con
  - More consumers means more TCP connections to the cluster (one per thread) - low cost has Kafka uses async IO and handles connections efficiently

One Consumer Per Thread: Runnable



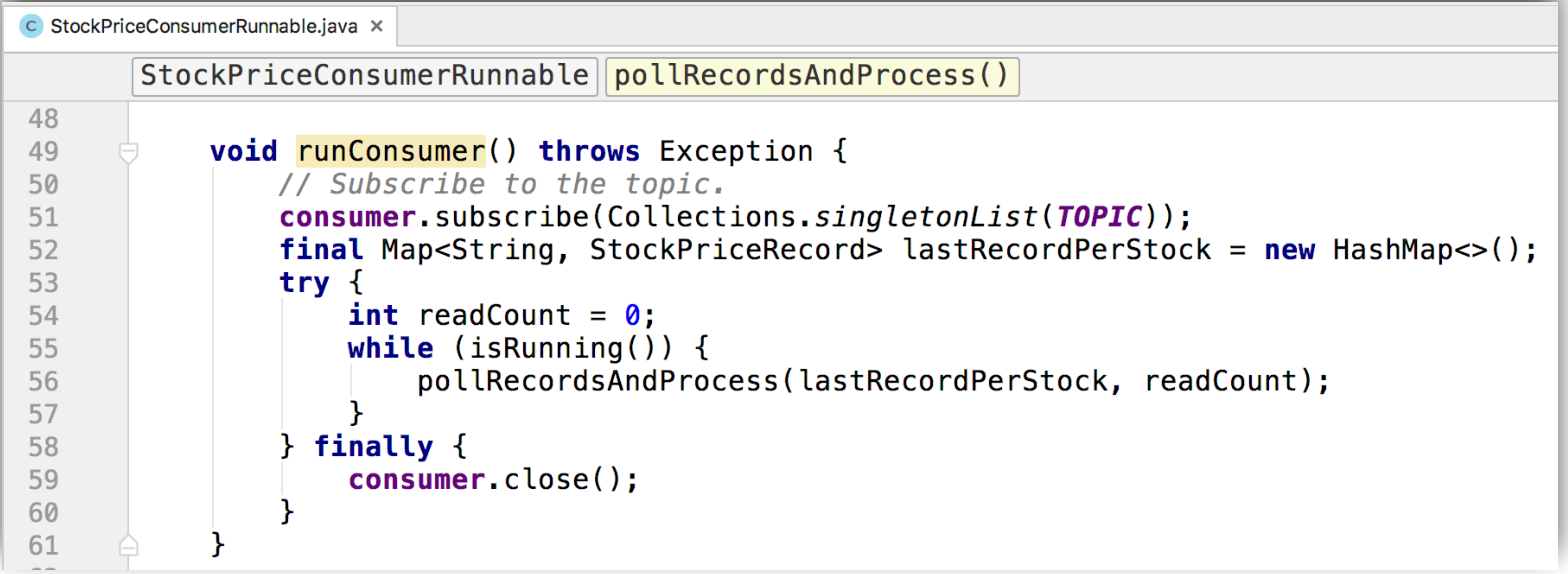
StockPriceConsumerRunnable

```java
import java.util.Map;
import java.util.concurrent.atomic.AtomicBoolean;

import static com.cloudurable.kafka.StockAppConstants.TOPIC;


public class StockPriceConsumerRunnable implements Runnable{
    private static final Logger logger =
            LoggerFactory.getLogger(StockPriceConsumerRunnable.class);

    private final Consumer<String, StockPrice> consumer;
    private final int readCountStatusUpdate;
    private final int threadIndex;
    private final AtomicBoolean stopAll;
    private boolean running = true;
```



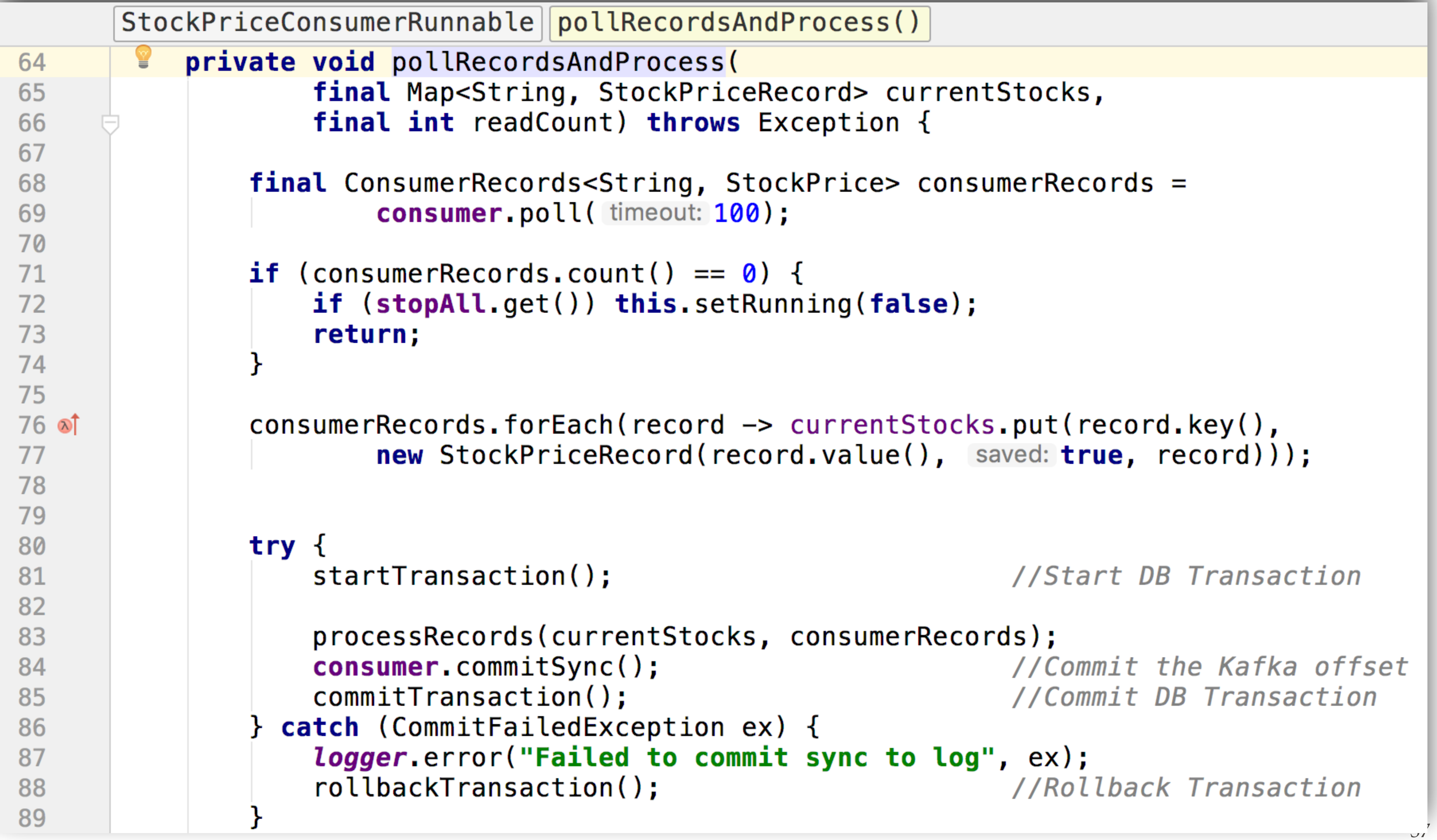
```java
@Override
public void run() {
    try {
        runConsumer();
    } catch (Exception ex) {
        logger.error("Run Consumer Exited with", ex);
        throw new RuntimeException(ex);
    }
}
```

One Consumer Per Thread: runConsumer



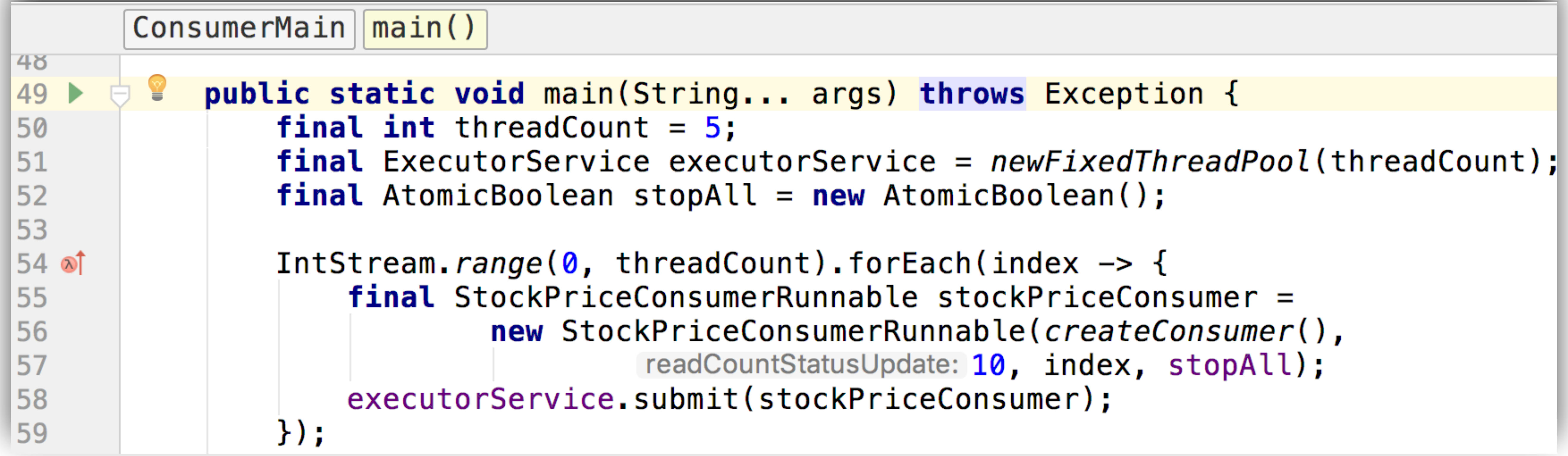
```java

    void runConsumer() throws Exception {
        // Subscribe to the topic.
        consumer.subscribe(Collections.singletonList(TOPIC));
        final Map<String, StockPriceRecord> lastRecordPerStock = new HashMap<>();
        try {
            int readCount = 0;
            while (isRunning()) {
                pollRecordsAndProcess(lastRecordPerStock, readCount);
            }
        } finally {
            consumer.close();
        }
    }
```

One Consumer Per Thread: runConsumer


StockPriceConsumerRunnable | pollRecordsAndProcess()

```
    private void pollRecordsAndProcess(
            final Map<String, StockPriceRecord> currentStocks,
            final int readCount) throws Exception {

        final ConsumerRecords<String, StockPrice> consumerRecords =
                consumer.poll( timeout: 100);

        if (consumerRecords.count() == 0) {
            if (stopAll.get()) this.setRunning(false);
            return;
        }

        consumerRecords.forEach(record -> currentStocks.put(record.key(),
                new StockPriceRecord(record.value(),  saved: true, record)));


        try {
            startTransaction();                                  //Start DB Transaction

            processRecords(currentStocks, consumerRecords);
            consumer.commitSync();                               //Commit the Kafka offset
            commitTransaction();                                 //Commit DB Transaction
        } catch (CommitFailedException ex) {
            logger.error("Failed to commit sync to log", ex);
            rollbackTransaction();                               //Rollback Transaction
        }
```

One Consumer Per Thread: Thread Pool

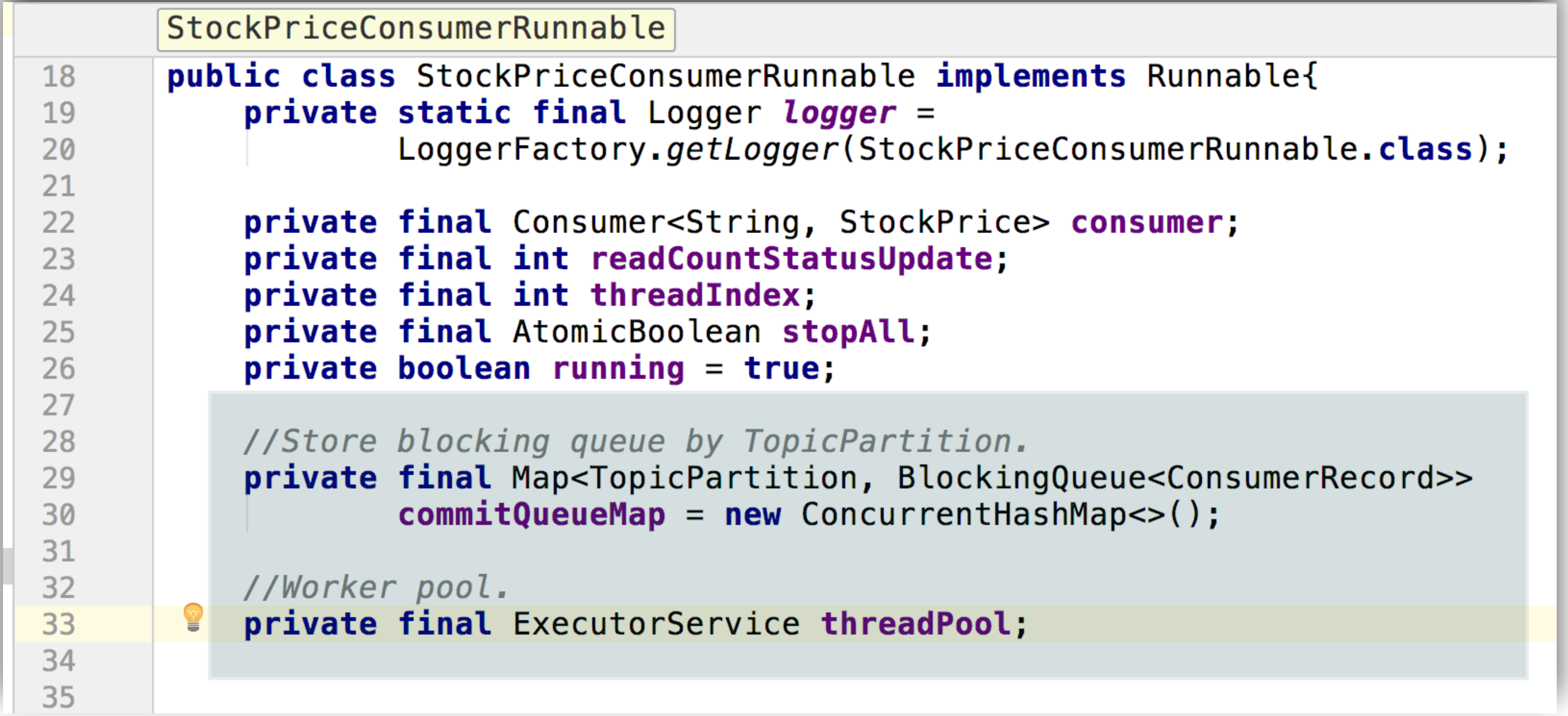

ConsumerMain main()

```
public static void main(String... args) throws Exception {
    final int threadCount = 5;
    final ExecutorService executorService = newFixedThreadPool(threadCount);
    final AtomicBoolean stopAll = new AtomicBoolean();

    IntStream.range(0, threadCount).forEach(index -> {
        final StockPriceConsumerRunnable stockPriceConsumer =
                new StockPriceConsumerRunnable(createConsumer(),
                        readCountStatusUpdate: 10, index, stopAll);
        executorService.submit(stockPriceConsumer);
    });
```

KafkaConsumer: One Consumer with Worker Threads

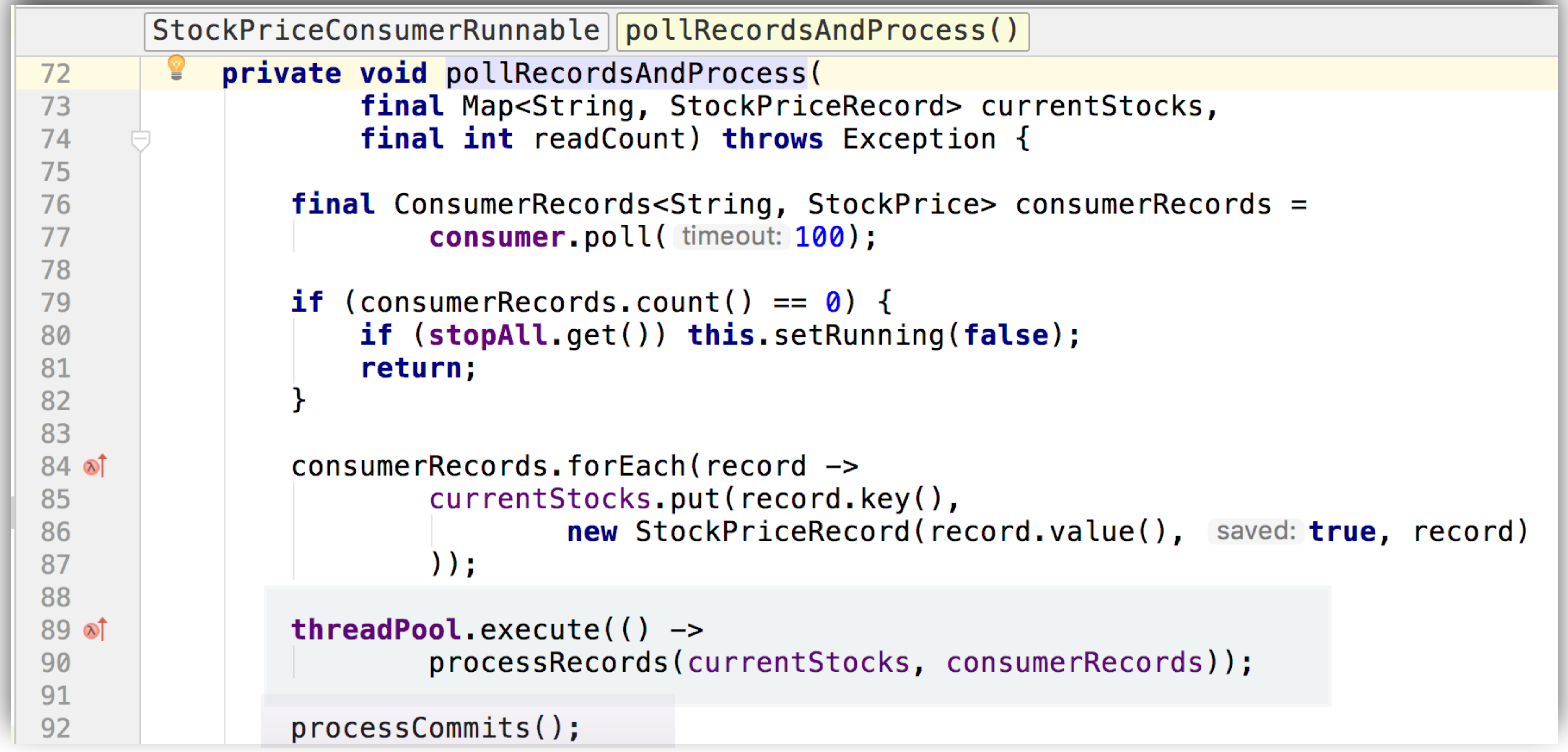
- Decouple Consumption and Processing: One or more consumer threads that consume from Kafka and hands off to ConsumerRecords instances to a blocking queue processed by a processor thread pool that process the records.
- PROs
  - This option allows independently scaling consumers count and processors count. Processor threads are independent of topic partition count
- CONs
  - Guaranteeing order across processors requires care as threads execute independently a later record could be processed before an earlier record and then you have to do consumer commits somehow
  - How do you committing the position unless there is some ordering? You have to provide the ordering. (Concurrently HashMap of BlockingQueues where topic, partition is the key (TopicPartition)?)

One Consumer with Worker Threads


StockPriceConsumerRunnable

```java
public class StockPriceConsumerRunnable implements Runnable{
    private static final Logger logger =
            LoggerFactory.getLogger(StockPriceConsumerRunnable.class);

    private final Consumer<String, StockPrice> consumer;
    private final int readCountStatusUpdate;
    private final int threadIndex;
    private final AtomicBoolean stopAll;
    private boolean running = true;

    //Store blocking queue by TopicPartition.
    private final Map<TopicPartition, BlockingQueue<ConsumerRecord>>
            commitQueueMap = new ConcurrentHashMap<>();

    //Worker pool.
    private final ExecutorService threadPool;

```

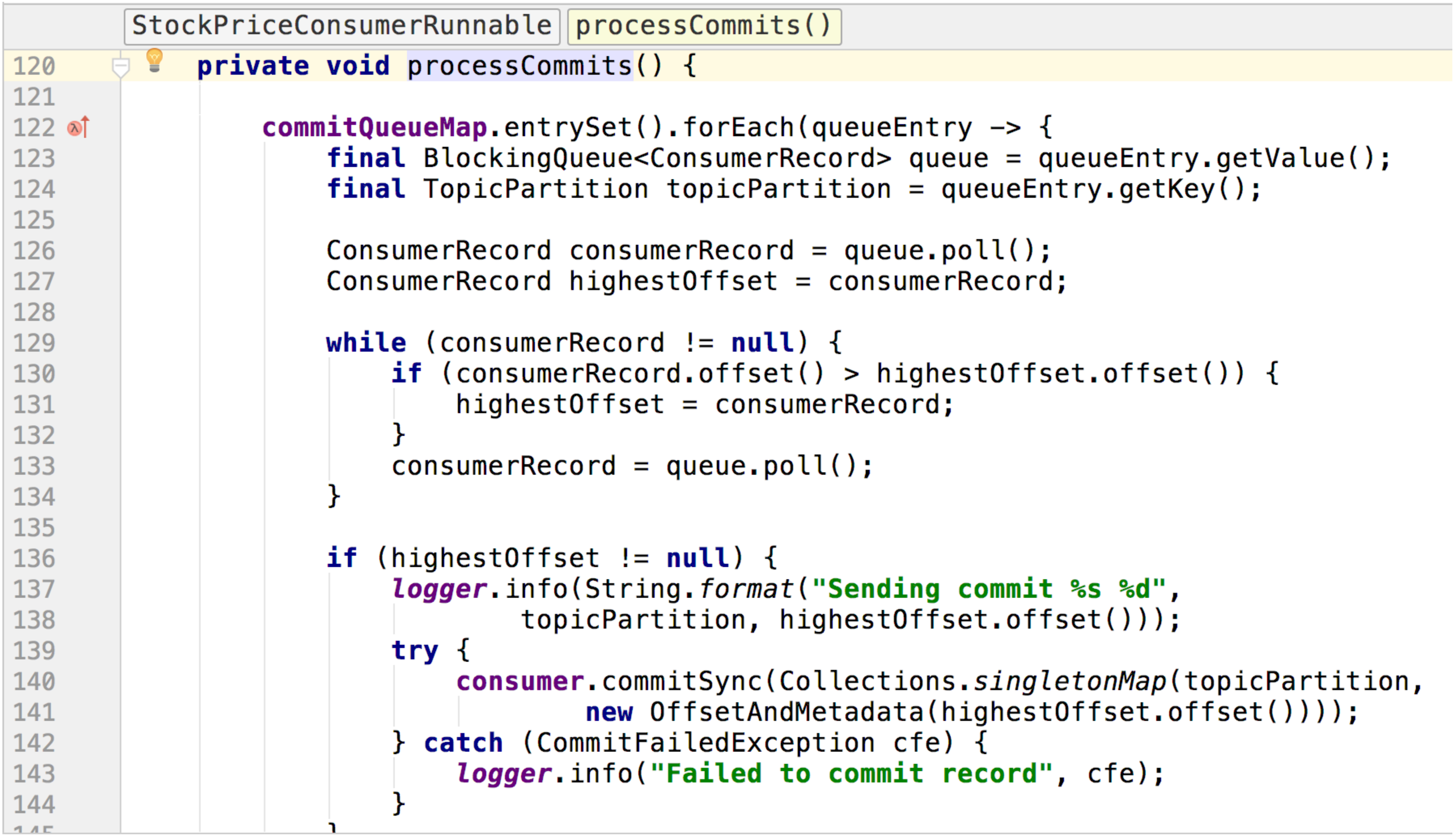
Consumer with Worker Threads: Use Worker



StockPriceConsumerRunnable | pollRecordsAndProcess()

```java
    private void pollRecordsAndProcess(
            final Map<String, StockPriceRecord> currentStocks,
            final int readCount) throws Exception {

        final ConsumerRecords<String, StockPrice> consumerRecords =
                consumer.poll( timeout: 100);

        if (consumerRecords.count() == 0) {
            if (stopAll.get()) this.setRunning(false);
            return;
        }

        consumerRecords.forEach(record ->
                currentStocks.put(record.key(),
                        new StockPriceRecord(record.value(),  saved: true, record)
                ));

        threadPool.execute(() ->
                processRecords(currentStocks, consumerRecords));

        processCommits();
```

Consumer w/ Worker Threads: processCommits

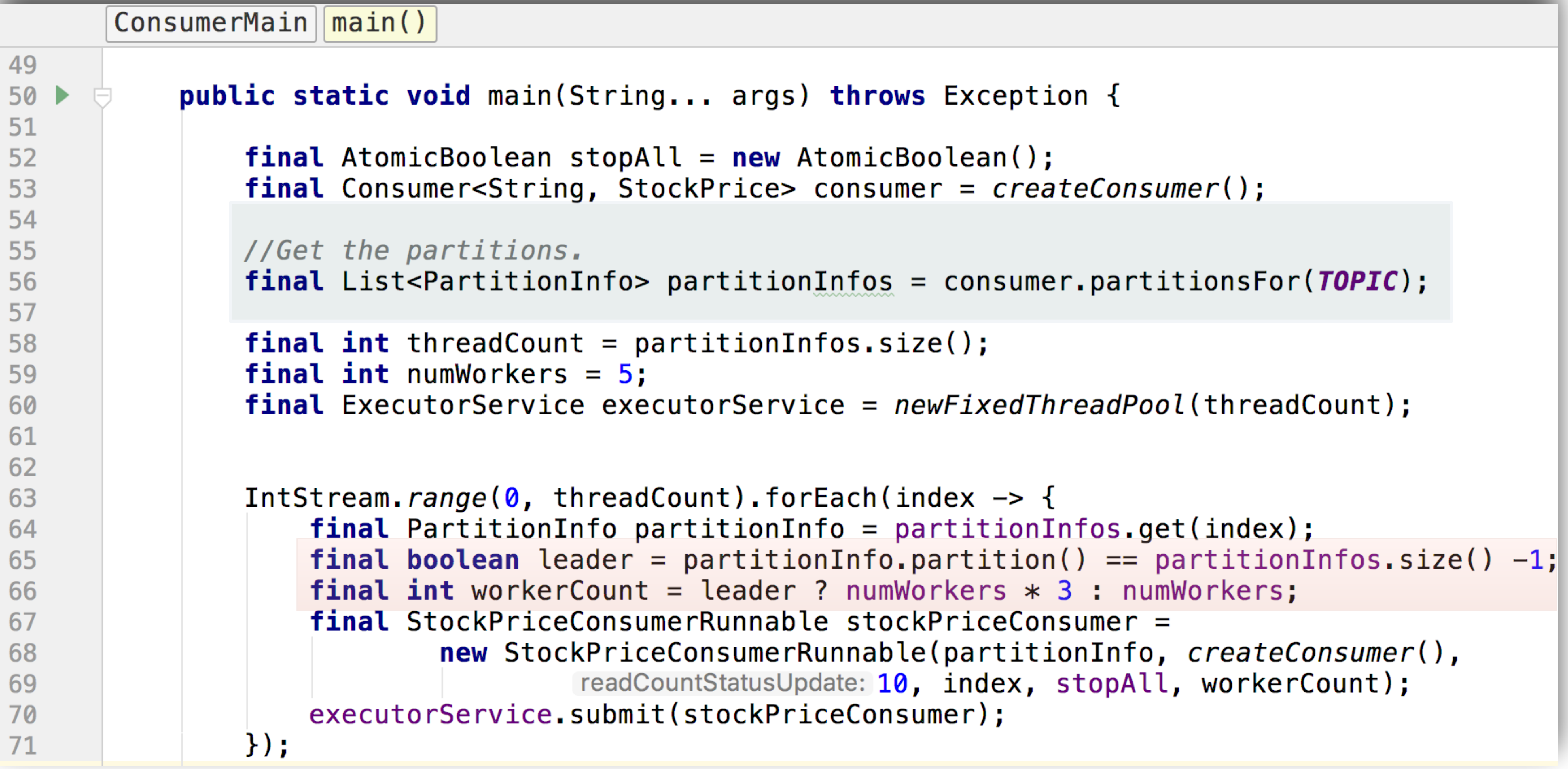


```java
private void processCommits() {

    commitQueueMap.entrySet().forEach(queueEntry -> {
        final BlockingQueue<ConsumerRecord> queue = queueEntry.getValue();
        final TopicPartition topicPartition = queueEntry.getKey();

        ConsumerRecord consumerRecord = queue.poll();
        ConsumerRecord highestOffset = consumerRecord;

        while (consumerRecord != null) {
            if (consumerRecord.offset() > highestOffset.offset()) {
                highestOffset = consumerRecord;
            }
            consumerRecord = queue.poll();
        }

        if (highestOffset != null) {
            logger.info(String.format("Sending commit %s %d",
                    topicPartition, highestOffset.offset()));
            try {
                consumer.commitSync(Collections.singletonMap(topicPartition,
                        new OffsetAndMetadata(highestOffset.offset())));
            } catch (CommitFailedException cfe) {
                logger.info("Failed to commit record", cfe);
            }
        }
```

Using partitionsFor() and assign() for priority queue

- Creating Priority processing queue
- Use **consumer.*partitionsFor(TOPIC)*** to get a list of partitions
- Usage like this simplest when the partition assignment is also done manually using ***assign***() instead of ***subscribe***()
    - Use assign(), pass TopicPartition to worker
- Use Partitioner from earlier example for Producer

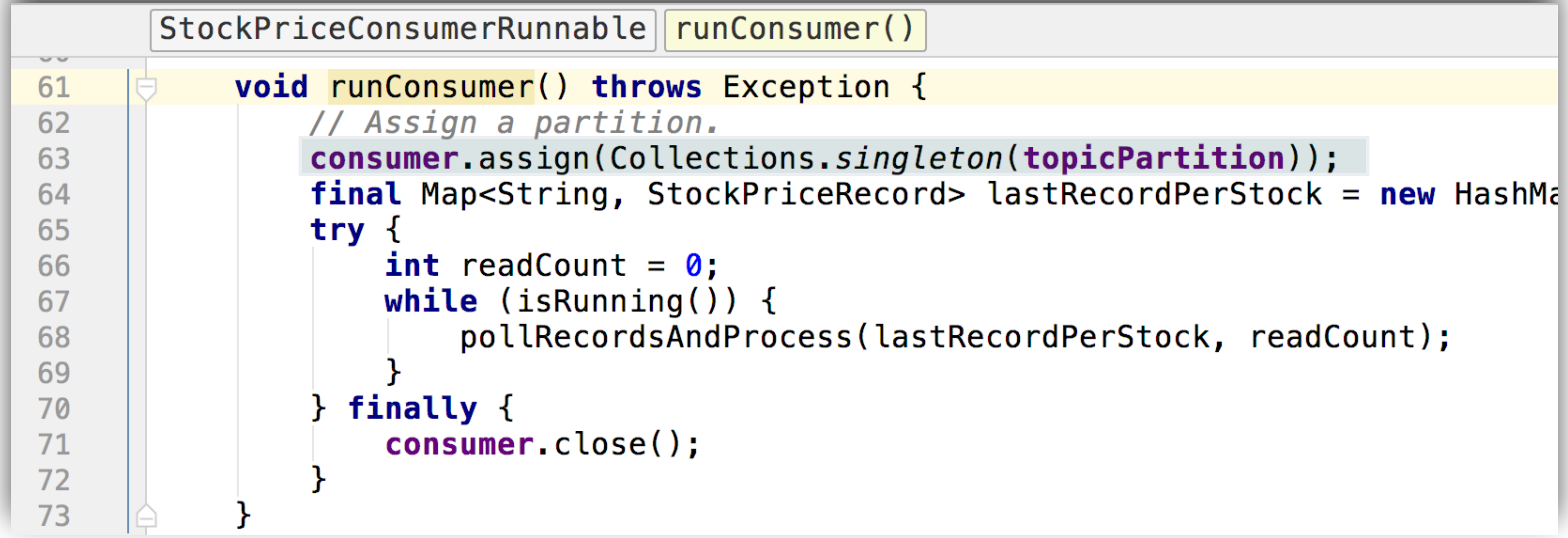
Using partitionsFor() for Priority Queue



ConsumerMain | main()

```java
    public static void main(String... args) throws Exception {

        final AtomicBoolean stopAll = new AtomicBoolean();
        final Consumer<String, StockPrice> consumer = createConsumer();

        //Get the partitions.
        final List<PartitionInfo> partitionInfos = consumer.partitionsFor(TOPIC);

        final int threadCount = partitionInfos.size();
        final int numWorkers = 5;
        final ExecutorService executorService = newFixedThreadPool(threadCount);


        IntStream.range(0, threadCount).forEach(index -> {
            final PartitionInfo partitionInfo = partitionInfos.get(index);
            final boolean leader = partitionInfo.partition() == partitionInfos.size() -1;
            final int workerCount = leader ? numWorkers * 3 : numWorkers;
            final StockPriceConsumerRunnable stockPriceConsumer =
                    new StockPriceConsumerRunnable(partitionInfo, createConsumer(),
                            readCountStatusUpdate: 10, index, stopAll, workerCount);
            executorService.submit(stockPriceConsumer);
        });
```

Using assign() for Priority Queue



StockPriceConsumerRunnable | runConsumer()

```
    void runConsumer() throws Exception {
        // Assign a partition.
        consumer.assign(Collections.singleton(topicPartition));
        final Map<String, StockPriceRecord> lastRecordPerStock = new HashMa
        try {
            int readCount = 0;
            while (isRunning()) {
                pollRecordsAndProcess(lastRecordPerStock, readCount);
            }
        } finally {
            consumer.close();
        }
    }
```