[redacted]

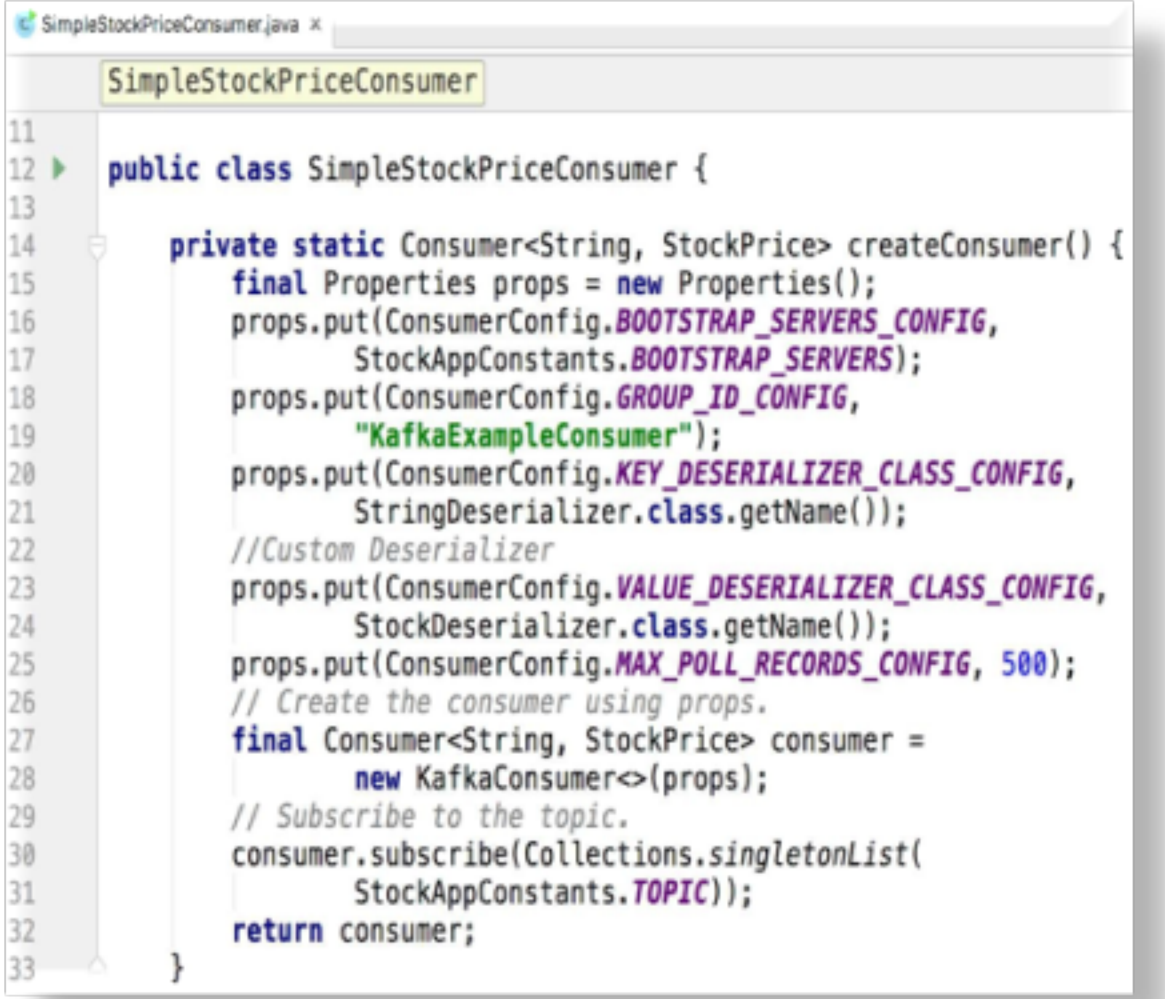
Kafka Consumers Advance

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

Consumer: createConsumer / Consumer Config

Tos



SimpleStockPriceConsumer.java ×

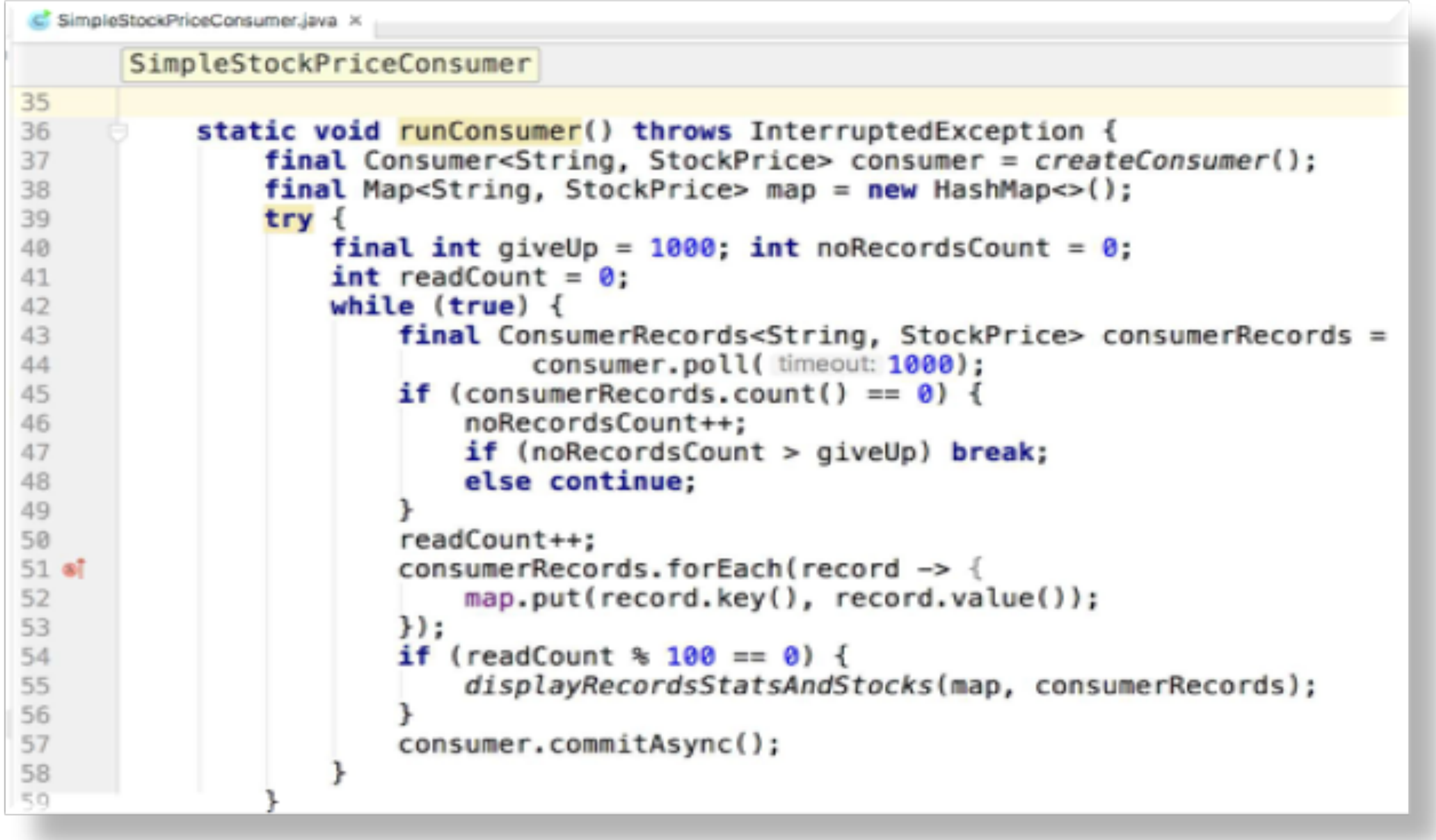
SimpleStockPriceConsumer

```
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

Tos

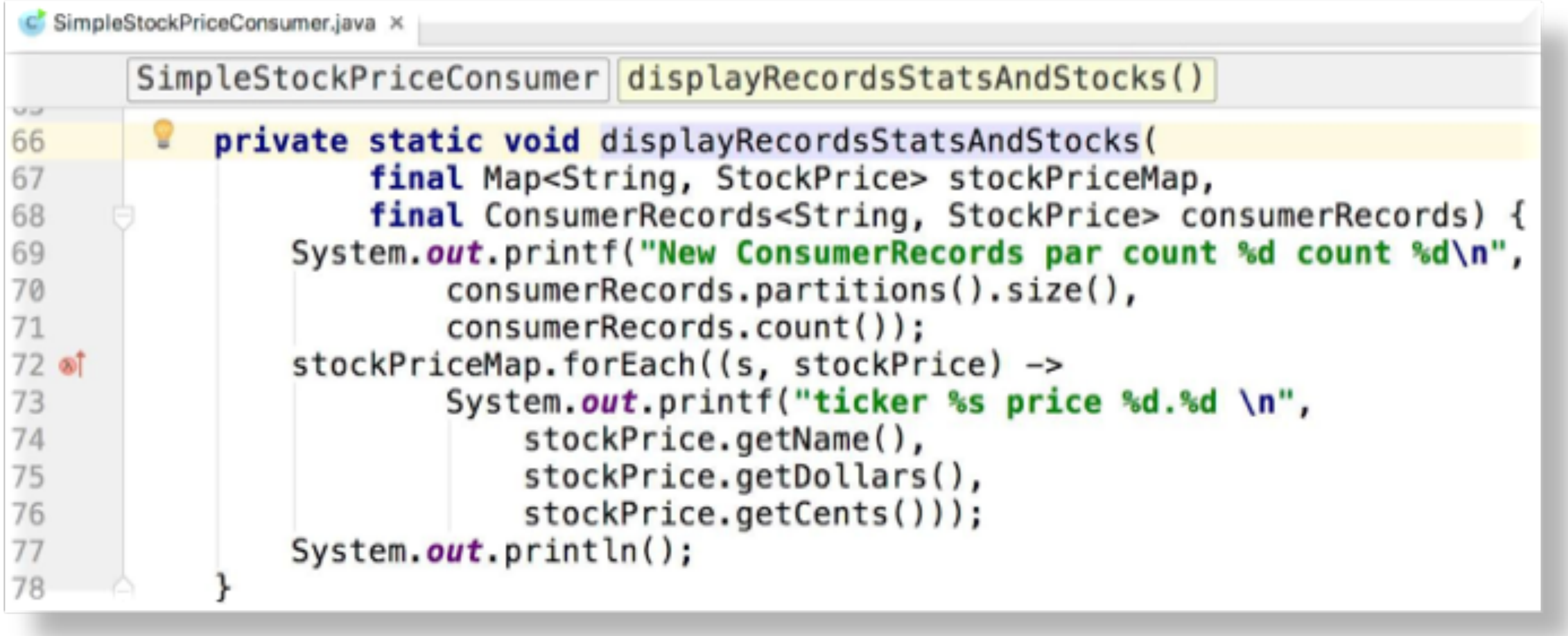


```java

    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

Tos



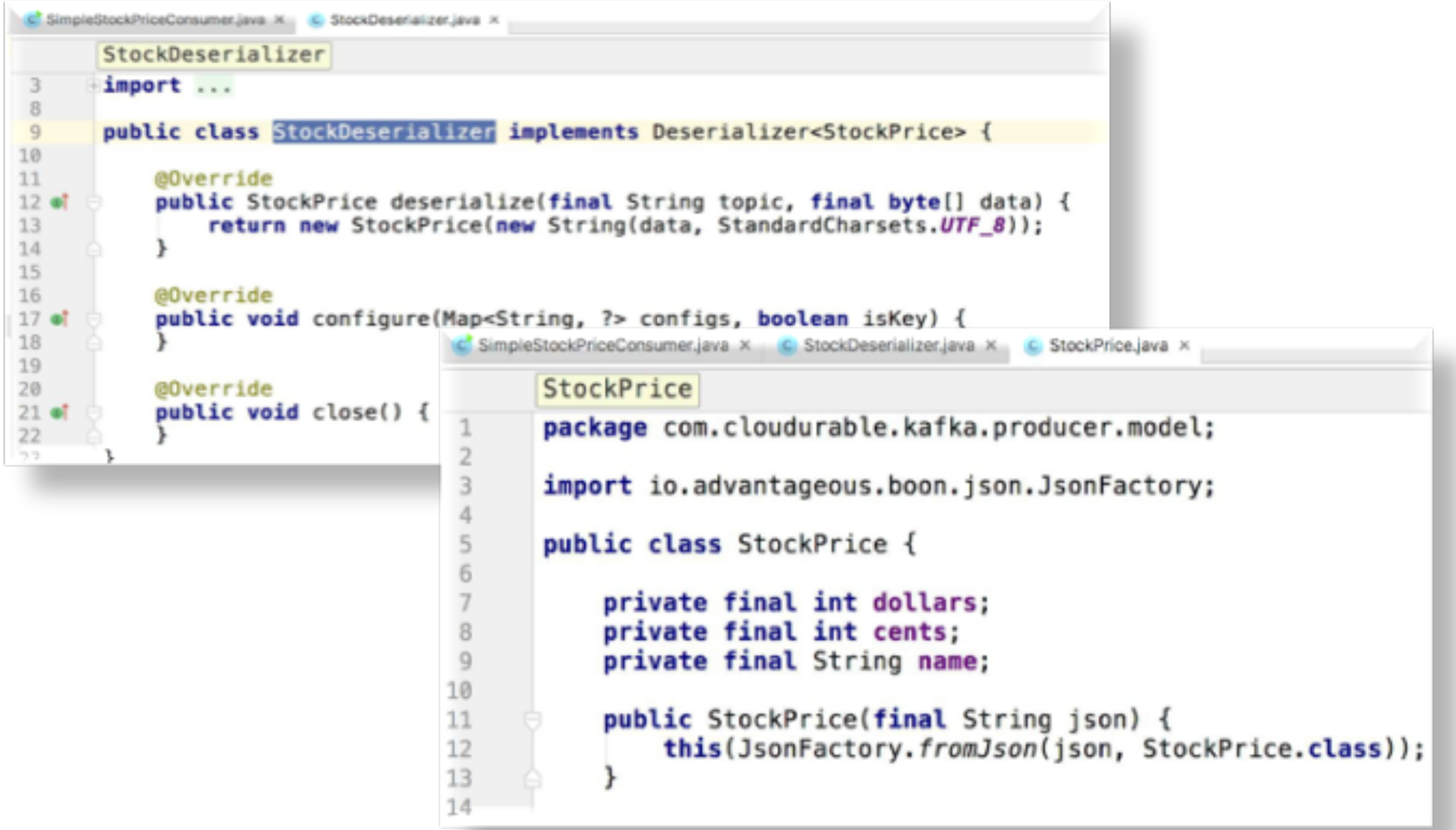
SimpleStockPriceConsumer.java ×
SimpleStockPriceConsumer
displayRecordsStatsAndStocks()

```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

Tos

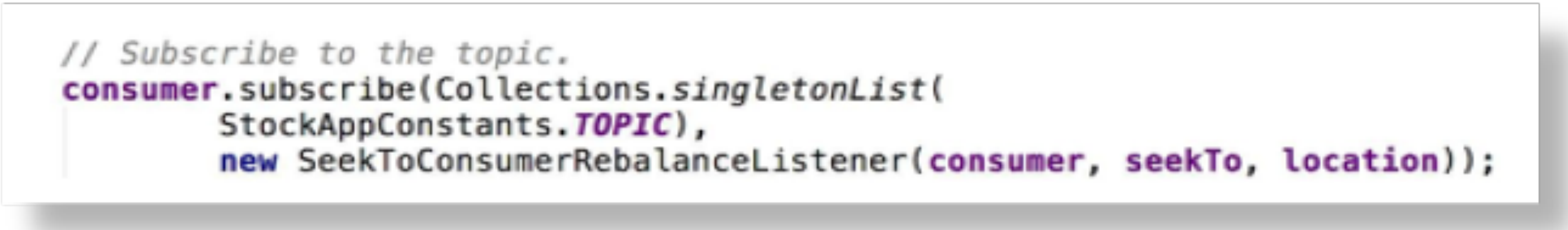


```java
import ...

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() {
    }
}
```

```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));
    }

```

Storing Offsets Outside: Managing Offsets

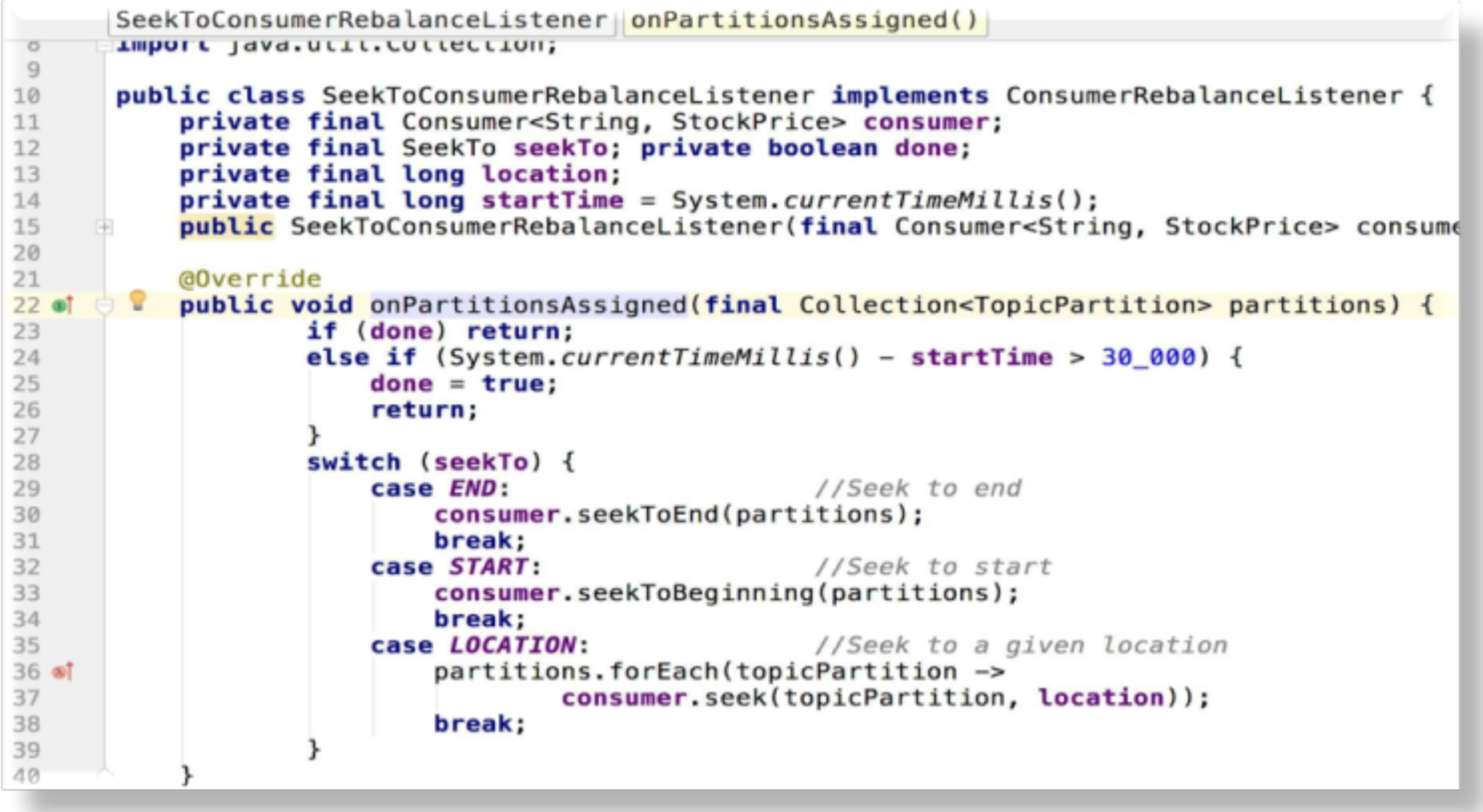
- If using partition assignment, you must handle cases where partition assignments change
  - Pass ***ConsumerRebalanceListener*** instance in call to ***kafkaConsumer.subscribe***(Collection, ConsumerRebalanceListener) and ***kafkaConsumer.subscribe***(Pattern, ConsumerRebalanceListener).
  - when partitions 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

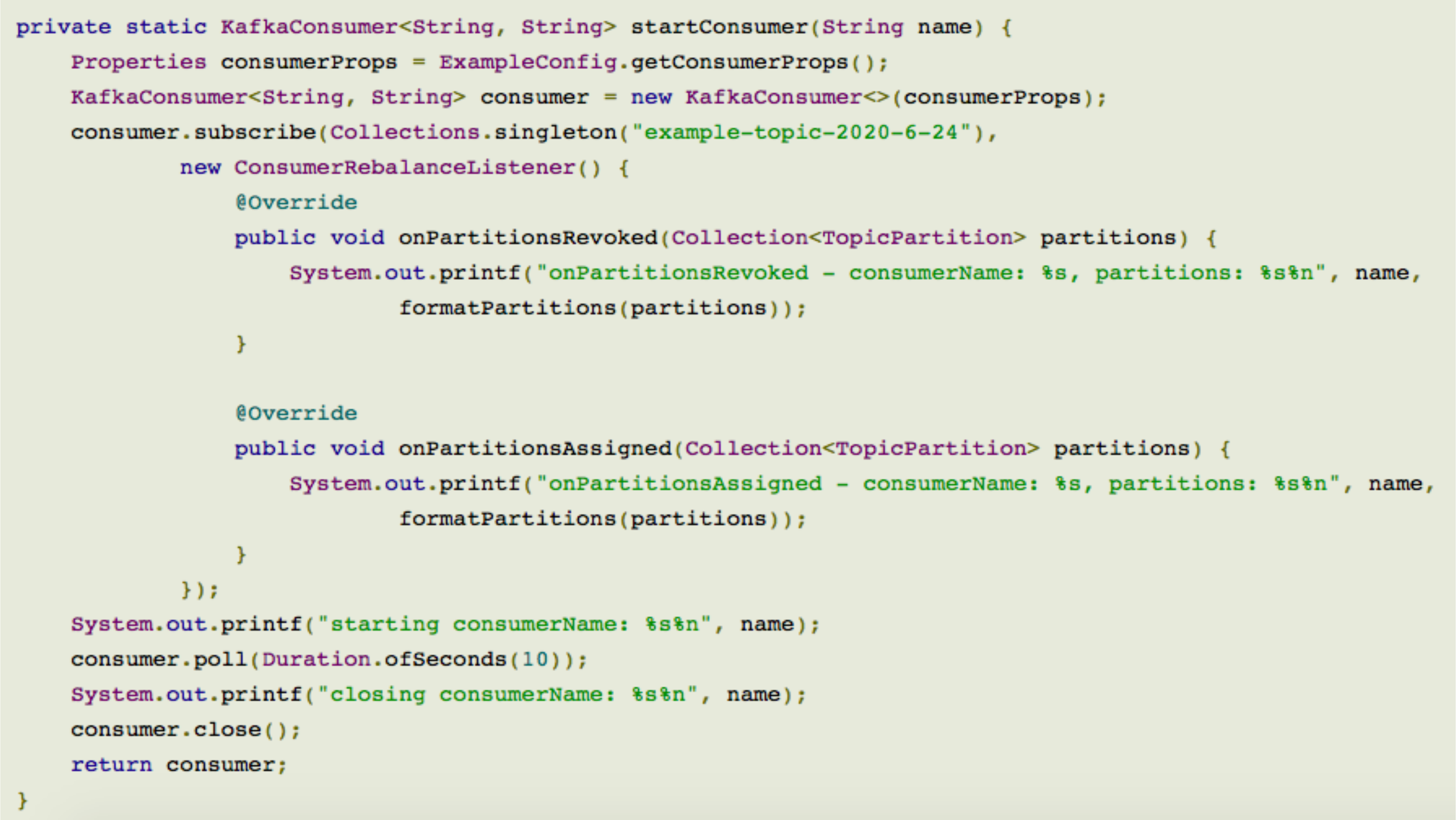
Tos



```
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;
            }
    }
```

Controlling Consumers Position Example

Tos



```
private static KafkaConsumer<String, String> startConsumer(String name) {
    Properties consumerProps = ExampleConfig.getConsumerProps();
    KafkaConsumer<String, String> consumer = new KafkaConsumer<>(consumerProps);
    consumer.subscribe(Collections.singleton("example-topic-2020-6-24"),
            new ConsumerRebalanceListener() {
                @Override
                public void onPartitionsRevoked(Collection<TopicPartition> partitions) {
                    System.out.printf("onPartitionsRevoked - consumerName: %s, partitions: %s%n", name,
                            formatPartitions(partitions));
                }

                @Override
                public void onPartitionsAssigned(Collection<TopicPartition> partitions) {
                    System.out.printf("onPartitionsAssigned - consumerName: %s, partitions: %s%n", name,
                            formatPartitions(partitions));
                }
            });
    System.out.printf("starting consumerName: %s%n", name);
    consumer.poll(Duration.ofSeconds(10));
    System.out.printf("closing consumerName: %s%n", name);
    consumer.close();
    return consumer;
}
```

KafkaConsumer:Cosumer Alive Detection

Tos

- 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

Tos

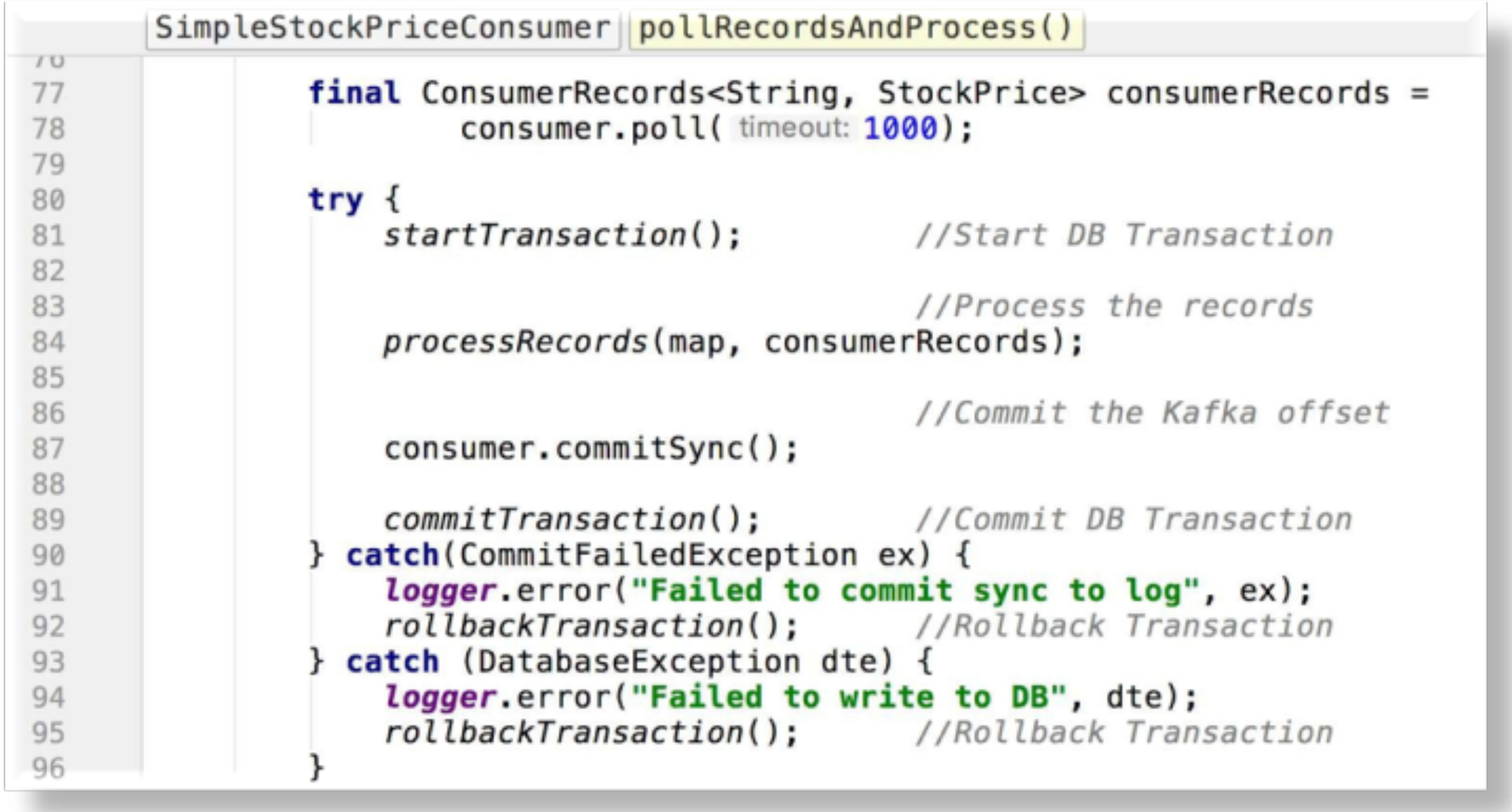
- Calling ***poll()*** marks consumer as alive
  - If consumer continues to call poll(), then consumer is alive and in consumer group and gets messages for partitions assigned (has to call before every ***max.poll.interval.ms interval***)
  - Not calling ***poll()***, even if consumer is sending heartbeats, 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

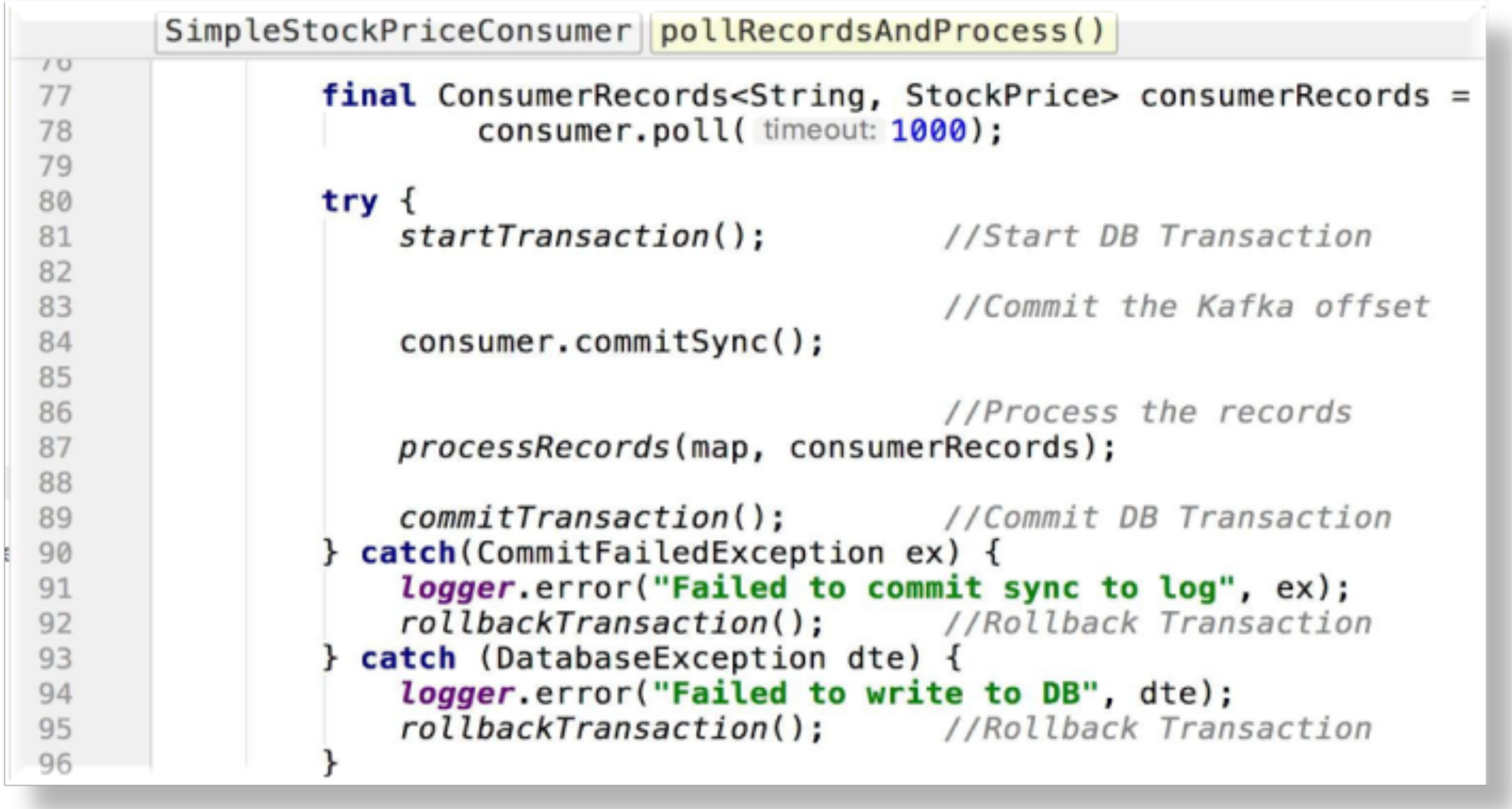
Tos



```java
        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

Tos



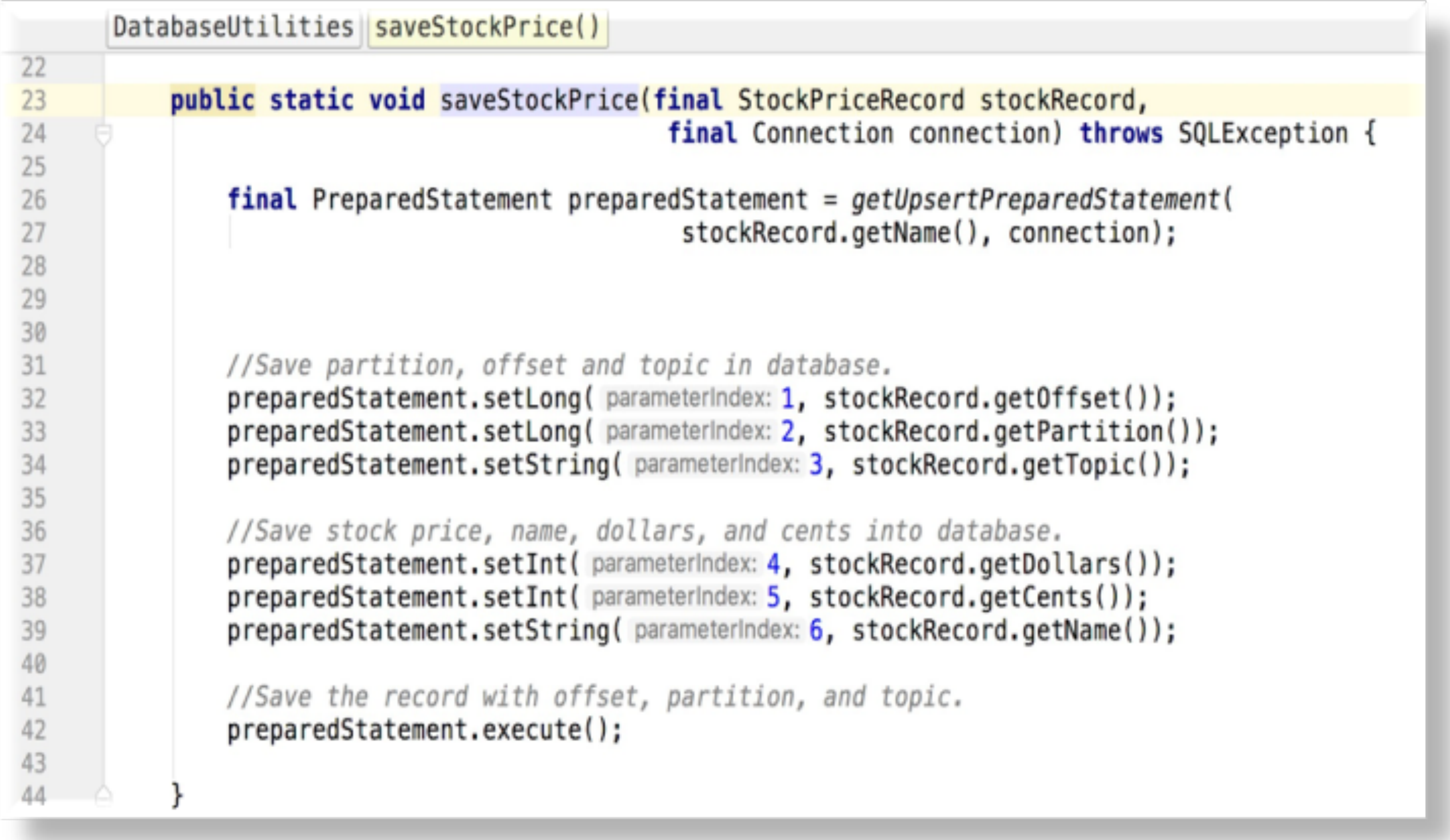
```
        final ConsumerRecords<String, StockPrice> consumerRecords =
                consumer.poll( timeout: 1000);

        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
        }
```

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

Tos



DatabaseUtilities | saveStockPrice()

```java

    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();

    }
```

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)***

Transaction

In order for this to work, consumers reading from transactional partitions should be configured to only read committed data.
This can be achieved by by setting the ***isolation.level=read_committed*** in the consumer's configuration.

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.

Message Deserialization - Avro

```
public class AvroConsumerExample {

public static void main(String[] str) throws InterruptedException {
    System.out.println("Starting AutoOffsetAvroConsumerExample ...");
    readMessages();
}
private static void readMessages() throws InterruptedException {
    KafkaConsumer<String, byte[]> consumer = createConsumer();
    // Assign to specific topic and partition.
    consumer.assign(Arrays.asList(new TopicPartition("avro-topic", 0)));
    processRecords(consumer);
}
```

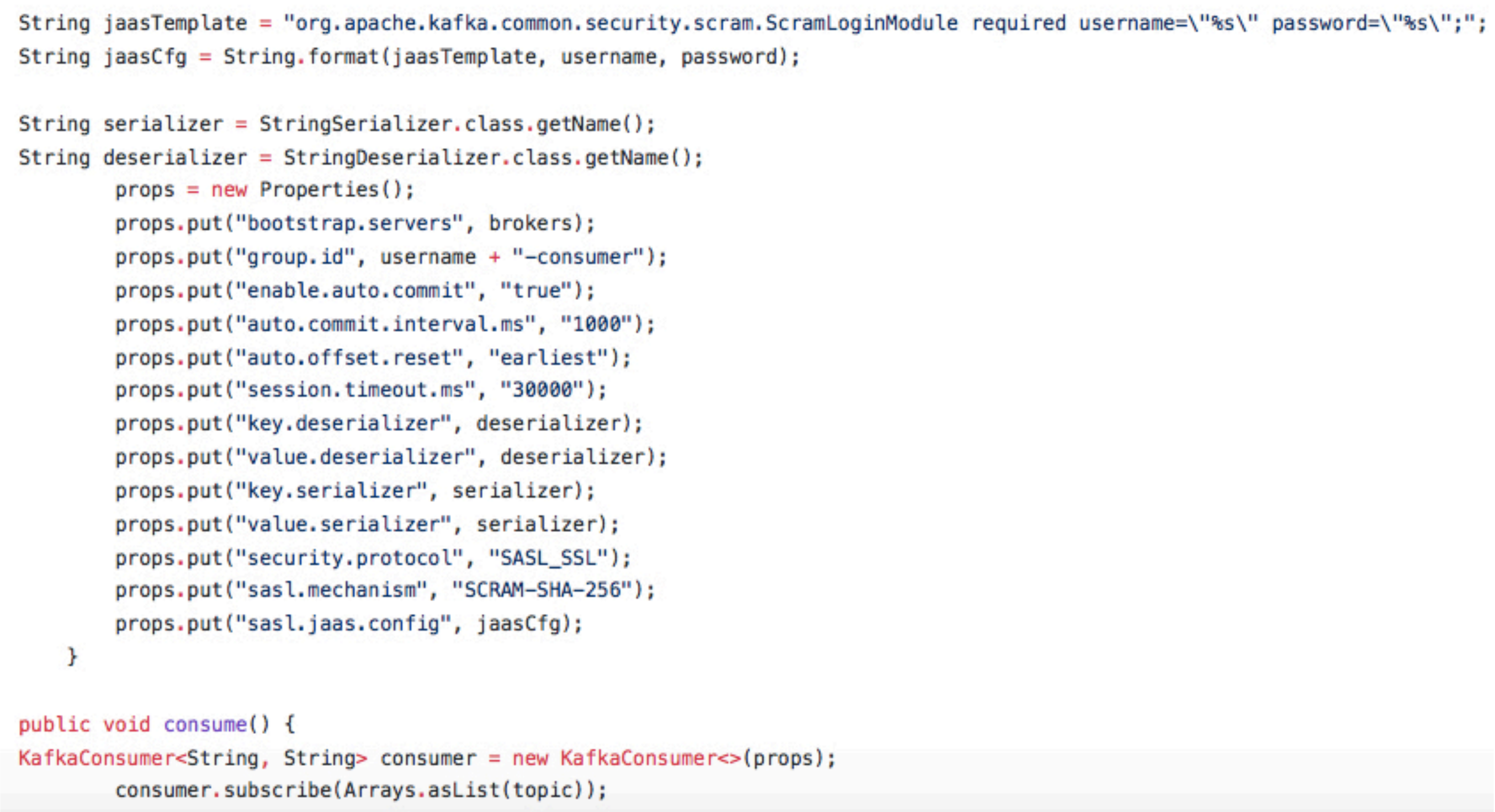
Message Deserialization - Avro

```
private static void processRecords(KafkaConsumer<String, byte[]> consumer) throws {
while (true) {
    ConsumerRecords<String, byte[]> records = consumer.poll(100);
    long lastOffset = 0;
for (ConsumerRecord<String, byte[]> record : records) {
    GenericRecord genericRecord = AvroSupport.byteArrayToData(AvroSupport.getSchema(), record.value());
    String firstName = AvroSupport.getValue(genericRecord, "firstName", String.class);
    System.out.printf("\n\roffset = %d, key = %s, value = %s", record.offset(), record.key(),
    firstName);
    lastOffset = record.offset();
}
        System.out.println("lastOffset read: " + lastOffset);
        consumer.commitSync();
}
}
```

Message Deserialization - Avro

```
private static KafkaConsumer<String, byte[]> createConsumer() {
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
String consumeGroup = "cg1";
props.put("group.id", consumeGroup);
props.put("enable.auto.commit", "true");
props.put("auto.offset.reset", "earliest");
props.put("auto.commit.interval.ms", "100");
props.put("heartbeat.interval.ms", "3000");
props.put("session.timeout.ms", "30000");
props.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
props.put("value.deserializer", "org.apache.kafka.common.serialization.ByteArrayDeserializer");
return new KafkaConsumer<String, byte[]>(props);
}
}
```

Client with Security



```
String jaasTemplate = "org.apache.kafka.common.security.scram.ScramLoginModule required username=\"%s\" password=\"%s\";";
String jaasCfg = String.format(jaasTemplate, username, password);

String serializer = StringSerializer.class.getName();
String deserializer = StringDeserializer.class.getName();
    props = new Properties();
    props.put("bootstrap.servers", brokers);
    props.put("group.id", username + "-consumer");
    props.put("enable.auto.commit", "true");
    props.put("auto.commit.interval.ms", "1000");
    props.put("auto.offset.reset", "earliest");
    props.put("session.timeout.ms", "30000");
    props.put("key.deserializer", deserializer);
    props.put("value.deserializer", deserializer);
    props.put("key.serializer", serializer);
    props.put("value.serializer", serializer);
    props.put("security.protocol", "SASL_SSL");
    props.put("sasl.mechanism", "SCRAM-SHA-256");
    props.put("sasl.jaas.config", jaasCfg);
  }

public void consume() {
KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);
    consumer.subscribe(Arrays.asList(topic));
```

Lab : Java API – Consumer – II