TEKCRUX

We ReDefine IT

kafka

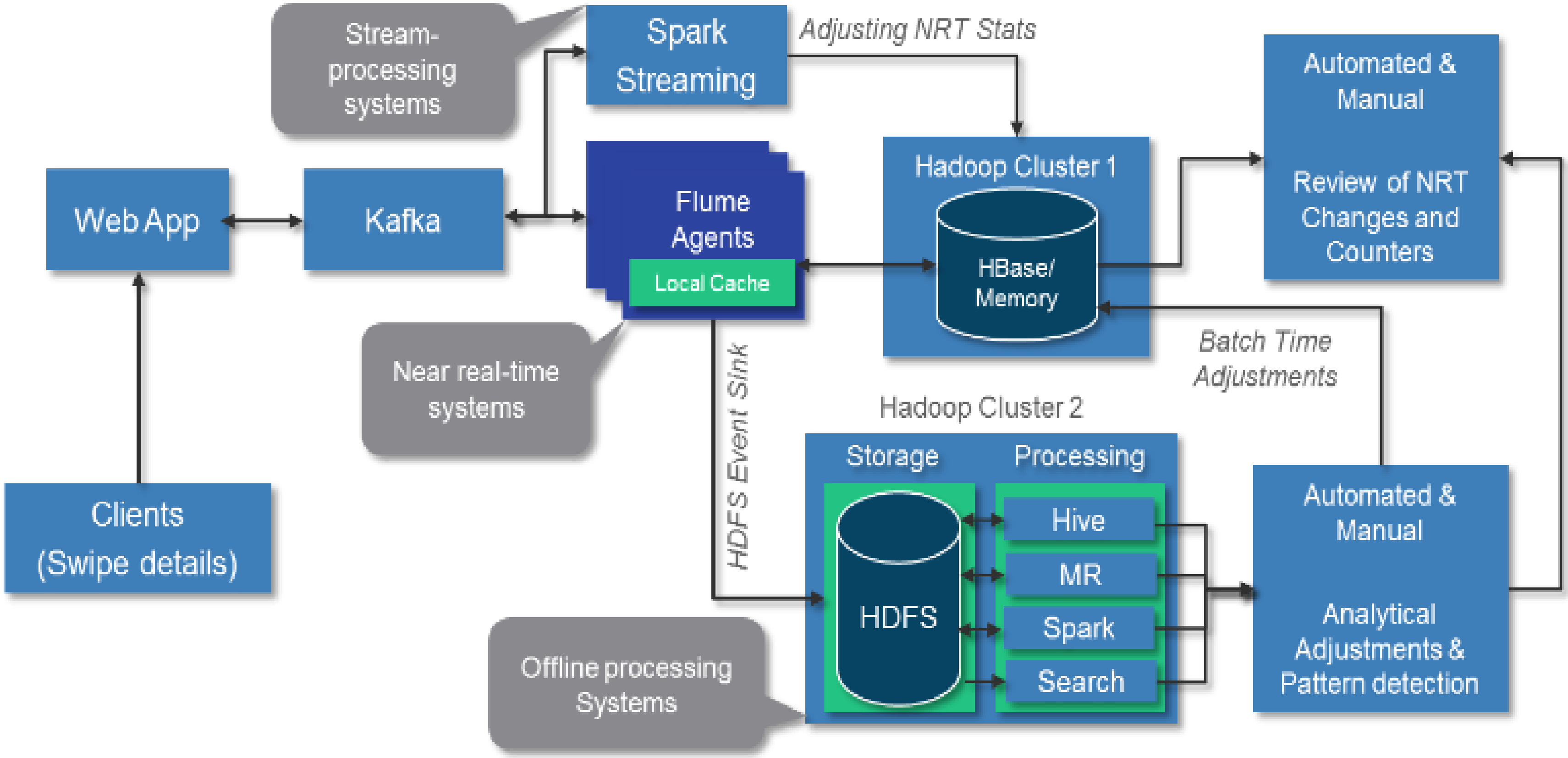
Module 3: Kafka Consumer

Objectives

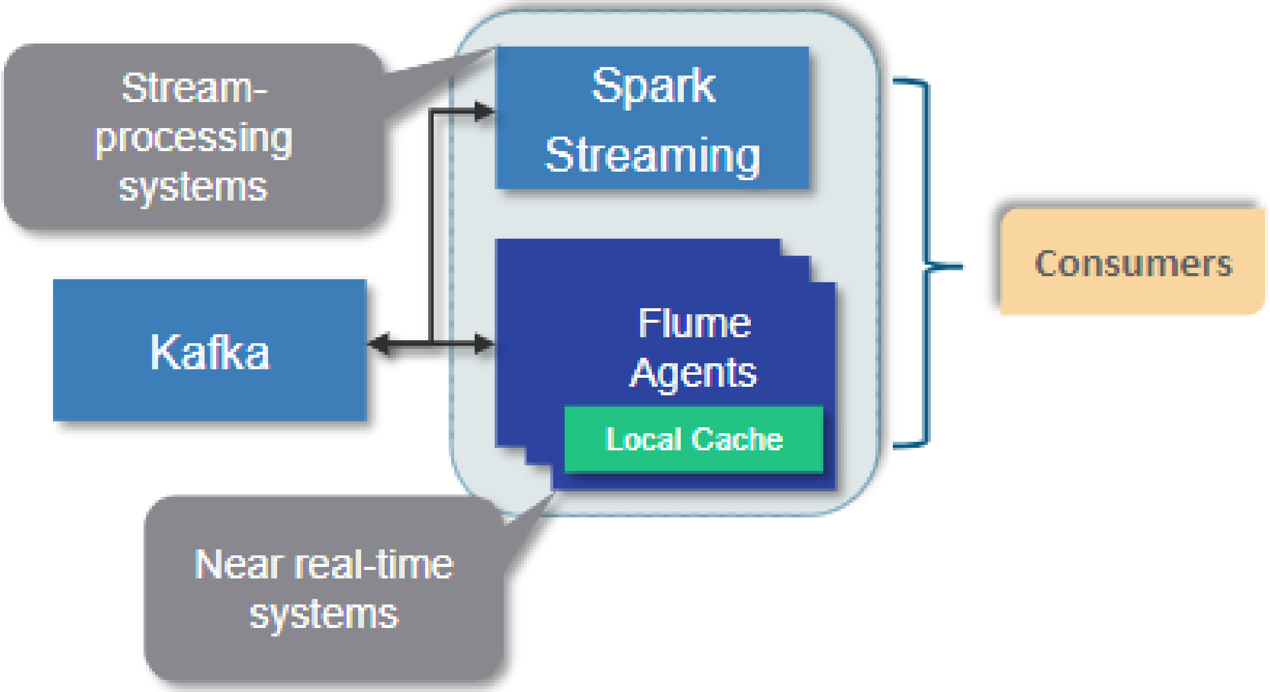
After completing of this module, you should be able to:

- ✓ Define Kafka consumer and Consumer Groups
- ✓ Understand partition rebalancing
- ✓ Define how partitions are assigned o Kafka Brokers
- ✓ Configure Kafka Consumer
- ✓ Create a Kafka consumer and subscribe to topics
- ✓ Describe and implement different types of commit
- ✓ Deserialize the received messages

Recall: Credit Card Transaction Processing



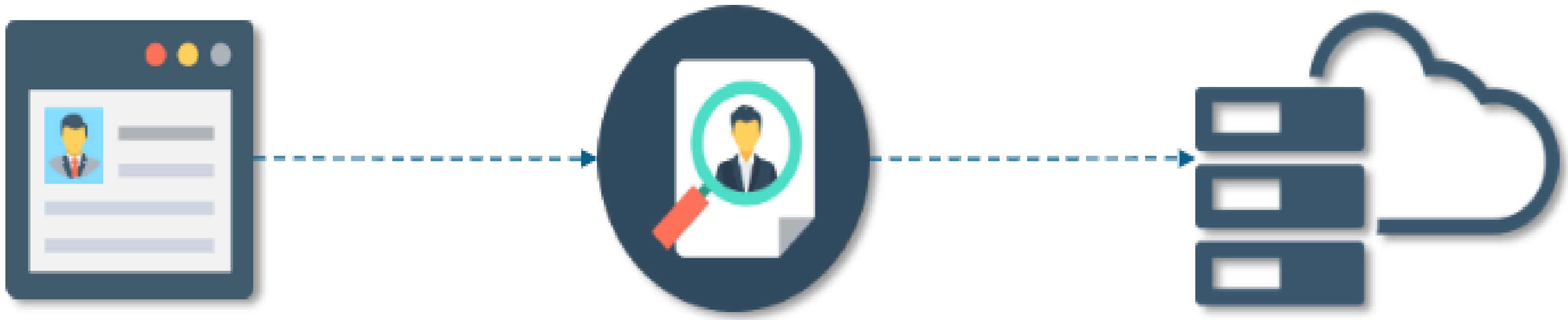
Credit Card System: Consumers



Kafka Consumer

A **consumer** can be any application that subscribes to a topic and consume the messages

You have an application that needs to read messages from a Kafka topic, run some validations against them, and write the results to another data store



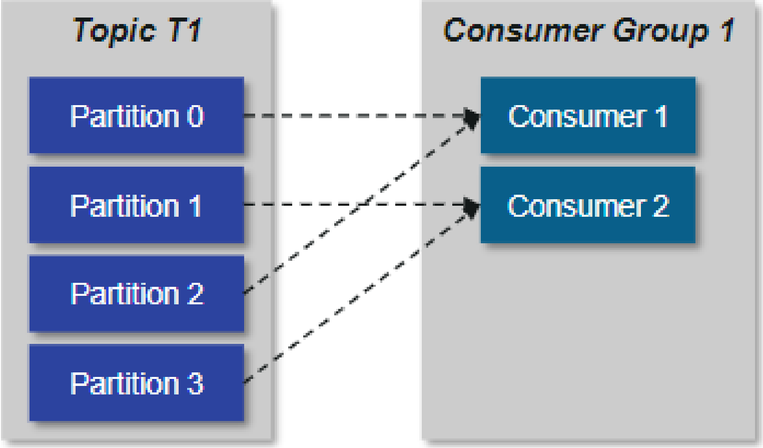
Application creating User Profiles

Validation of User Profiles

Storing data in Data Store

Kafka Consumer – Adding more Consumers

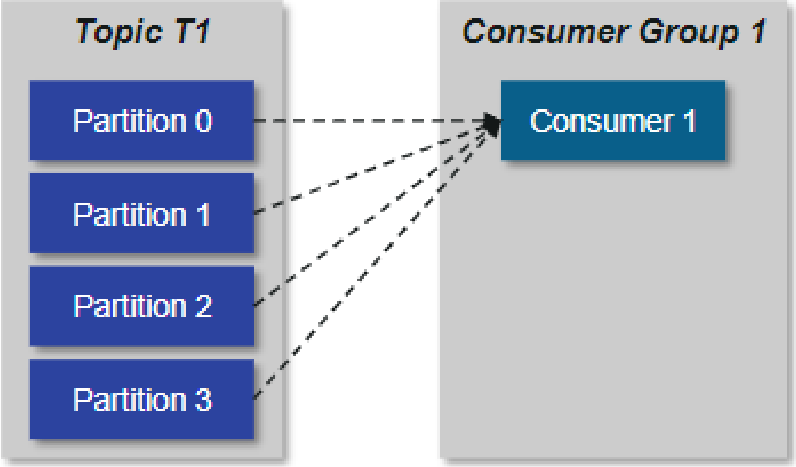
If you are limited to a single consumer, the problem would be lag in the message consumption, as it will not be able to cope up with the rate of incoming messages
Consumer Lag is a serious problem so there will be message pileup in kafka



Just like multiple producers can write to the same topic, we can allow multiple consumers to read from the same topic, splitting the data between them.

Kafka Consumer - Creating Consumer Object

Your application will create a consumer object, subscribe to the appropriate topic, and start receiving messages, validating them and writing the results

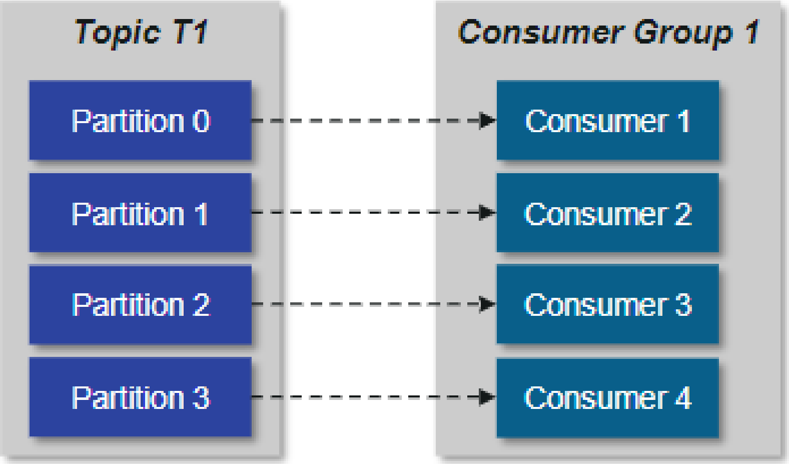


☛ *This may work well for a while, but what if the rate at which producers write messages to the topic exceeds the rate at which your application can validate them?*

Kafka Consumer - Creating Consumer Object

If Consumer Group 1 has four consumers, then each will read messages from a single partition

Create topics with a large number of partitions-it allows adding more consumers when the load increases

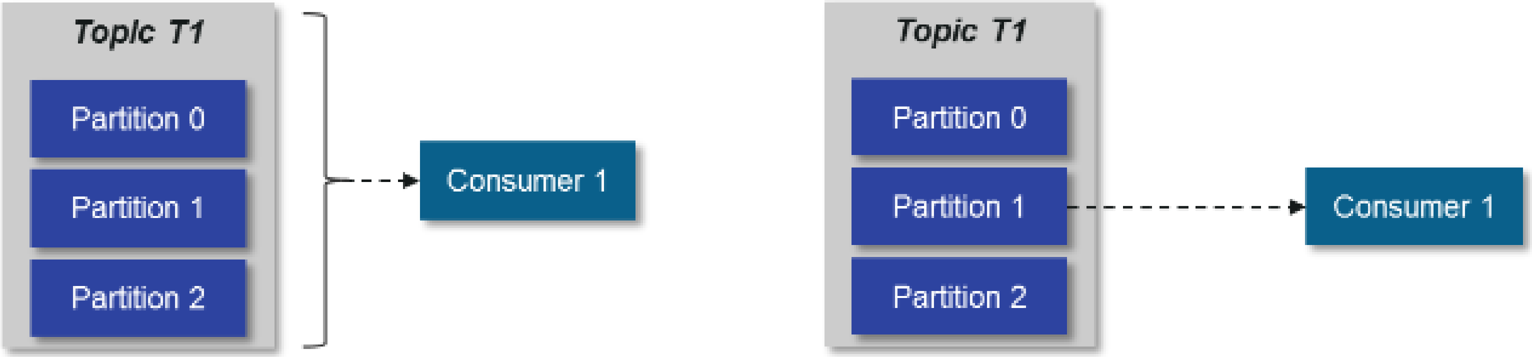


☛ *If we add more consumers to a single group with a single topic, some of the consumers will be idle and get no messages at all.*

Before moving onto Consumer Groups
Let’s see what is a Standalone Consumer

Standalone Consumer

You can have a single consumer & it can read data from all the partitions in a topic or from a specific partition in a topic



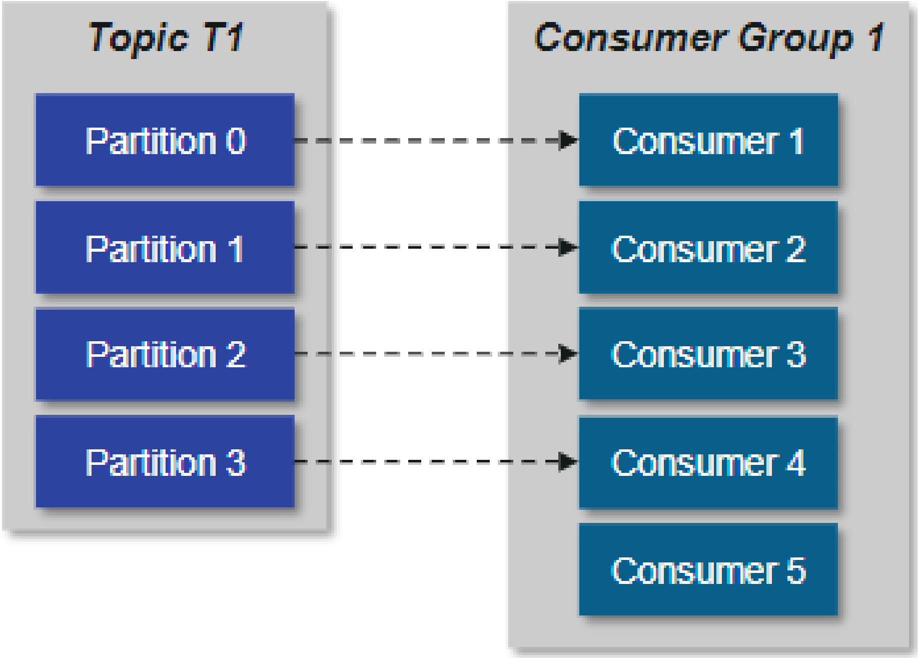
- In this case, there is no reason for groups or rebalances
- Just assign the consumer-specific topic and/or partitions, consume messages & commit offsets
- If we know which partitions the consumer should read, instead of subscribing to a topic we can assign partitions

A consumer can either subscribe to topics, or assign itself partitions, but not both at the same time

Kafka Consumer Groups

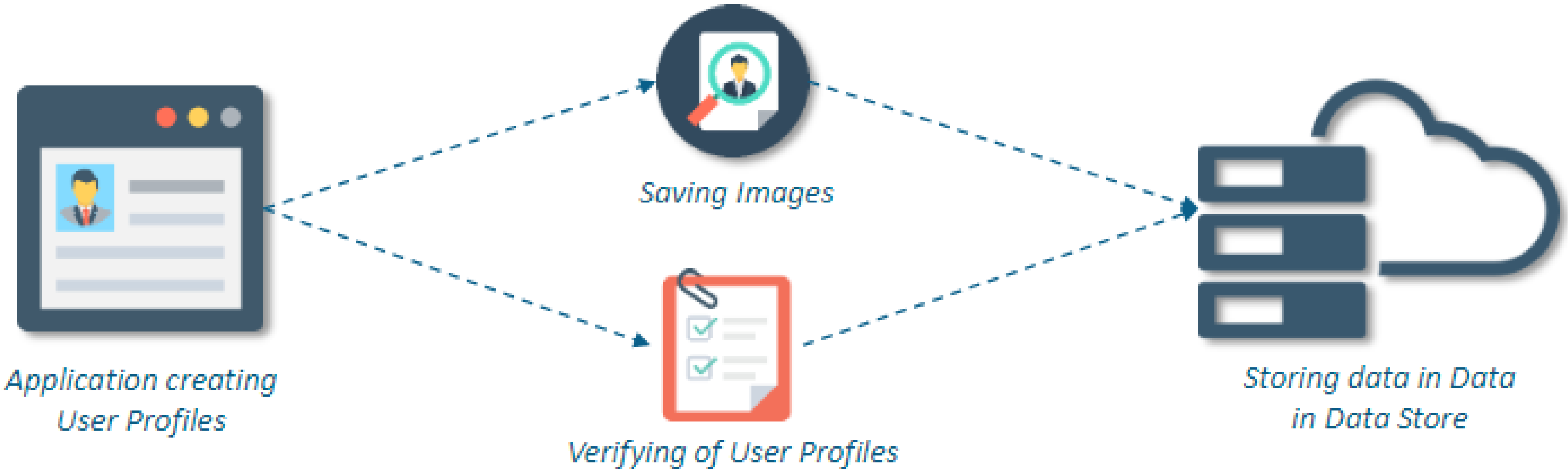
Kafka Consumer Groups

It is a good practice to have 1 or 2 idle consumers in the consumer group as these idle consumer works as fail over for other consumer in the consumer group



Kafka Consumer Groups

Suppose the data is going to two separate applications, one will save user images and the other will verify the user profiles

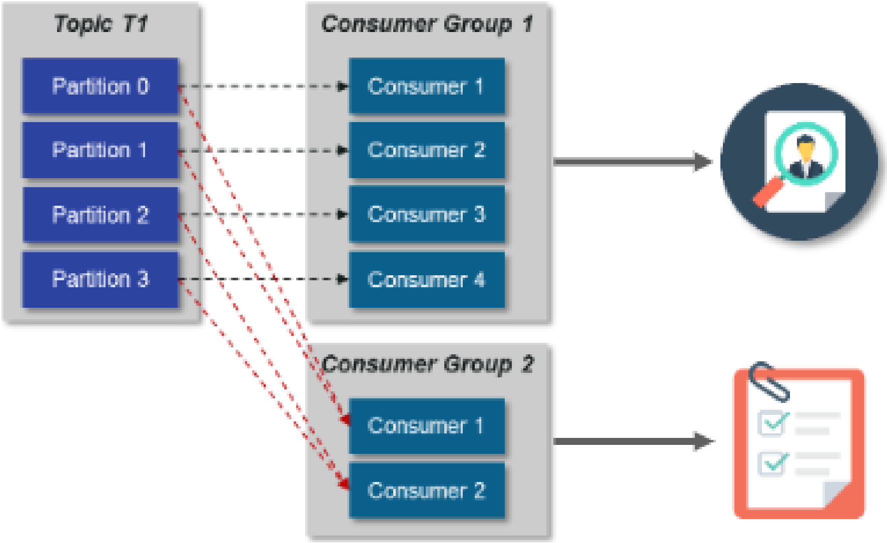

Saving Images
Application creating User Profiles
Verifying of User Profiles
Storing data in Data in Data Store

- Design goals of Kafka is to make the produced data available for many use cases throughout the organization
- To make sure an application gets all the messages in a topic, ensure the application has its own consumer group

Kafka Consumer Groups

In order to scale a single application, it is very common to have multiple applications that need to read data from the same topic

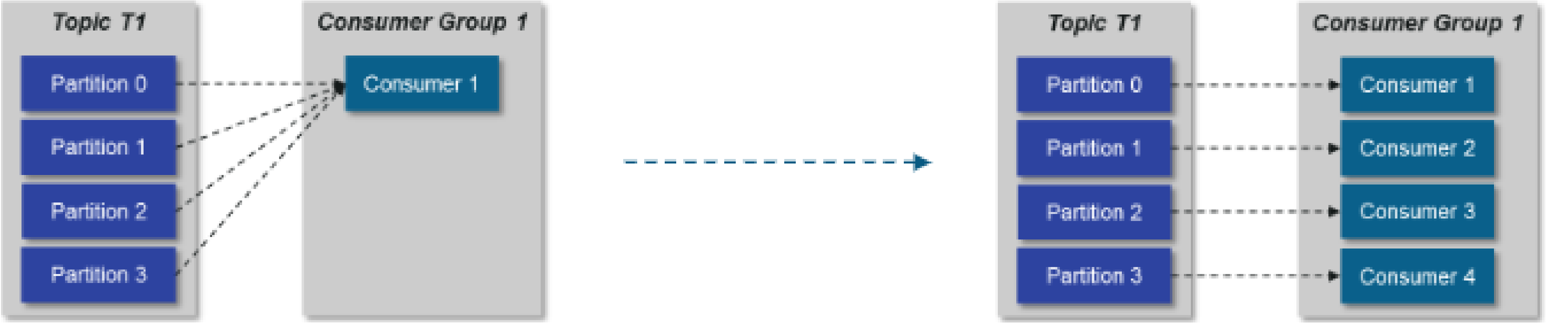
Kafka scales to a large number of consumers and consumer groups without reducing performance



☛ *When multiple consumers are subscribed to a topic and belong to the same consumer group, each consumer in the group will receive messages from a different subset of the partitions in the topic*

Kafka Consumer Groups

When we add a new consumer to the group, it starts consuming messages from partitions previously consumed by another consumer



Reassignment of partitions to consumers also happen when the topics the consumer group is consuming are modified

☛ *Same thing happens when a consumer shuts down or crashes, it leaves the group, and the partitions it used to consume will be consumed by one of the remaining consumers.*

Kafka Consumer Groups
&
Partition Rebalancing

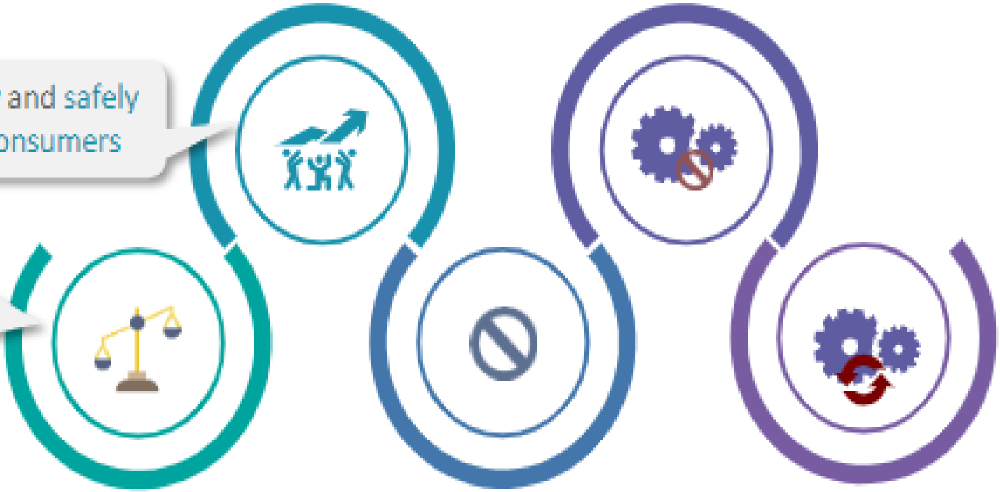
Kafka Consumer Groups & Partition Rebalance

Moving partition ownership from one consumer to another is called a *rebalance*.

Rebalances are important as they provide consumer group with *high availability* and *scalability*

Kafka Consumer Groups & Partition Rebalance

Moving partition ownership from one consumer to another is called a *rebalance*.

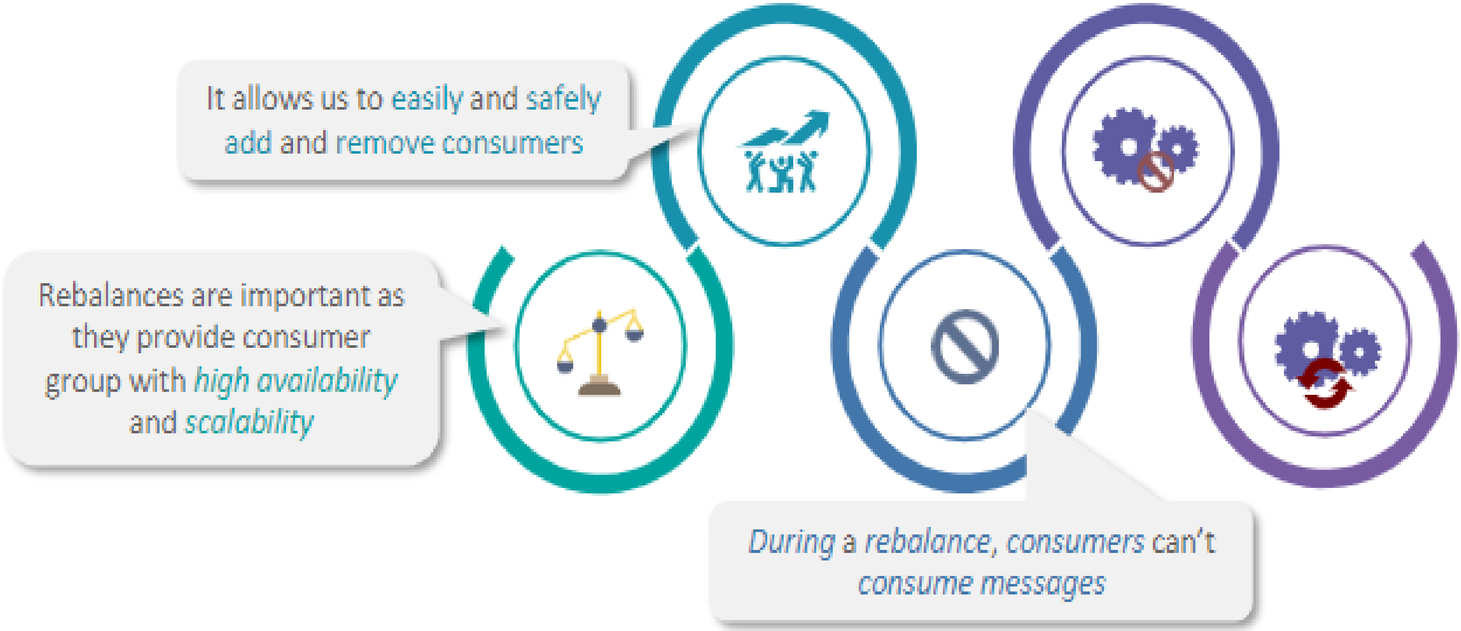


It allows us to easily and safely add and remove consumers

Rebalances are important as they provide consumer group with *high availability* and *scalability*

Kafka Consumer Groups & Partition Rebalance

Moving partition ownership from one consumer to another is called a *rebalance*.



It allows us to easily and safely add and remove consumers

Rebalances are important as they provide consumer group with *high availability* and *scalability*

During a *rebalance, consumers* can't *consume messages*

Kafka Consumer Groups & Partition Rebalance

Moving partition ownership from one consumer to another is called a *rebalance*.

It allows us to easily and safely add and remove consumers

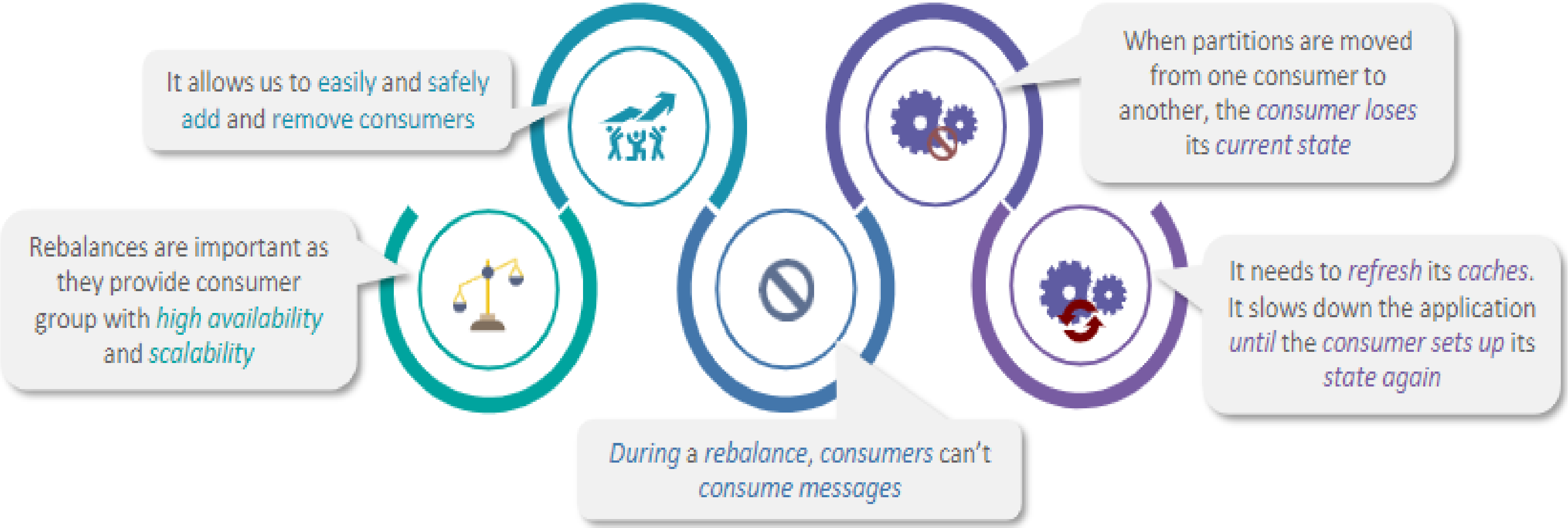
Rebalances are important as they provide consumer group with *high availability* and *scalability*

During a *rebalance, consumers* can't *consume messages*

When partitions are moved from one consumer to another, the *consumer loses* its *current state*

Kafka Consumer Groups & Partition Rebalance

Moving partition ownership from one consumer to another is called a *rebalance*.



It allows us to easily and safely add and remove consumers

Rebalances are important as they provide consumer group with *high availability* and *scalability*

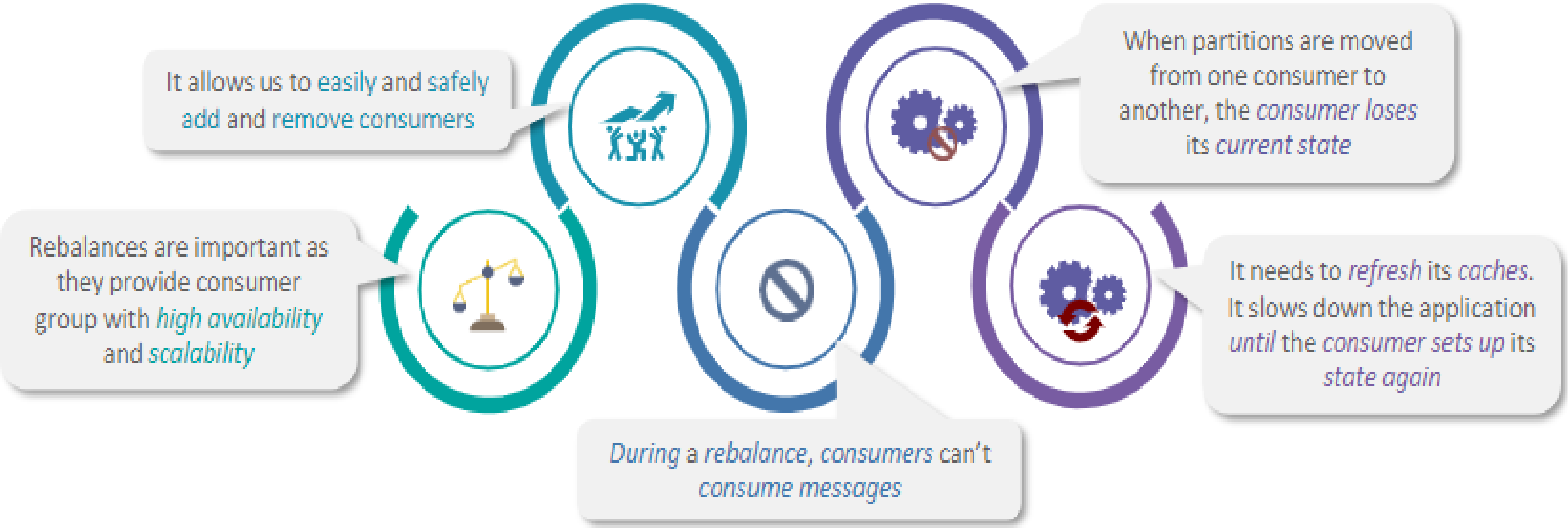
During a rebalance, consumers can't *consume messages*

When partitions are moved from one consumer to another, the *consumer loses* its *current state*

It needs to *refresh* its *caches*. It slows down the application *until* the *consumer sets up* its *state again*

Kafka Consumer Groups & Partition Rebalance

Moving partition ownership from one consumer to another is called a *rebalance*.



It allows us to easily and safely add and remove consumers

Rebalances are important as they provide consumer group with *high availability* and *scalability*

During a *rebalance, consumers* can't *consume messages*

When partitions are moved from one consumer to another, the *consumer loses* its *current state*

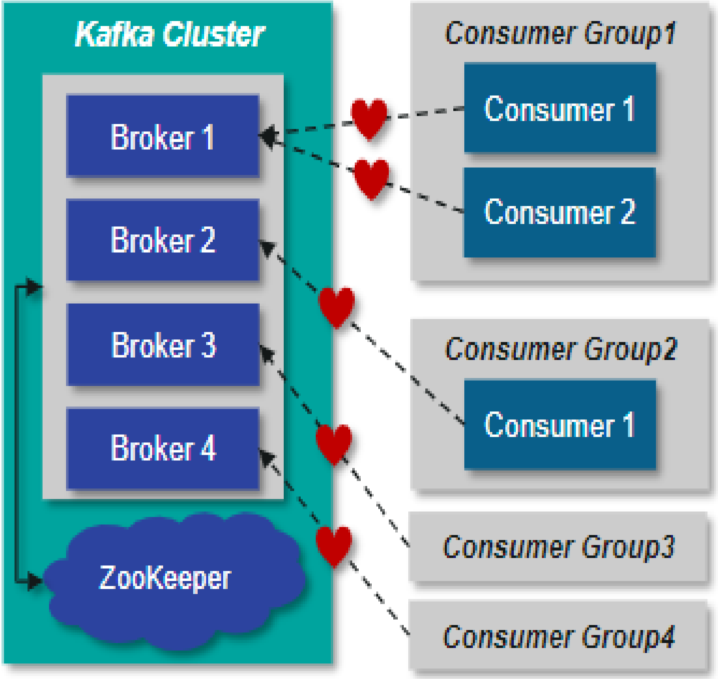
It needs to *refresh* its *caches*. It slows down the application *until* the *consumer sets up* its *state again*

- *Rebalance is a short window of unavailability of the entire consumer group*
- *Important thing to learn is how to safely handle rebalances and how to avoid unnecessary ones*

Kafka Consumer Groups & Partition Rebalance

Consumers maintains membership in a consumer group and ownership of the partitions assigned to them by sending *heartbeats* to a Kafka broker designated as the *group coordinator*

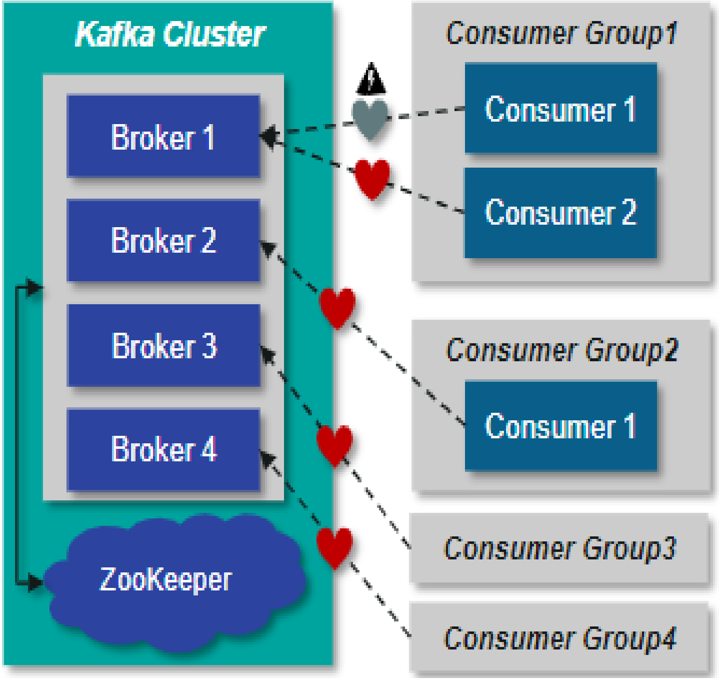
- Broker can be different for different consumer groups
- *As long as the consumer is sending heartbeats at regular intervals, it is assumed to be alive & processing messages from its partitions*
- Heartbeats are sent when the consumer polls & when it commits records it has consumed



Kafka Consumer Groups & Partition Rebalance

If the consumer stops sending heartbeats for long enough, its session will time out and the group coordinator will consider it dead and trigger a *rebalance*

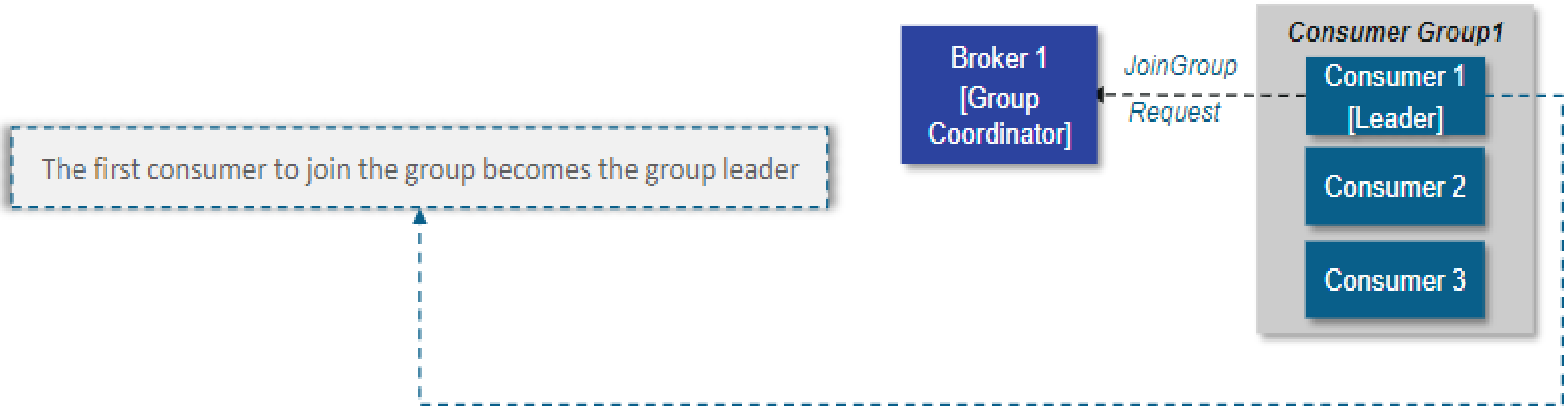
- If a *consumer stops sending heartbeats, group coordinator* will *consider it* as *dead* and *trigger* the *rebalance*
- *During rebalance, no messages* will be *processed* from the partitions owned by the dead consumer
- When *closing* a *consumer cleanly*, the *consumer* will *notify* the *group coordinator* that it is leaving, and the *group coordinator* will *trigger* a *rebalance* immediately



Process of assigning Partitions to Brokers

Process of assigning Partitions to Brokers

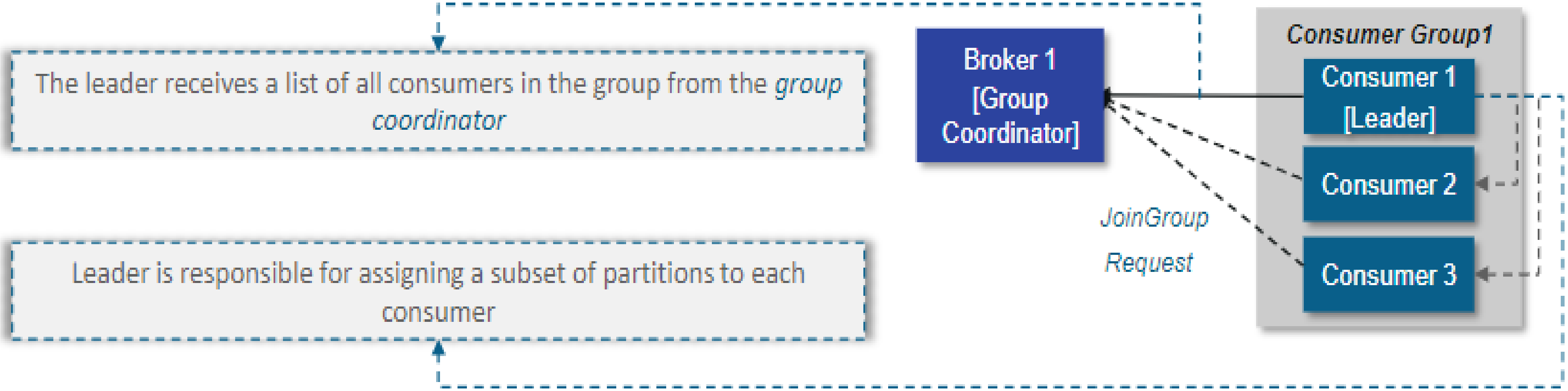
When a consumer wants to join a group, it sends a *JoinGroup* request to the group coordinator



The first consumer to join the group becomes the group leader

Process of assigning Partitions to Brokers

When a consumer wants to join a group, it sends a *JoinGroup* request to the group coordinator



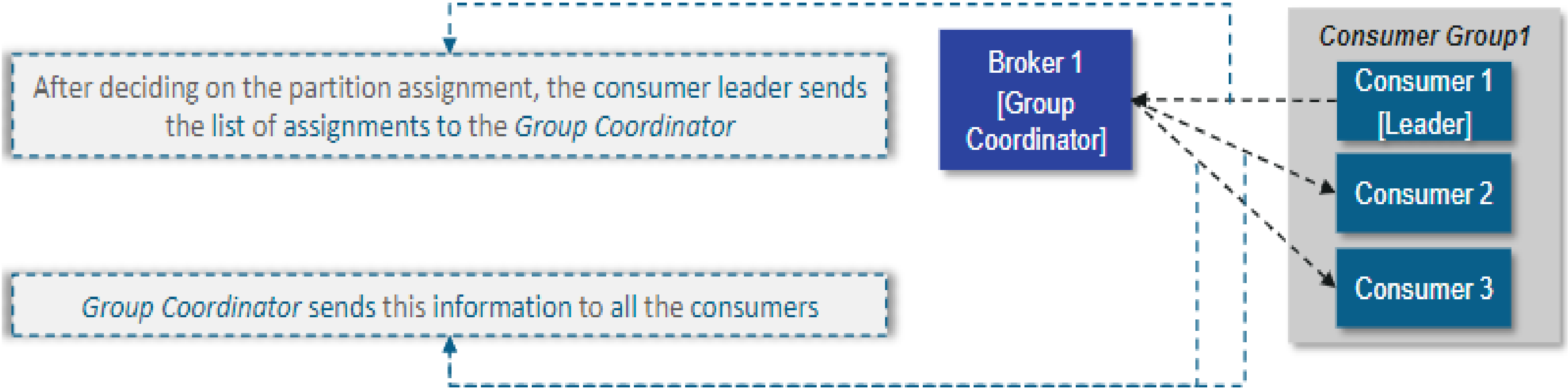
The leader receives a list of all consumers in the group from the *group coordinator*

Leader is responsible for assigning a subset of partitions to each consumer

It uses an implementation of Partition Assignor to decide which partitions should be handled by which consumer

Process of assigning Partitions to Brokers

When a consumer wants to join a group, it sends a *JoinGroup* request to the group coordinator

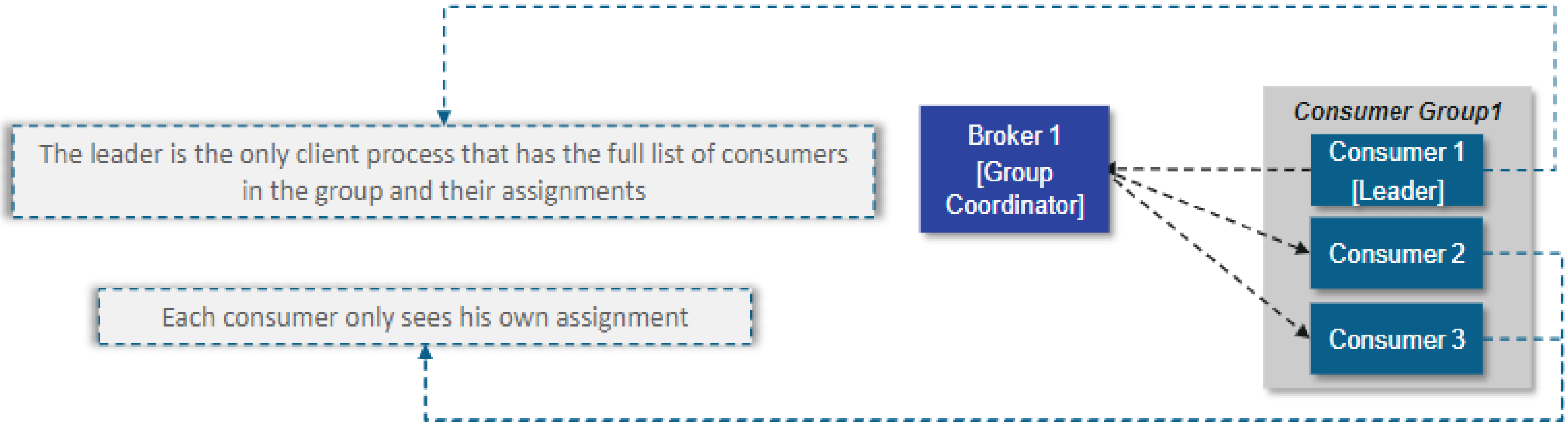


After deciding on the partition assignment, the consumer leader sends the list of assignments to the *Group Coordinator*

Group Coordinator sends this information to all the consumers

Process of assigning Partitions to Brokers

When a consumer wants to join a group, it sends a *JoinGroup* request to the group coordinator



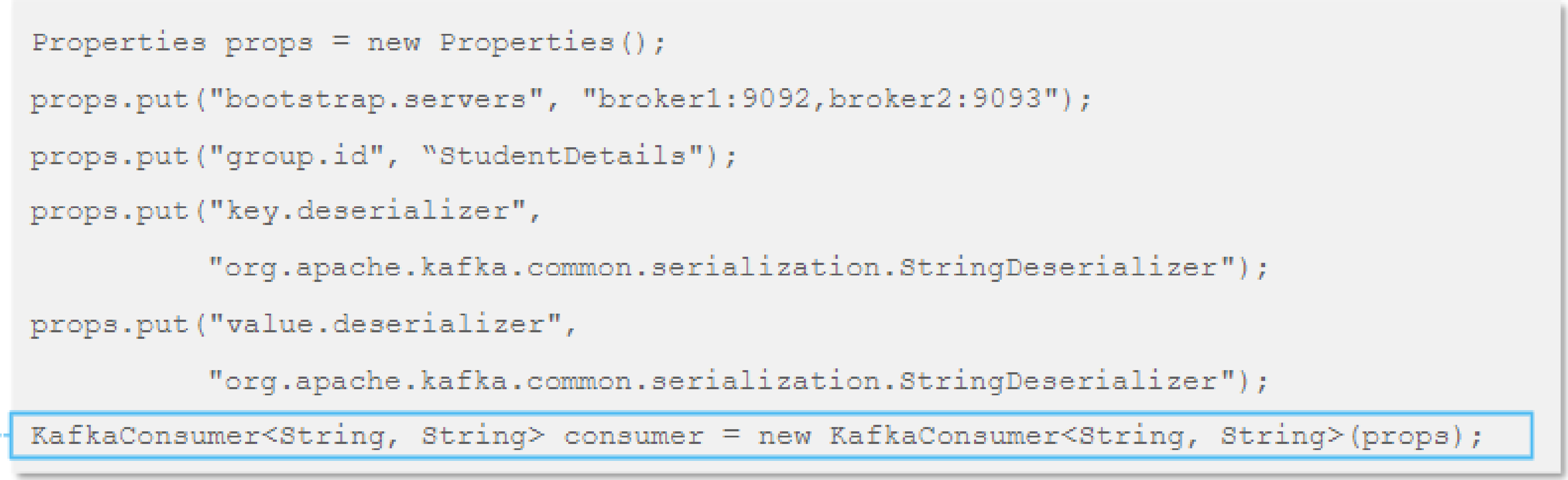
The leader is the only client process that has the full list of consumers in the group and their assignments

Each consumer only sees his own assignment

☛ *This process repeats every time a rebalance happens*

Let’s create a Kafka Consumer

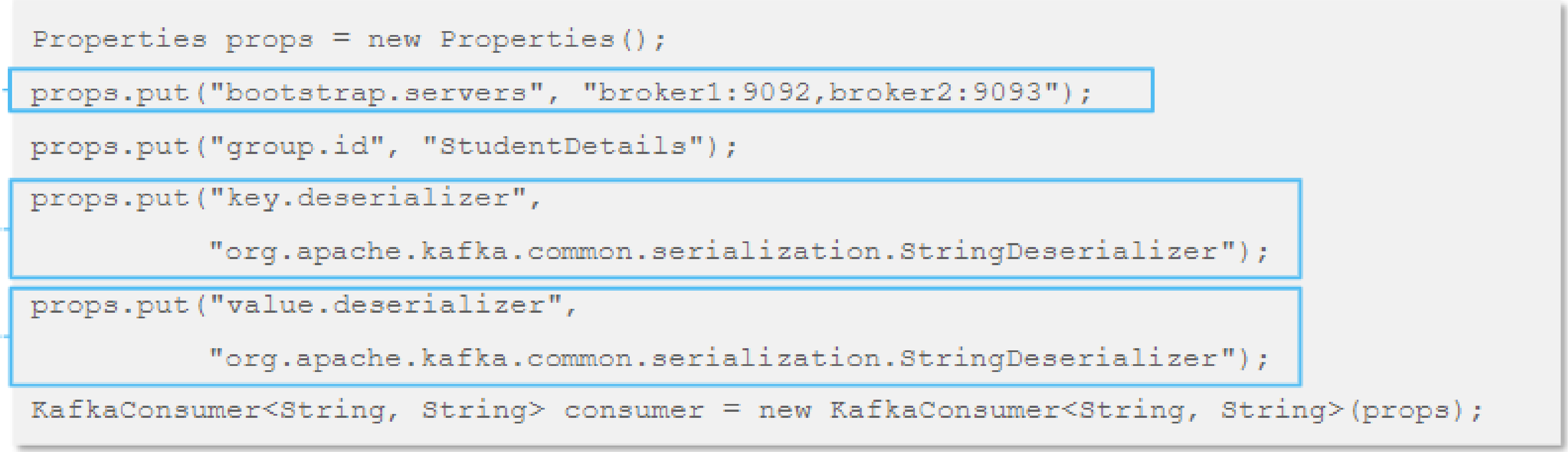
Creating a Kafka Consumer



```
Properties props = new Properties();
props.put("bootstrap.servers", "broker1:9092,broker2:9093");
props.put("group.id", “StudentDetails");
props.put("key.deserializer",
          "org.apache.kafka.common.serialization.StringDeserializer");
props.put("value.deserializer",
          "org.apache.kafka.common.serialization.StringDeserializer");
KafkaConsumer<String, String> consumer = new KafkaConsumer<String, String>(props);
```

The first step to start consuming records is to create a KafkaConsumer instance (object)

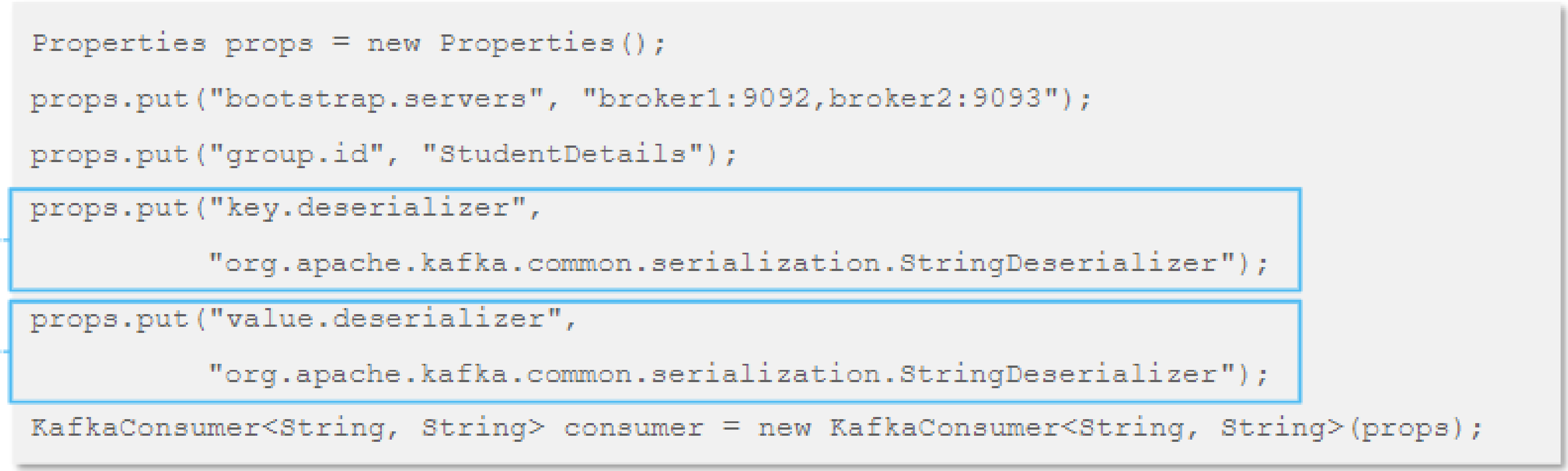
Creating a Kafka Consumer



```
Properties props = new Properties();
props.put("bootstrap.servers", "broker1:9092,broker2:9093");
props.put("group.id", "StudentDetails");
props.put("key.deserializer",
          "org.apache.kafka.common.serialization.StringDeserializer");
props.put("value.deserializer",
          "org.apache.kafka.common.serialization.StringDeserializer");
KafkaConsumer<String, String> consumer = new KafkaConsumer<String, String>(props);
```

The three mandatory properties are: *bootstrap.servers*, *key.deserializer*, and *value.deserializer*

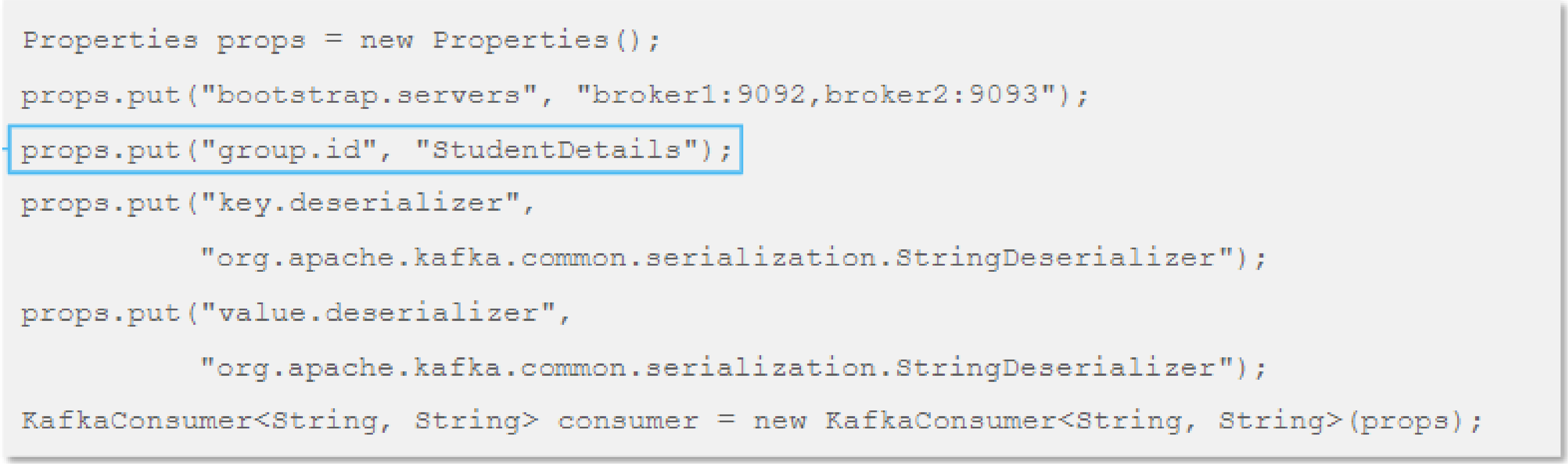
Creating a Kafka Consumer



```
Properties props = new Properties();
props.put("bootstrap.servers", "broker1:9092,broker2:9093");
props.put("group.id", "StudentDetails");
props.put("key.deserializer",
          "org.apache.kafka.common.serialization.StringDeserializer");
props.put("value.deserializer",
          "org.apache.kafka.common.serialization.StringDeserializer");
KafkaConsumer<String, String> consumer = new KafkaConsumer<String, String>(props);
```

key.deserializer and *value.deserializer*, are similar to the producer serializers, but here a byte array turned into a Java object

Creating a Kafka Consumer



```
Properties props = new Properties();
props.put("bootstrap.servers", "broker1:9092,broker2:9093");
props.put("group.id", "StudentDetails");
props.put("key.deserializer",
          "org.apache.kafka.common.serialization.StringDeserializer");
props.put("value.deserializer",
          "org.apache.kafka.common.serialization.StringDeserializer");
KafkaConsumer<String, String> consumer = new KafkaConsumer<String, String>(props);
```

The property *group.id* specifies the consumer group in which the *KafkaConsumer* instance belongs

☛ *It is possible to create consumers that do not belong to any consumer group, but this is not recommended*

Let’s have a look at how to subscribe to topics

Subscribing to Topics

After creating a consumer, the next step is to subscribe to one or more topics

```
consumer.subscribe(Collections.singletonList(“StudentData"));
```

subcribe() method takes a list of topics as a parameter

Its possible to call subscribe with a regular expression

The expression can match multiple topic names

This is useful for applications that need to consume from multiple topics

```
consumer.subscribe("test.*");
```

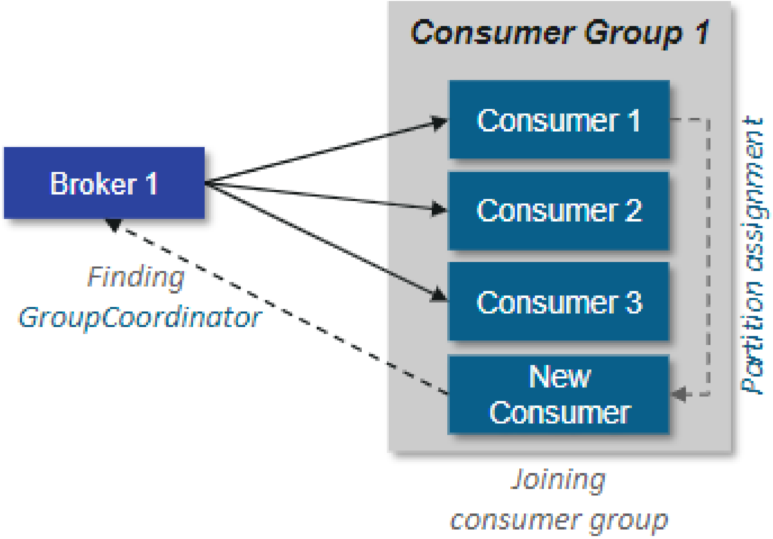
☛ *If a new topic is created which fulfils the condition, a immediate rebalance will take place and the consumers will start consuming*

The “Poll Loop”

The Poll Loop

The first time you call *poll()* with a new consumer, it is responsible for finding the *GroupCoordinator*, joining the consumer group, and receiving a partition assignment

- If a rebalance is triggered, it will be handled inside the poll loop as well
- *Heartbeats* are sent within the poll loop (version 0.10.0 and earlier)

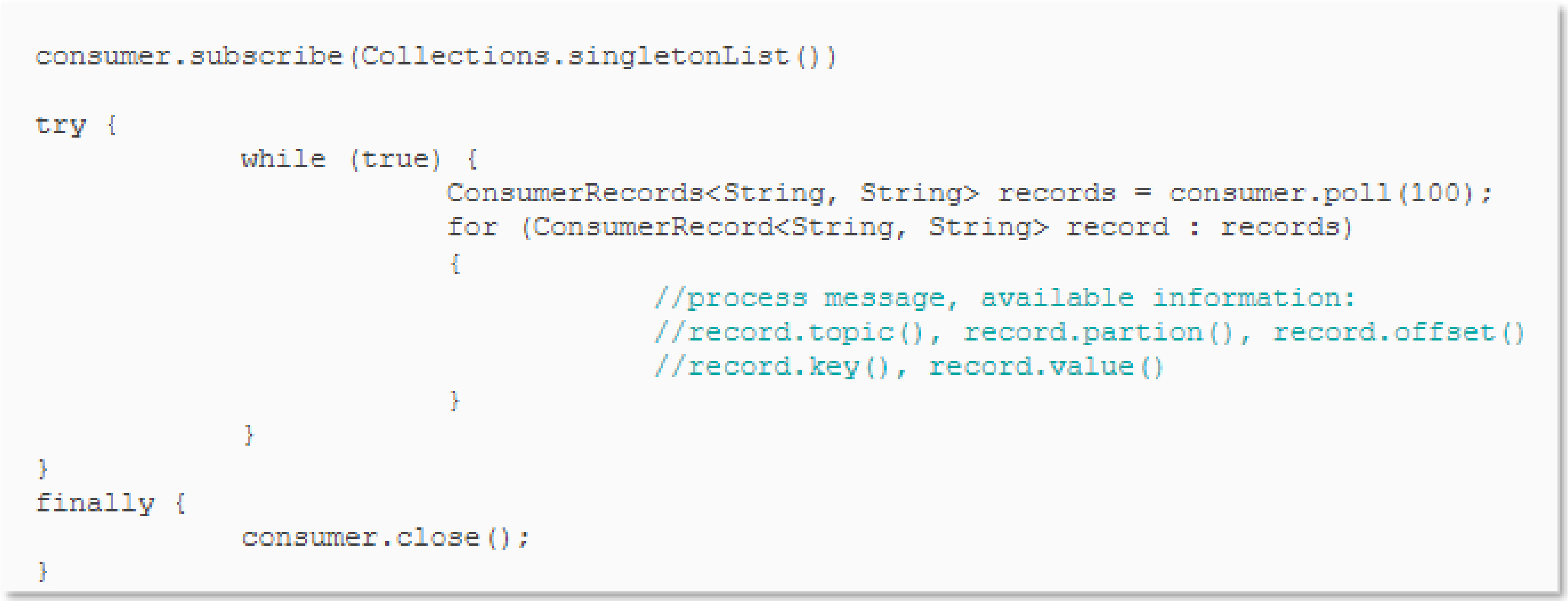


☛ *We need to make sure that whatever processing we do between iterations is fast and efficient*

The Poll Loop

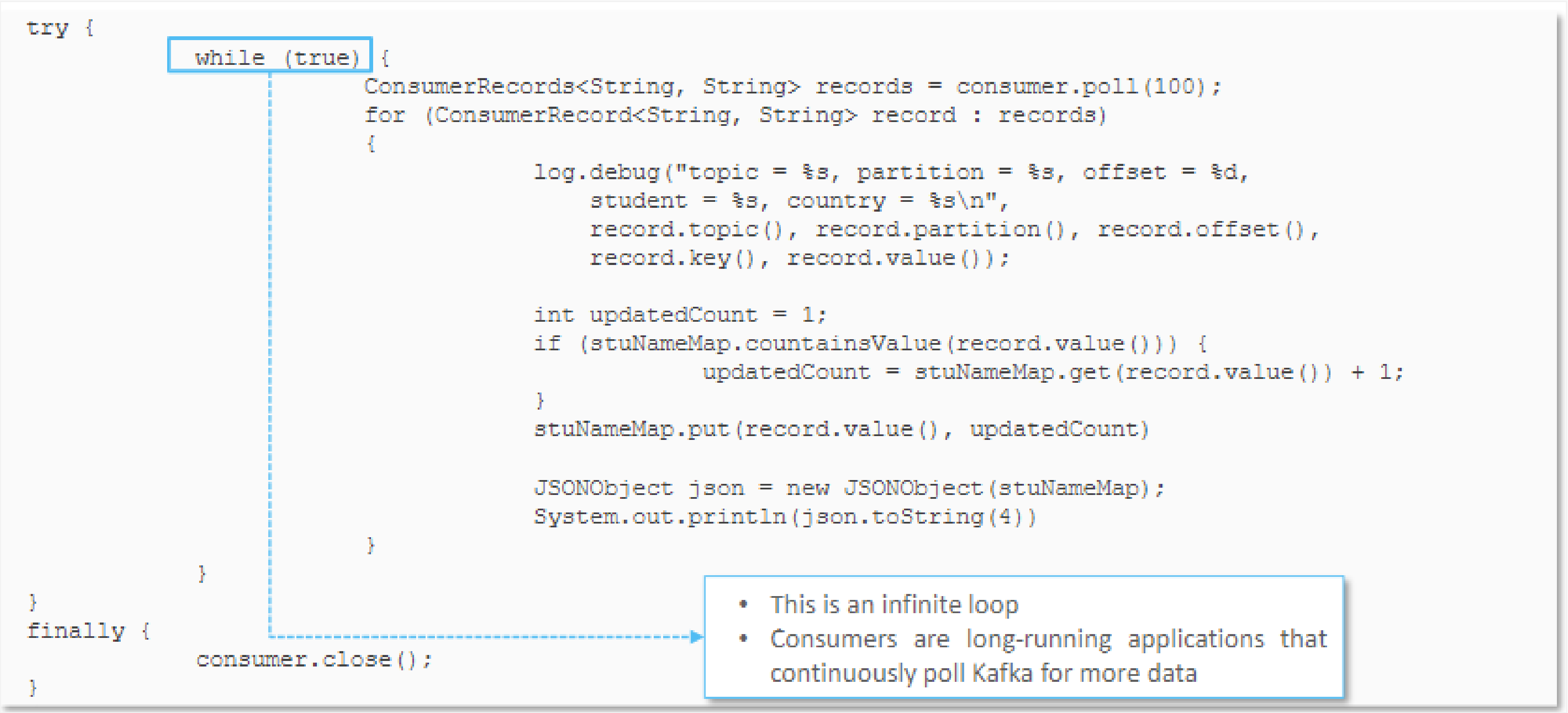
Once the consumer subscribes to topics, the poll loop handles all details of coordination, partition rebalances, heartbeats, and data fetching

It gives a API that returns available data from the assigned partitions



```
consumer.subscribe(Collections.singletonList())

try {
          while (true) {
                    ConsumerRecords<String, String> records = consumer.poll(100);
                    for (ConsumerRecord<String, String> record : records)
                    {
                              //process message, available information:
                              //record.topic(), record.partion(), record.offset()
                              //record.key(), record.value()
                    }
          }
}
finally {
          consumer.close();
}
```

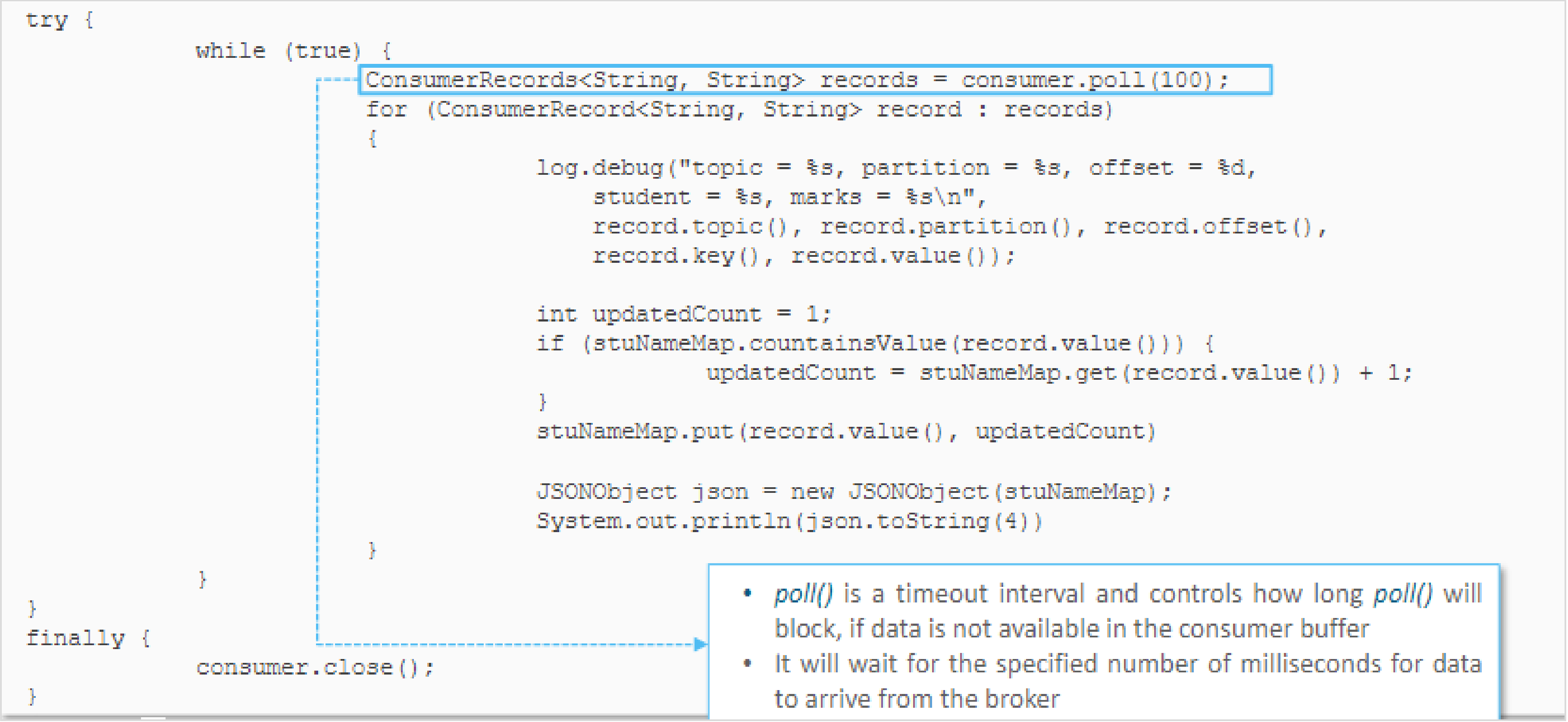
The Poll Loop



```
try {
        while (true) {
                ConsumerRecords<String, String> records = consumer.poll(100);
                for (ConsumerRecord<String, String> record : records)
                {
                        log.debug("topic = %s, partition = %s, offset = %d,
                           student = %s, country = %s\n",
                           record.topic(), record.partition(), record.offset(),
                           record.key(), record.value());

                        int updatedCount = 1;
                        if (stuNameMap.countainsValue(record.value())) {
                                updatedCount = stuNameMap.get(record.value()) + 1;
                        }
                        stuNameMap.put(record.value(), updatedCount)

                        JSONObject json = new JSONObject(stuNameMap);
                        System.out.println(json.toString(4))
                }
        }
}
finally {
        consumer.close();
}
```

- This is an infinite loop
- Consumers are long-running applications that continuously poll Kafka for more data

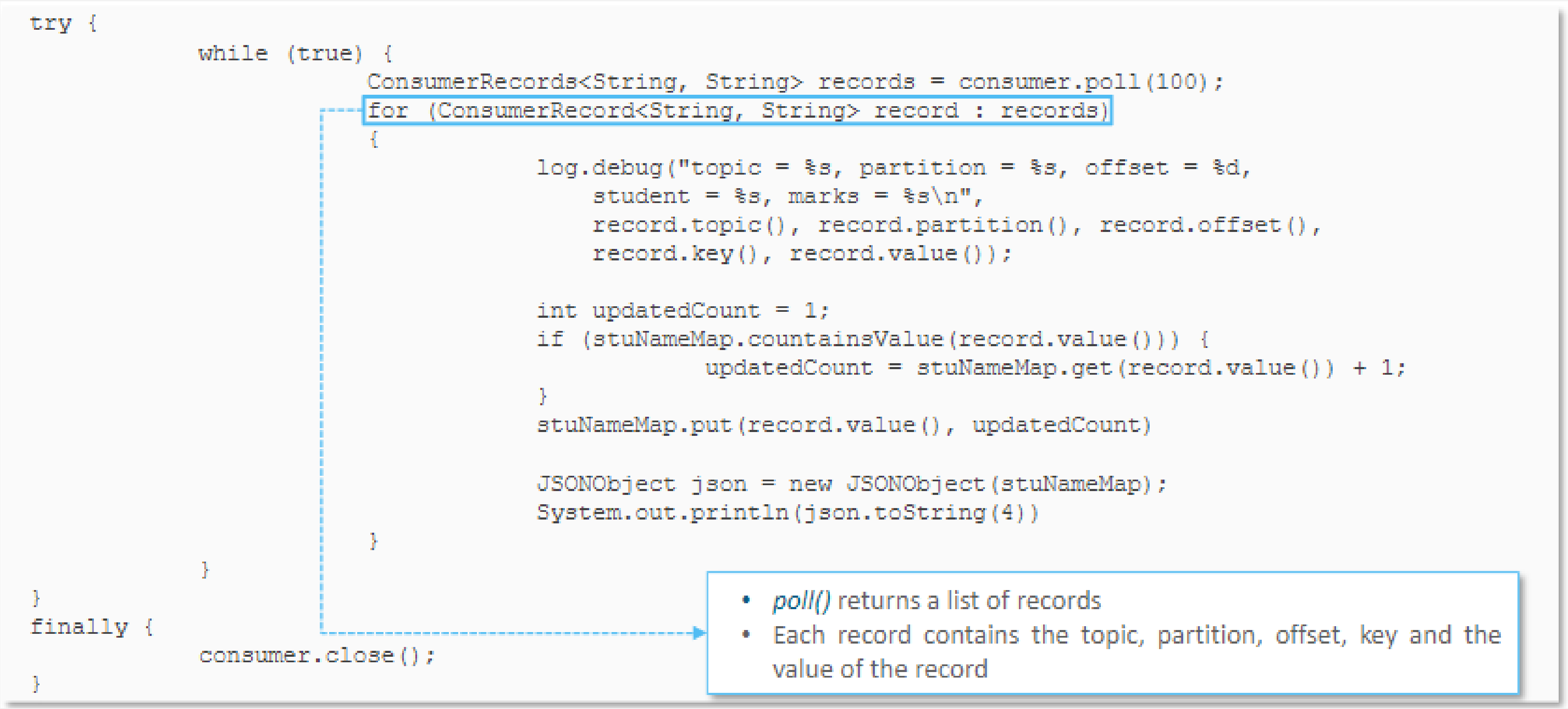
The Poll Loop



```
try {
            while (true) {
                        ConsumerRecords<String, String> records = consumer.poll(100);
                        for (ConsumerRecord<String, String> record : records)
                        {
                                    log.debug("topic = %s, partition = %s, offset = %d,
                                        student = %s, marks = %s\n",
                                        record.topic(), record.partition(), record.offset(),
                                        record.key(), record.value());

                                    int updatedCount = 1;
                                    if (stuNameMap.countainsValue(record.value())) {
                                                updatedCount = stuNameMap.get(record.value()) + 1;
                                    }
                                    stuNameMap.put(record.value(), updatedCount)

                                    JSONObject json = new JSONObject(stuNameMap);
                                    System.out.println(json.toString(4))
                        }
            }
}
finally {
            consumer.close();
}
```

- *poll()* is a timeout interval and controls how long *poll()* will block, if data is not available in the consumer buffer
- It will wait for the specified number of milliseconds for data to arrive from the broker

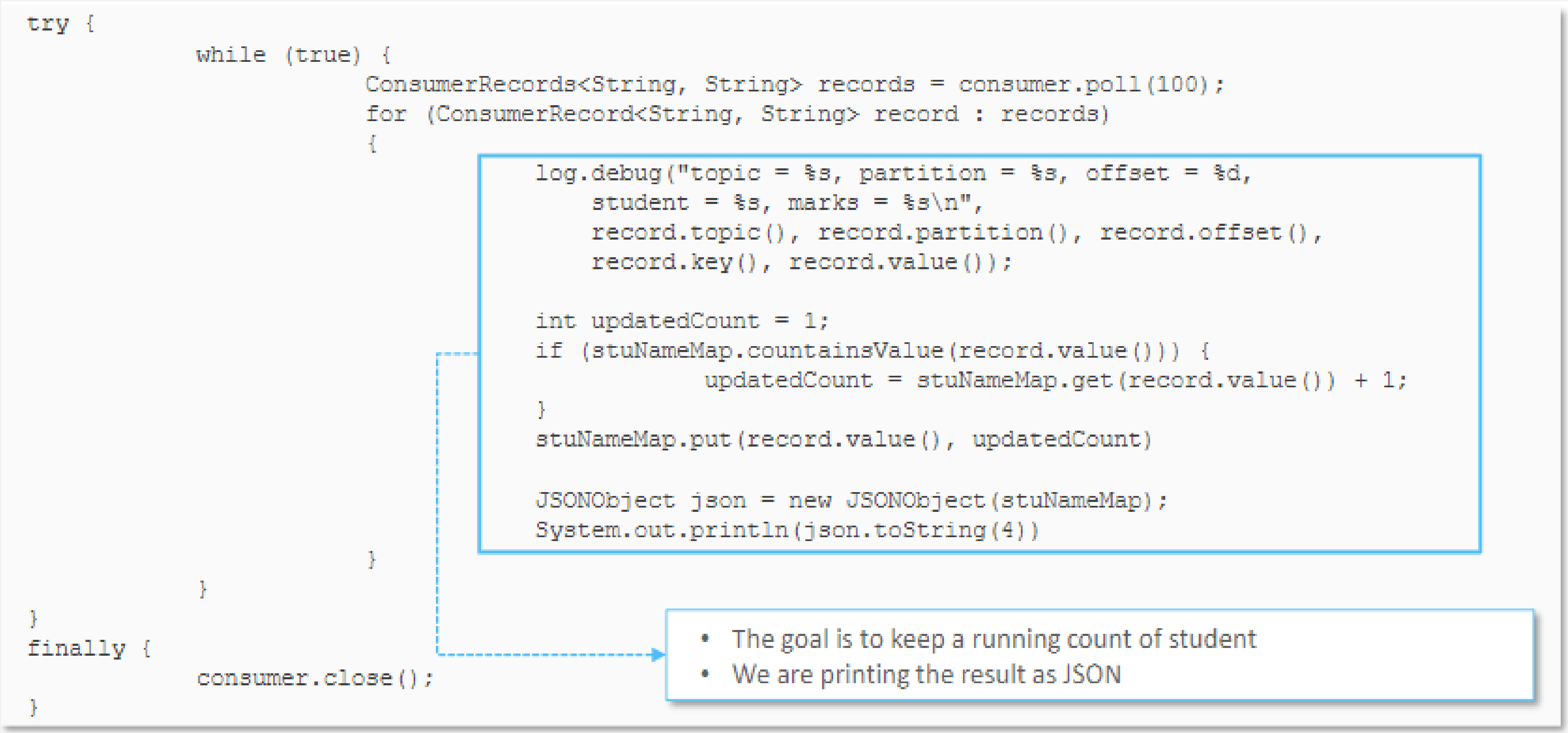
The Poll Loop



```
try {
            while (true) {
                        ConsumerRecords<String, String> records = consumer.poll(100);
                        for (ConsumerRecord<String, String> record : records)
                        {
                                    log.debug("topic = %s, partition = %s, offset = %d,
                                        student = %s, marks = %s\n",
                                        record.topic(), record.partition(), record.offset(),
                                        record.key(), record.value());

                                    int updatedCount = 1;
                                    if (stuNameMap.countainsValue(record.value())) {
                                                updatedCount = stuNameMap.get(record.value()) + 1;
                                    }
                                    stuNameMap.put(record.value(), updatedCount)

                                    JSONObject json = new JSONObject(stuNameMap);
                                    System.out.println(json.toString(4))
                        }
            }
}
finally {
            consumer.close();
}
```

- *poll()* returns a list of records
- Each record contains the topic, partition, offset, key and the value of the record

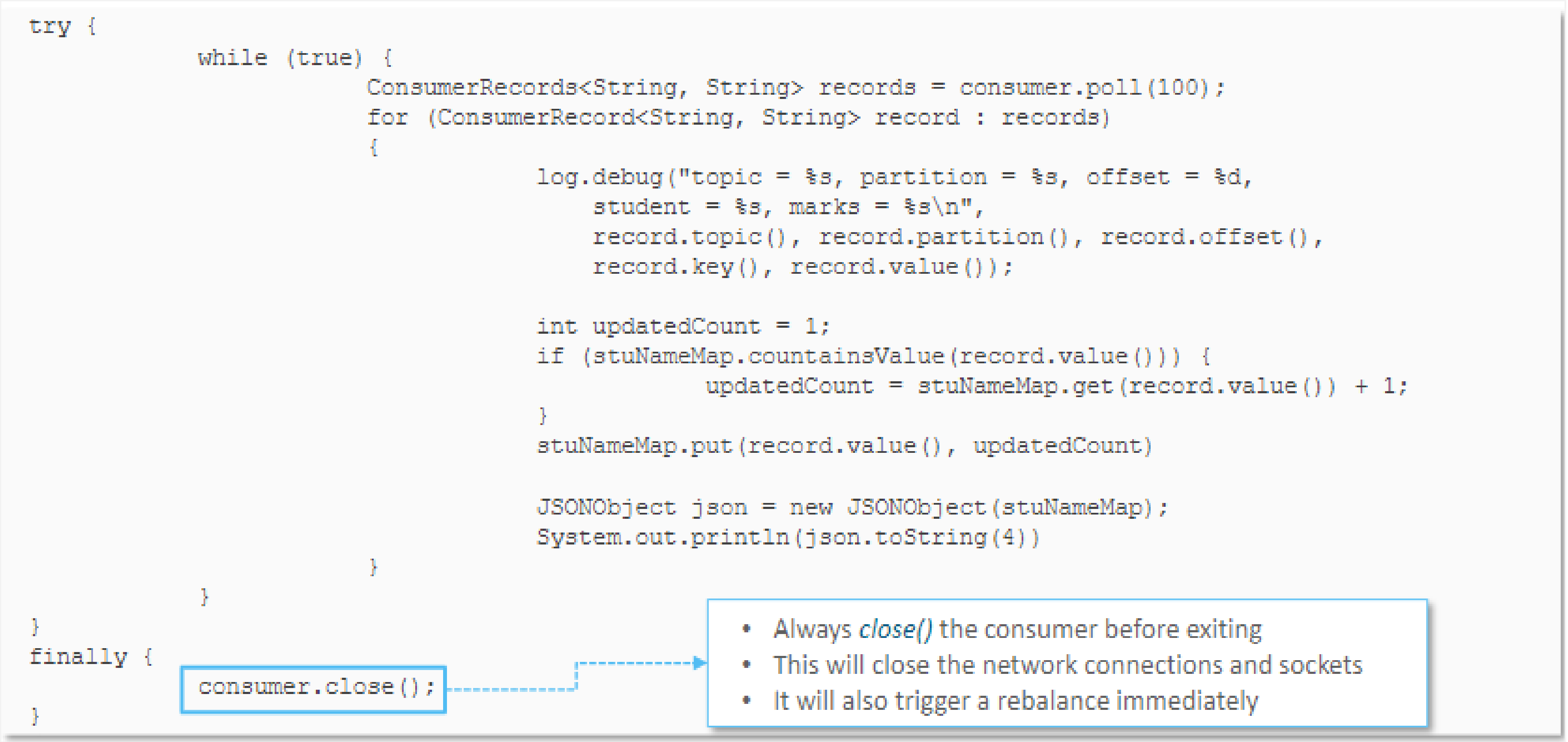
The Poll Loop



```
try {
            while (true) {
                        ConsumerRecords<String, String> records = consumer.poll(100);
                        for (ConsumerRecord<String, String> record : records)
                        {
                                    log.debug("topic = %s, partition = %s, offset = %d,
                                        student = %s, marks = %s\n",
                                        record.topic(), record.partition(), record.offset(),
                                        record.key(), record.value());

                                    int updatedCount = 1;
                                    if (stuNameMap.countainsValue(record.value())) {
                                                updatedCount = stuNameMap.get(record.value()) + 1;
                                    }
                                    stuNameMap.put(record.value(), updatedCount)

                                    JSONObject json = new JSONObject(stuNameMap);
                                    System.out.println(json.toString(4))
                        }
            }
}
finally {
            consumer.close();
}
```

- The goal is to keep a running count of student
- We are printing the result as JSON

The Poll Loop

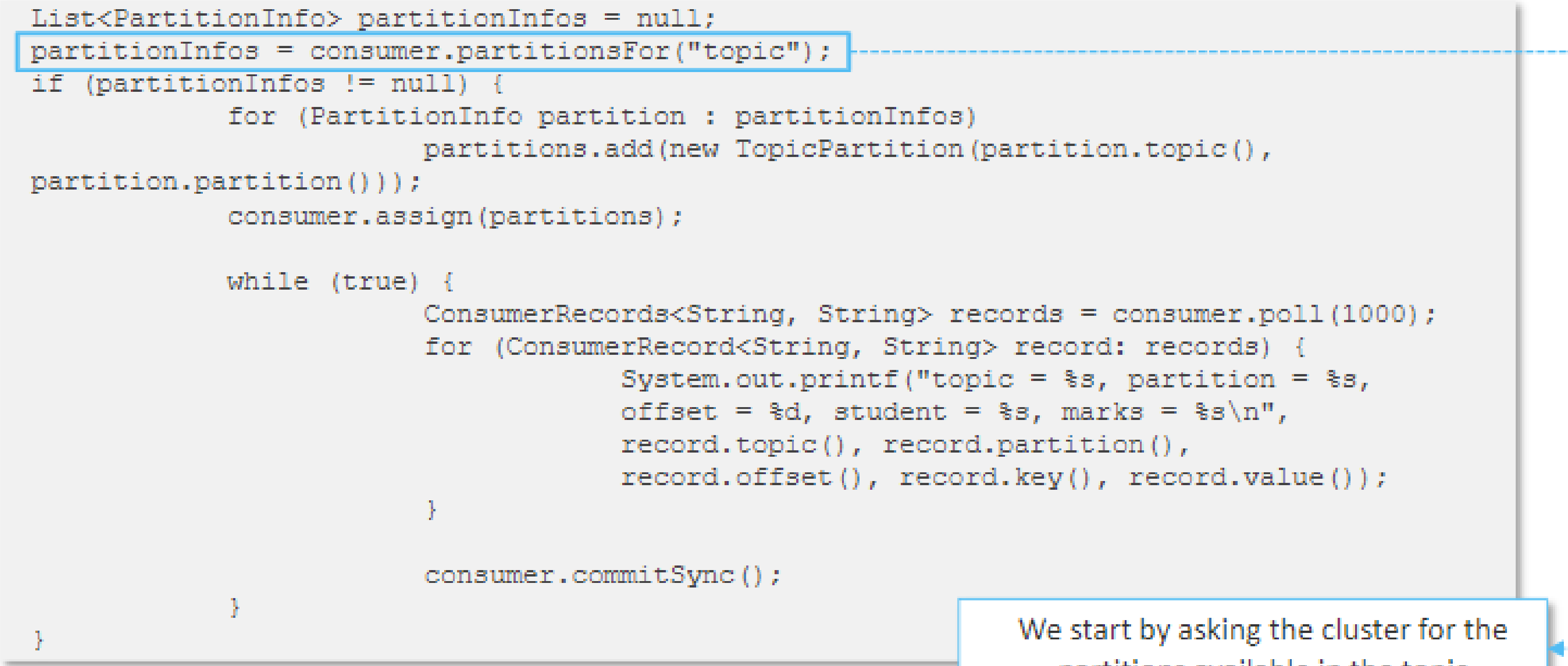


```
try {
        while (true) {
                ConsumerRecords<String, String> records = consumer.poll(100);
                for (ConsumerRecord<String, String> record : records)
                {
                        log.debug("topic = %s, partition = %s, offset = %d,
                           student = %s, marks = %s\n",
                           record.topic(), record.partition(), record.offset(),
                           record.key(), record.value());

                        int updatedCount = 1;
                        if (stuNameMap.countainsValue(record.value())) {
                                updatedCount = stuNameMap.get(record.value()) + 1;
                        }
                        stuNameMap.put(record.value(), updatedCount)

                        JSONObject json = new JSONObject(stuNameMap);
                        System.out.println(json.toString(4))
                }
        }
}
finally {
        consumer.close();
}
```

- Always *close()* the consumer before exiting
- This will close the network connections and sockets
- It will also trigger a rebalance immediately

Let’s understand
How to create a standalone Consumer

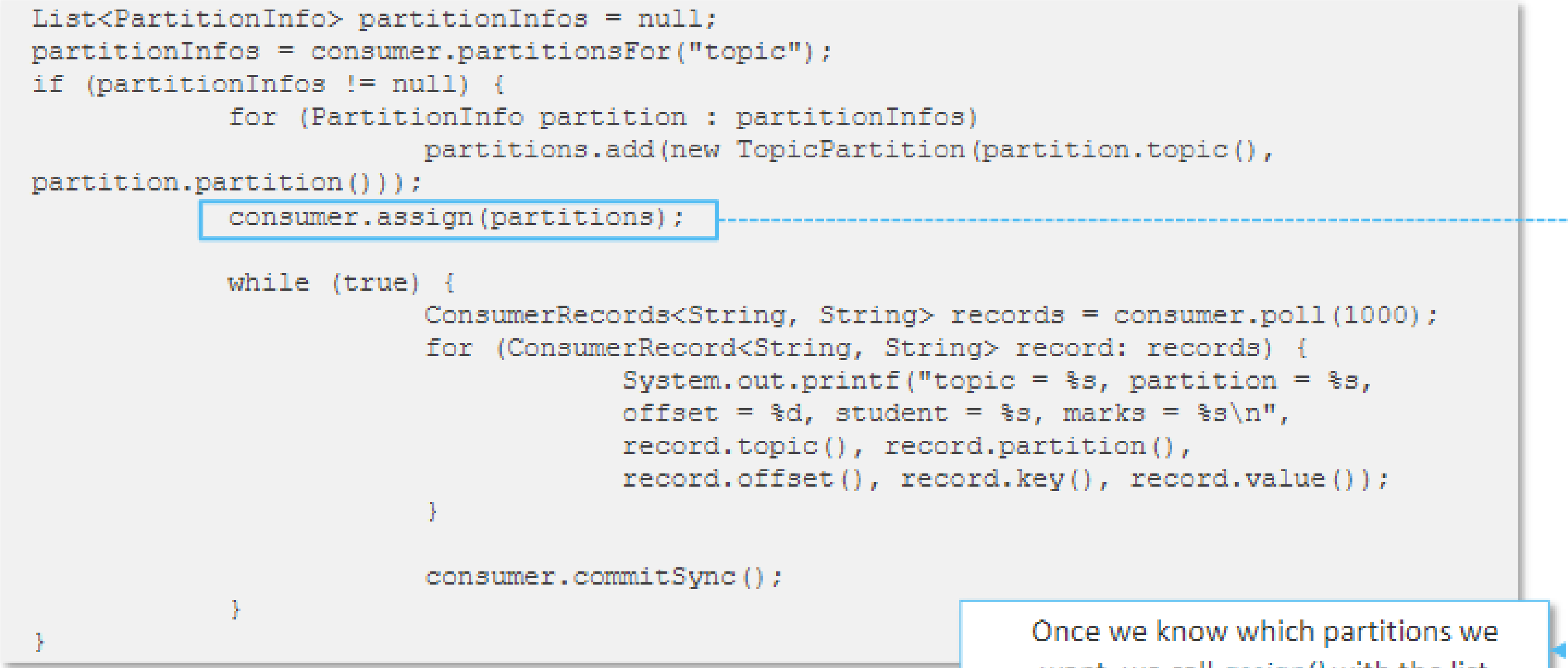
Standalone Consumer



```
List<PartitionInfo> partitionInfos = null;
partitionInfos = consumer.partitionsFor("topic");
if (partitionInfos != null) {
            for (PartitionInfo partition : partitionInfos)
                        partitions.add(new TopicPartition(partition.topic(),
partition.partition()));
            consumer.assign(partitions);

            while (true) {
                        ConsumerRecords<String, String> records = consumer.poll(1000);
                        for (ConsumerRecord<String, String> record: records) {
                                    System.out.printf("topic = %s, partition = %s,
                                    offset = %d, student = %s, marks = %s\n",
                                    record.topic(), record.partition(),
                                    record.offset(), record.key(), record.value());
                        }

                        consumer.commitSync();
            }
}
```

We start by asking the cluster for the partitions available in the topic

Standalone Consumer



```
List<PartitionInfo> partitionInfos = null;
partitionInfos = consumer.partitionsFor("topic");
if (partitionInfos != null) {
            for (PartitionInfo partition : partitionInfos)
                        partitions.add(new TopicPartition(partition.topic(),
partition.partition()));
            consumer.assign(partitions);

            while (true) {
                        ConsumerRecords<String, String> records = consumer.poll(1000);
                        for (ConsumerRecord<String, String> record: records) {
                                    System.out.printf("topic = %s, partition = %s,
                                    offset = %d, student = %s, marks = %s\n",
                                    record.topic(), record.partition(),
                                    record.offset(), record.key(), record.value());
                        }

                        consumer.commitSync();
            }
}
```

Once we know which partitions we want, we call *assign()* with the list

Let's look at some of the important Consumer Configurations

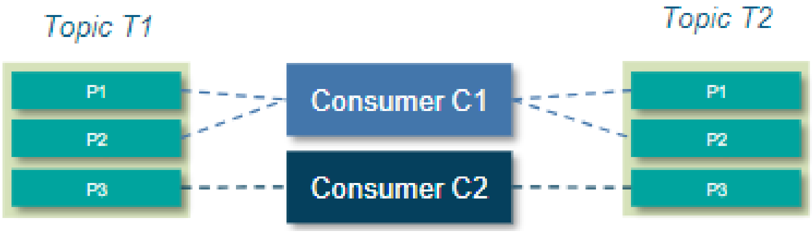
Kafka Consumer Configurations

partition.assignment.strategy

- A *PartitionAssignor* is a class that, the consumers and topics are subscribed to. It decides which partitions will be assigned to which consumer. By default, Kafka has two assignment strategies:

Range

Assigns to each consumer a consecutive subset of partitions from each topic it subscribes to.



Topic T1

P1

P2

P3

Consumer C1

Consumer C2

Topic T2

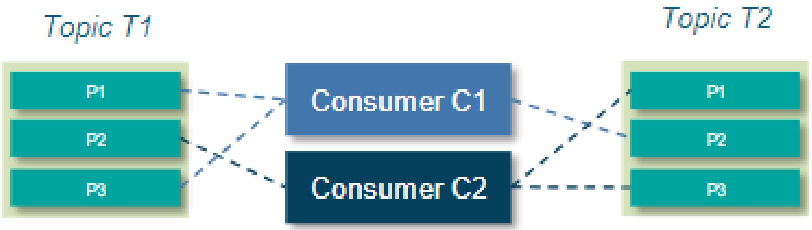
P1

P2

P3

RoundRobin

Takes all the partitions from all subscribed topics and assigns them to consumers sequentially, one by one.



Topic T1

P1

P2

P3

Consumer C1

Consumer C2

Topic T2

P1

P2

P3

Kafka Consumer Configurations

fetch.min.bytes

- This *property allows* a consumer to *specify the minimum amount of data* that it wants to receive from the broker when fetching records

fetch.max.wait.ms

- The maximum amount of time the server will block before answering the fetch request if there isn't sufficient data to immediately satisfy the requirement given by *fetch.min.bytes*

max.partition.fetch.bytes

- This *property controls* the *maximum number of bytes* the server will return per partition. The default is 1 MB

Kafka Consumer Configurations

auto.offset.reset

- *This property controls the behavior of the consumer* when it starts reading a partition for which it doesn't have a committed offset or if the committed offset it has is invalid.

enable.auto.commit

- This parameter controls whether the *consumer will commit offsets automatically*, and defaults to true

session.timeout.ms

- The *amount of time a consumer can be out of contact* with the brokers while still considered alive defaults to 3 seconds.

Kafka Consumer Configurations

receive.buffer.bytes & send.buffer.bytes

- These are the *sizes of the TCP send and receive buffers* used by the *sockets* when *writing* and *reading data*. If these are set to -1, the OS defaults will be used.

max.poll.records

- This *controls the maximum number of records that a single call to poll() will return*. This is useful to help control the amount of data your application will need to process in the polling loop.

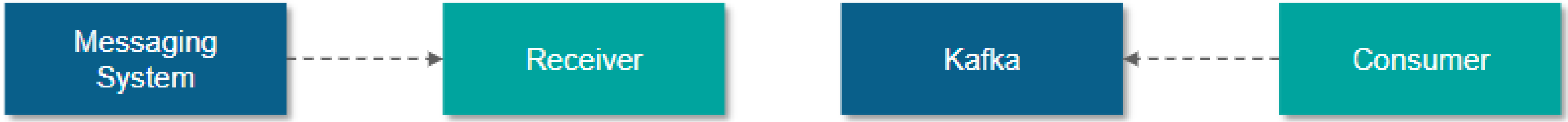
client.id

- This can be any string, and will be *used by the brokers to identify messages sent from the client*. It is used in logging and metrics, and for quotas.

Let’s talk about Commits and Offsets

Commits and Offsets

Whenever we *poll()*, it return records written to Kafka that consumers in our group have not read yet
We have a way of tracking which records were read by a consumer


Messaging System

Receiver

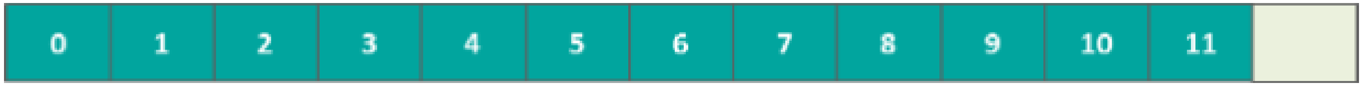
Kafka

Consumer

JMS keeps track of received messages

Consumer keeps the track of received messages

- Kafka's unique characteristics is that it does not track acknowledgments like JMS queues
- It *allows consumers* to *use Kafka* to *track their position* (offset) in each *partition*
- The *action of updating* the current position in the partition is called a *commit*



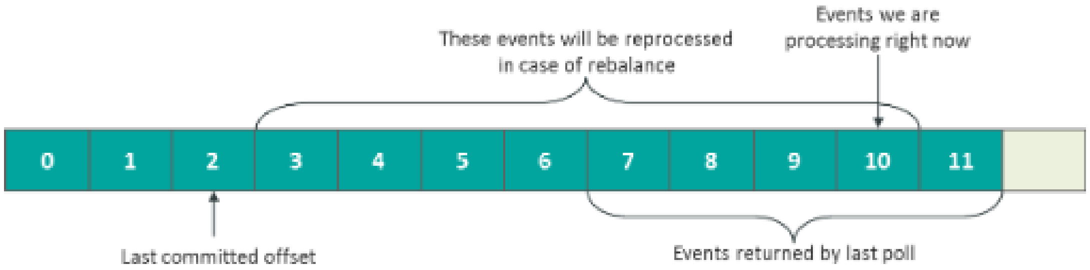
Messages in queue to be commited

How does a Consumer Commit an Offset?

Consumer sends a message to Kafka, to a special consumer_offsets topic, with the committed offset for each partition

If a consumer crashes or a new consumer joins, this will trigger a rebalance

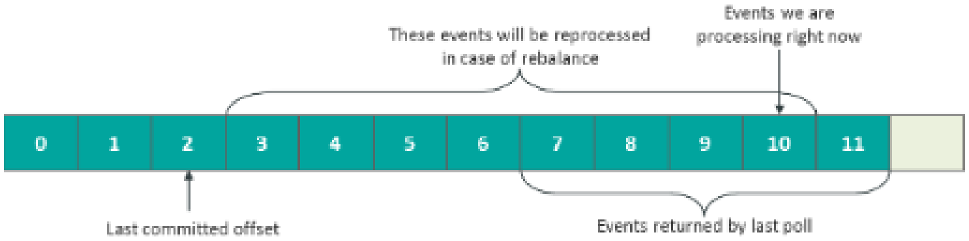
To know where to start, the consumer will read the latest committed offset of each partition and continue from there

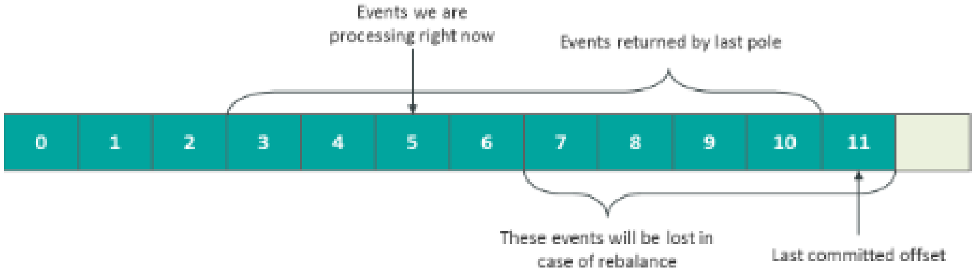

Events we are processing right now
These events will be reprocessed in case of rebalance
0
1
2
3
4
5
6
7
8
9
10
11
Last committed offset
Events returned by last poll

☛ *If the committed offset is smaller than the offset of the last message the client processed, the messages between the last processed offset and the committed offset will be processed twice*

How does a Consumer Commit an Offset?

If the committed offset is larger than the offset of the last message, all messages between the last processed offset and the committed offset will be missed by the consumer group

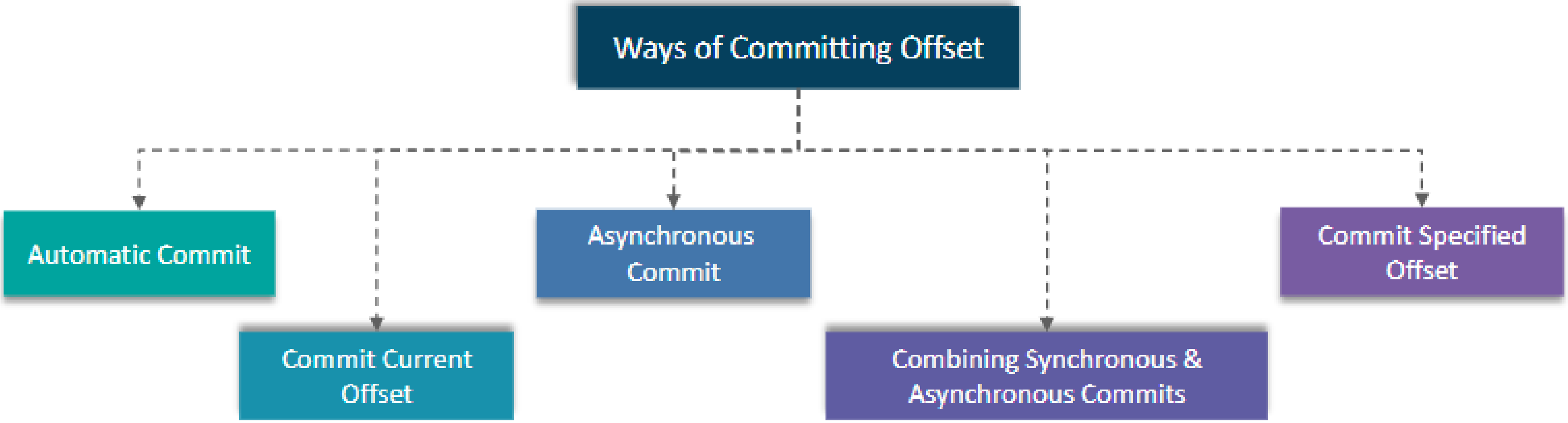

These events will be reprocessed in case of rebalance
Events we are processing right now
0
1
2
3
4
5
6
7
8
9
10
11
Last committed offset
Events returned by last poll


Events we are processing right now
Events returned by last pole
0
1
2
3
4
5
6
7
8
9
10
11
These events will be lost in case of rebalance
Last committed offset

Let’s understand the ways of Committing Offset

Ways of Committing Offset

Managing offsets has a big impact on the client application, KafkaConsumer API provides multiple ways of committing offsets:



Automatic Commit

Automatic Commit
Commit Current Offset
Asynchronous Commit
Combining Sync & Async Commits
Commit Specified Offset

If you configure *enable.auto.commit*=true, then every five seconds the consumer will commit the largest offset your client received from *poll()*

- It can be controlled by setting *auto.commit.interval.ms* parameter
- The automatic commits are driven by the *poll loop*
- Whenever you poll, the consumer checks if it is time to commit, and if it is, it will commit the offsets it returned in the last poll
- *close()* also commits offsets automatically

Commit Current Offset

Automatic Commit

Commit Current Offset

Asynchronous Commit

Combining Sync & Async Commits

Commit Specified Offset

By setting *auto.commit.offset*=false, offsets will only be committed when the application explicitly chooses to do so

- It eliminates the possibility of missing messages and reduce the number of messages duplicated during rebalancing
- The consumer API has the option of committing the current offset at any point
- The simplest and most reliable of the commit APIs is *commitSync()*
- API will commit the latest offset returned by *poll()* & return when the offset is committed
- Throws an exception if commit fails for some reason

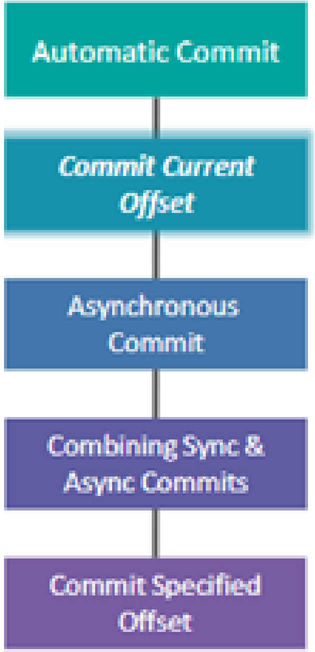
Commit Current Offset

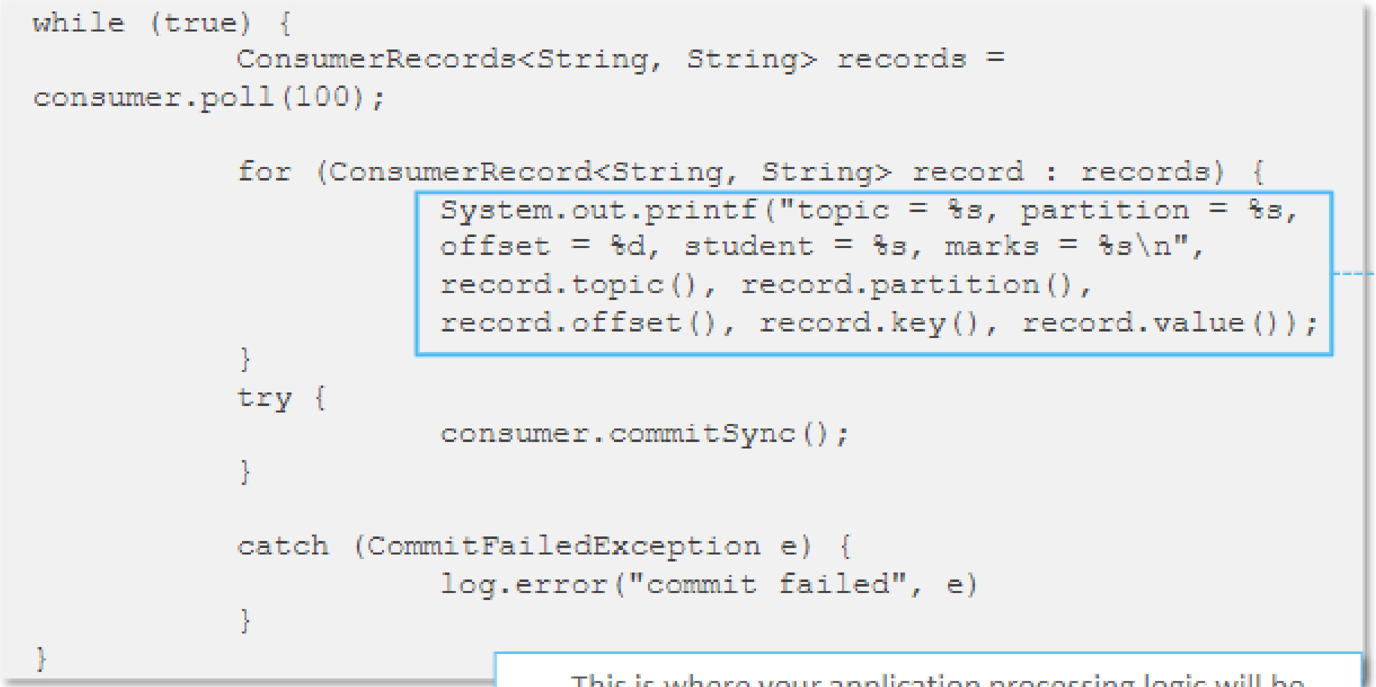
Automatic Commit
Commit Current Offset
Asynchronous Commit
Combining Sync & Async Commits
Commit Specified Offset

By setting *auto.commit.offset*=false, offsets will only be committed when the application explicitly chooses to do so

- *commitSync()* will commit the latest offset returned by *poll()*
- Call *commitSync()* after you are done processing all the records in the collection
- Otherwise, you risk missing messages.
- When rebalance is triggered, all the messages from the beginning of the most recent batch until the time of the rebalance will be processed twice.

Commit Current Offset

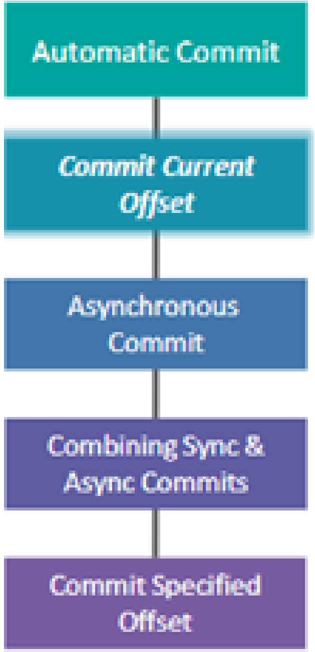


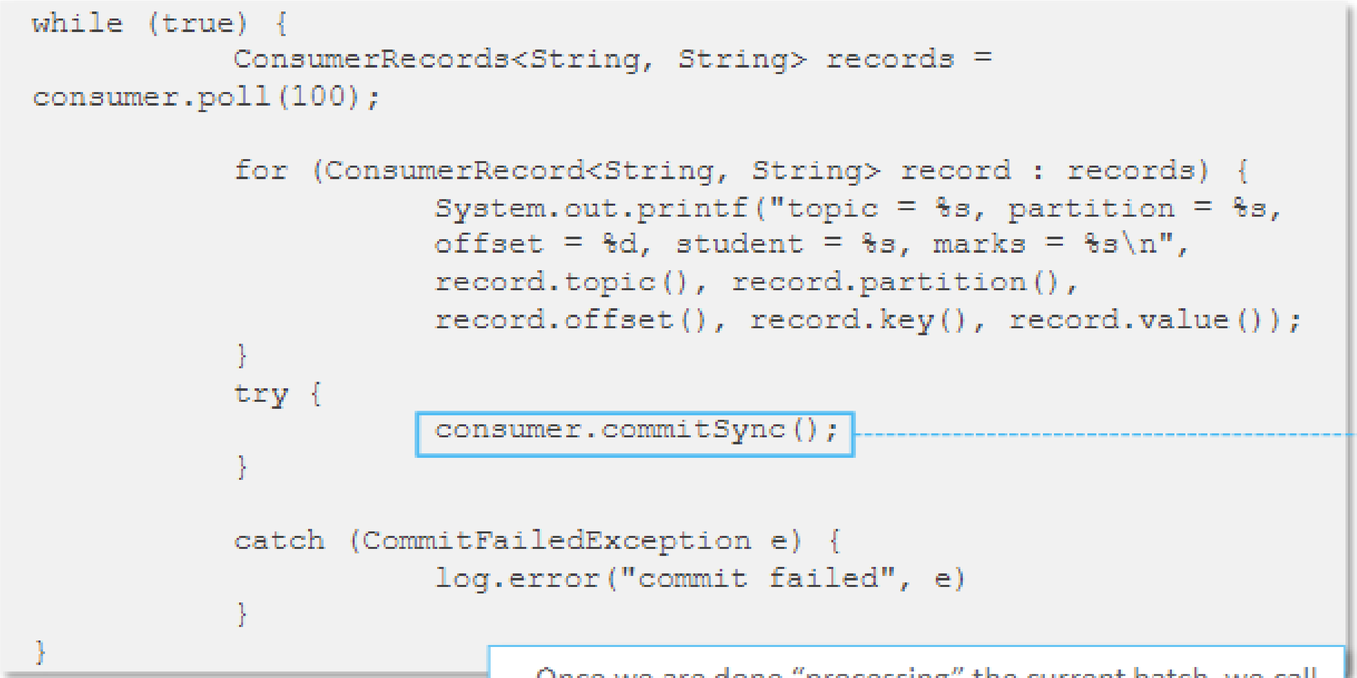


```
while (true) {
          ConsumerRecords<String, String> records =
consumer.poll(100);

          for (ConsumerRecord<String, String> record : records) {
                    System.out.printf("topic = %s, partition = %s,
                    offset = %d, student = %s, marks = %s\n",
                    record.topic(), record.partition(),
                    record.offset(), record.key(), record.value());
          }
          try {
                    consumer.commitSync();
          }

          catch (CommitFailedException e) {
                    log.error("commit failed", e)
          }
}
```

This is where your application processing logic will be applied, your application will do a lot more – modify, enrich, aggregate, display etc.

Commit Current Offset

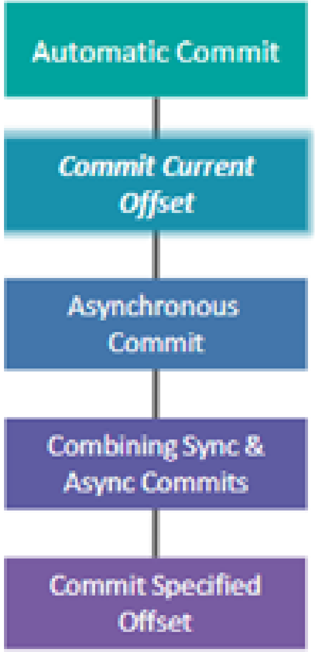


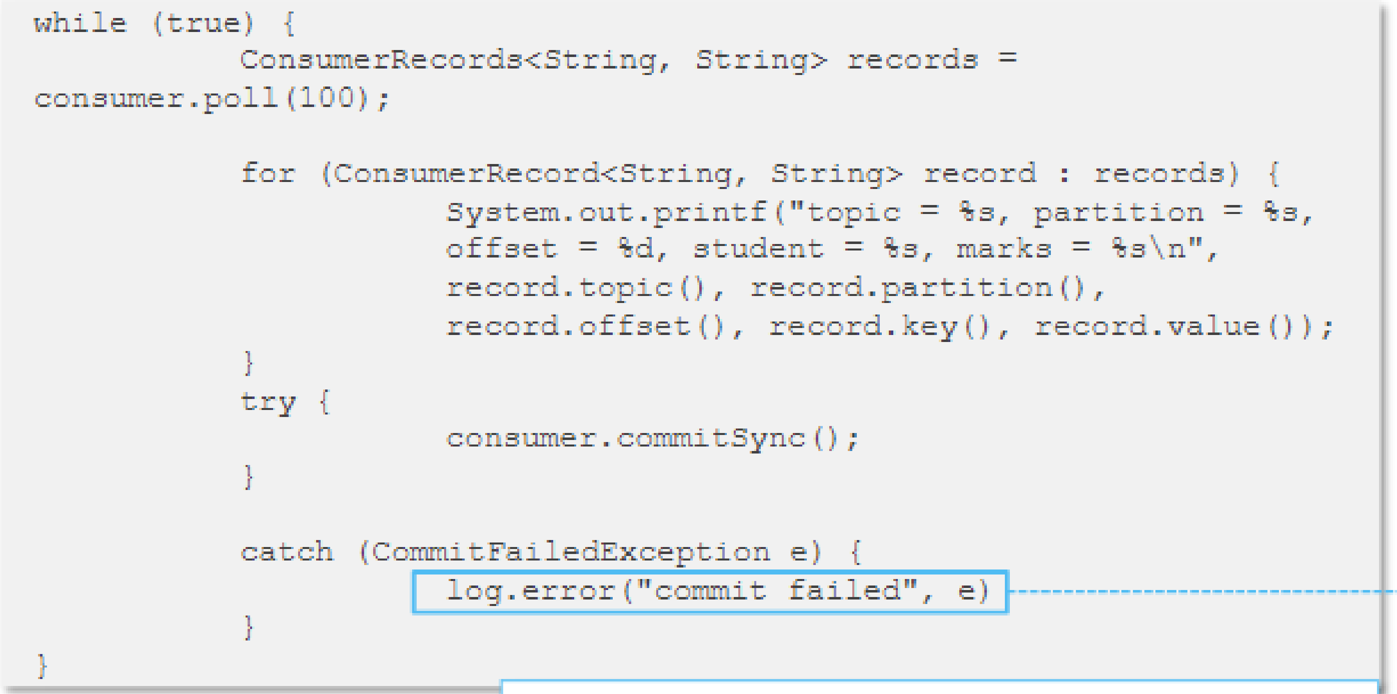


```
while (true) {
            ConsumerRecords<String, String> records =
consumer.poll(100);

            for (ConsumerRecord<String, String> record : records) {
                        System.out.printf("topic = %s, partition = %s,
                        offset = %d, student = %s, marks = %s\n",
                        record.topic(), record.partition(),
                        record.offset(), record.key(), record.value());
            }
            try {
                        consumer.commitSync();
            }

            catch (CommitFailedException e) {
                        log.error("commit failed", e)
            }
}
```

Once we are done “processing” the current batch, we call *commitSync* to commit the last offset in the batch, before polling for additional messages

Commit Current Offset





```
while (true) {
        ConsumerRecords<String, String> records =
consumer.poll(100);

        for (ConsumerRecord<String, String> record : records) {
                System.out.printf("topic = %s, partition = %s,
                offset = %d, student = %s, marks = %s\n",
                record.topic(), record.partition(),
                record.offset(), record.key(), record.value());
        }
        try {
                consumer.commitSync();
        }

        catch (CommitFailedException e) {
                log.error("commit failed", e)
        }
}
```

commitSync retries committing as long as there is no error which can't be recovered, if this happens, we log the error

Asynchronous Commit

Automatic Commit

Commit Current Offset

Asynchronous Commit

Combining Sync & Async Commits

Commit Specified Offset

Asynchronous commit API, instead of waiting for the broker to respond to a commit, it just sends the request and continue.

- Drawback of manual commit is that the application is blocked until the broker responds
- This limits the throughput of the application
- Throughput can be improved by less frequent commits, but it increases potential duplicates

Asynchronous Commit

Automatic Commit

Commit Current Offset

Asynchronous Commit

Combining Sync & Async Commits

Commit Specified Offset

commitAsync() will not retry, as by the time it receives a response, there may have been a later successful commits

- The request to commit offset 2000 failed due to temporary communication problem & broker never gets the request.
- Meanwhile, we processed another batch successfully.
- If commitAsync() now retries the previously failed commit, it might cause more duplicates

Asynchronous Commit

Automatic Commit

Commit Current Offset

Asynchronous Commit

Combining Sync & Async Commits

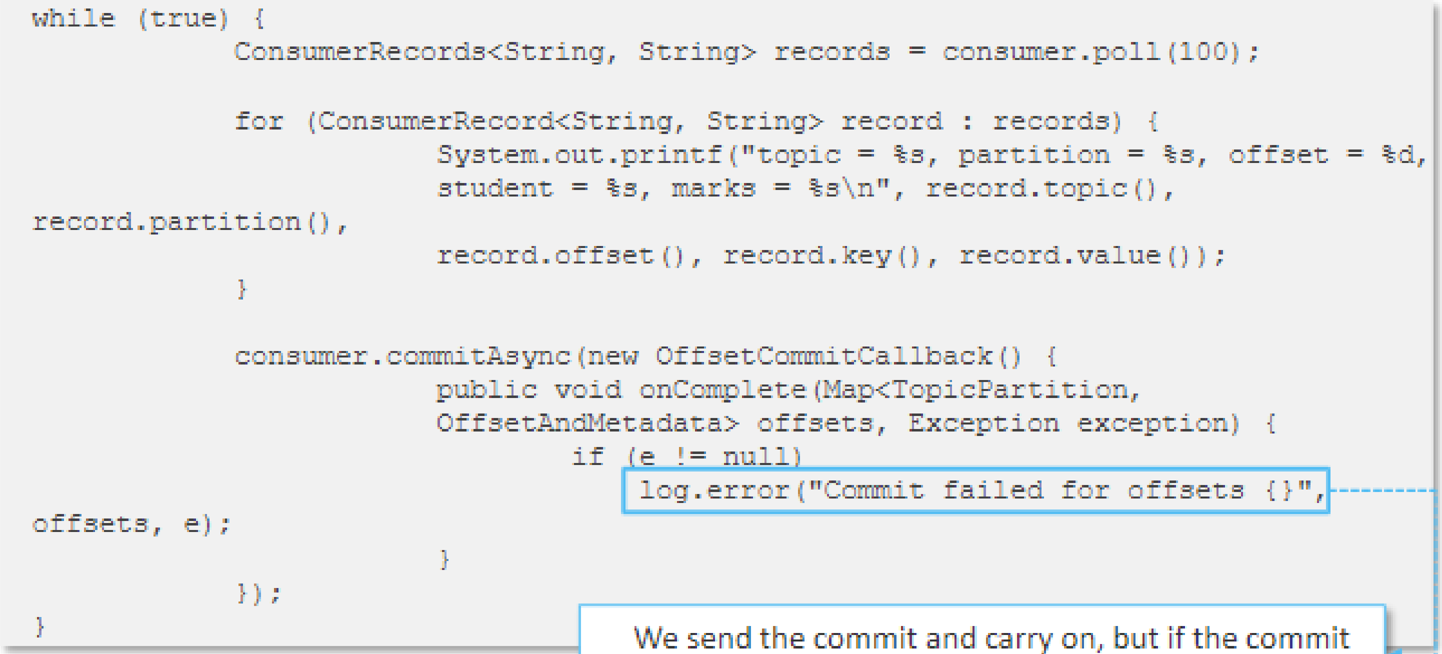
Commit Specified Offset

commitAsync() will not retry, as by the time it receives a response, there may have been a later successful commits

- commitAsync() provides an option to pass in a callback that will be triggered when the broker responds
- It is common to use the callback to log commit errors or to count them in a metric.
- If you want to use the callback for retries, you need to be aware of the problem with commit order

Asynchronous Commit

Automatic Commit
Commit Current Offset
Asynchronous Commit
Combining Sync & Async Commits
Commit Specified Offset



```
while (true) {
            ConsumerRecords<String, String> records = consumer.poll(100);

            for (ConsumerRecord<String, String> record : records) {
                        System.out.printf("topic = %s, partition = %s, offset = %d,
                        student = %s, marks = %s\n", record.topic(),
record.partition(),
                        record.offset(), record.key(), record.value());
            }

            consumer.commitAsync(new OffsetCommitCallback() {
                        public void onComplete(Map<TopicPartition,
                        OffsetAndMetadata> offsets, Exception exception) {
                                if (e != null)
                                   log.error("Commit failed for offsets {}",
offsets, e);
                        }
            });
}
```

We send the commit and carry on, but if the commit fails, the failure and the offsets will be logged

Combining Synchronous & Asynchronous Commits

Automatic Commit

Commit Current Offset

Asynchronous Commit

Combining Sync & Async Commits

Commit Specified Offset

A common pattern is to combine *commitAsync()* with *commitSync()* just before shutdown

- If the problem is temporary, the commit will be successful
- But if we know that this is the last commit before a rebalance or closing the consumer, we need to make extra sure that the commit succeeds

Combining Synchronous & Asynchronous Commits

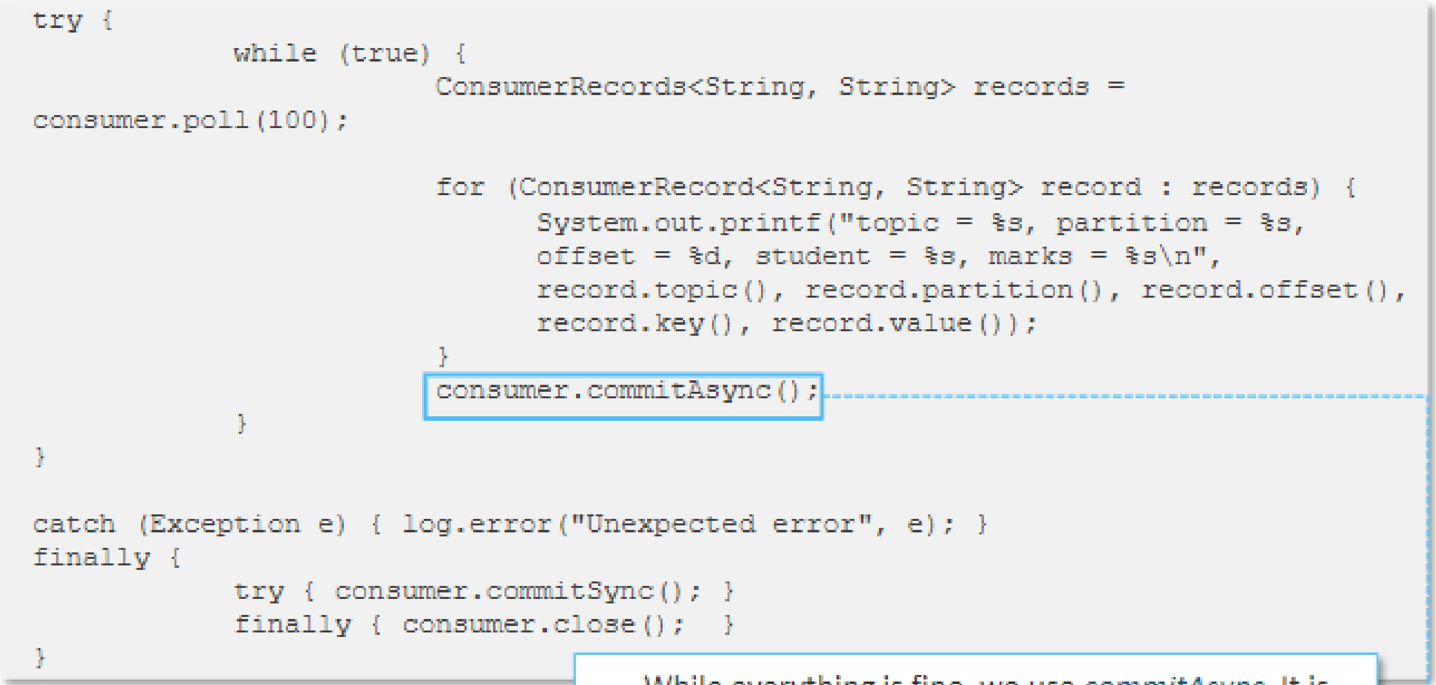
Automatic Commit

Commit Current Offset

Asynchronous Commit

Combining Sync & Async Commits

Commit Specified Offset

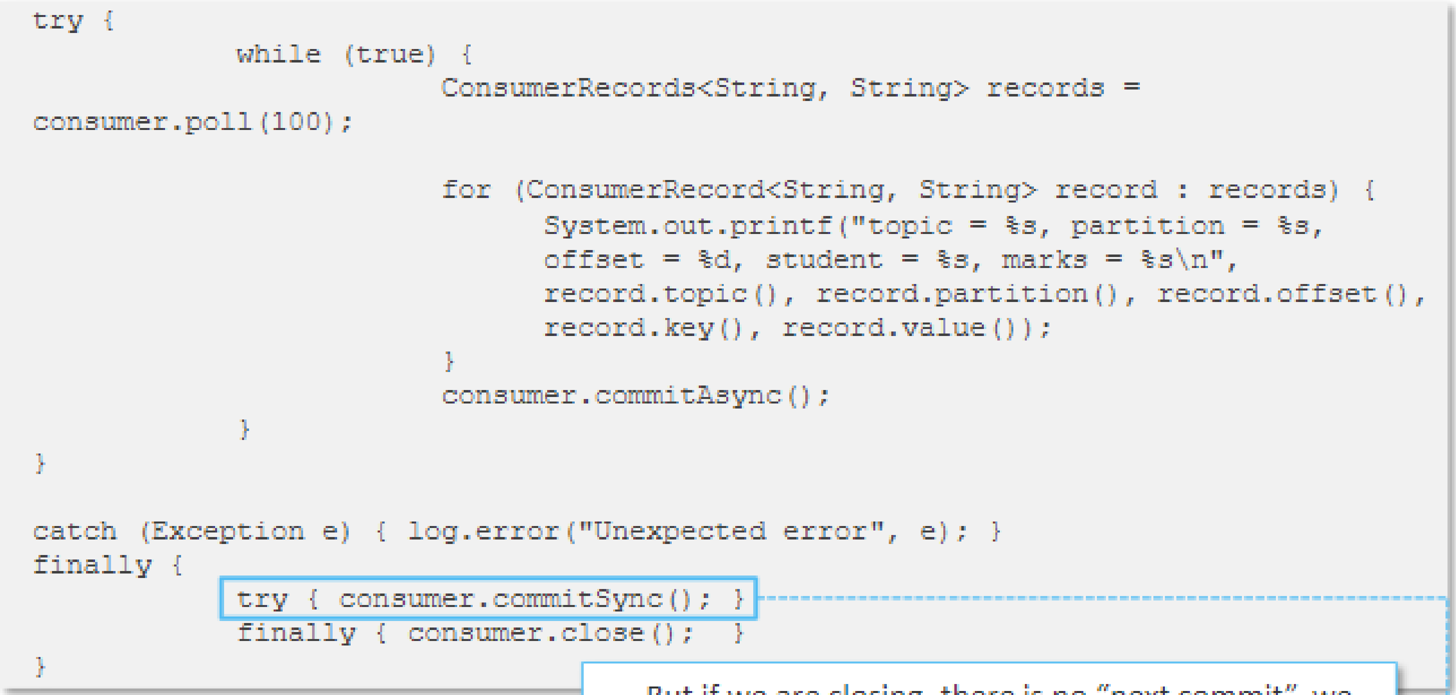


```
try {
            while (true) {
                        ConsumerRecords<String, String> records =
consumer.poll(100);

                        for (ConsumerRecord<String, String> record : records) {
                             System.out.printf("topic = %s, partition = %s,
                             offset = %d, student = %s, marks = %s\n",
                             record.topic(), record.partition(), record.offset(),
                             record.key(), record.value());
                        }
                        consumer.commitAsync();
            }
}

catch (Exception e) { log.error("Unexpected error", e); }
finally {
            try { consumer.commitSync(); }
            finally { consumer.close();  }
}
```

While everything is fine, we use *commitAsync*. It is faster, and if one commit fails, the next commit will serve as a retry.

Combining Synchronous & Asynchronous Commits

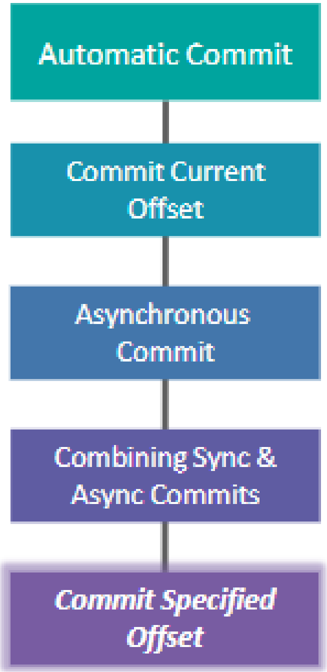
Automatic Commit
Commit Current Offset
Asynchronous Commit
Combining Sync & Async Commits
Commit Specified Offset



```
try {
            while (true) {
                        ConsumerRecords<String, String> records =
consumer.poll(100);

                        for (ConsumerRecord<String, String> record : records) {
                              System.out.printf("topic = %s, partition = %s,
                              offset = %d, student = %s, marks = %s\n",
                              record.topic(), record.partition(), record.offset(),
                              record.key(), record.value());
                        }
                        consumer.commitAsync();
            }
}

catch (Exception e) { log.error("Unexpected error", e); }
finally {
            try { consumer.commitSync(); }
            finally { consumer.close();  }
}
```

But if we are closing, there is no "next commit", we call *commitSync()*, because it will retry until it succeeds or suffers unrecoverable failure

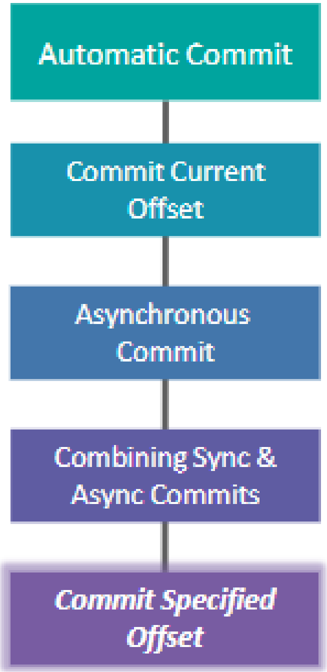
Commit Specified Offset



To commit a specific offsets, you can’t just call *commitSync()* or *commitAsync()* and pass a map of partitions and offsets that you wish to commit

- Consumer API allows you to call *commitSync()* and *commitAsync()* and pass a map of partitions and offsets that you wish to commit
- If you are in the middle of processing a batch of records, and the last message you got from partition 3 in topic “students” has offset 5000, you can call *commitSync()* to commit offset 5000 for partition 3 in topic “students.”

Commit Specified Offset

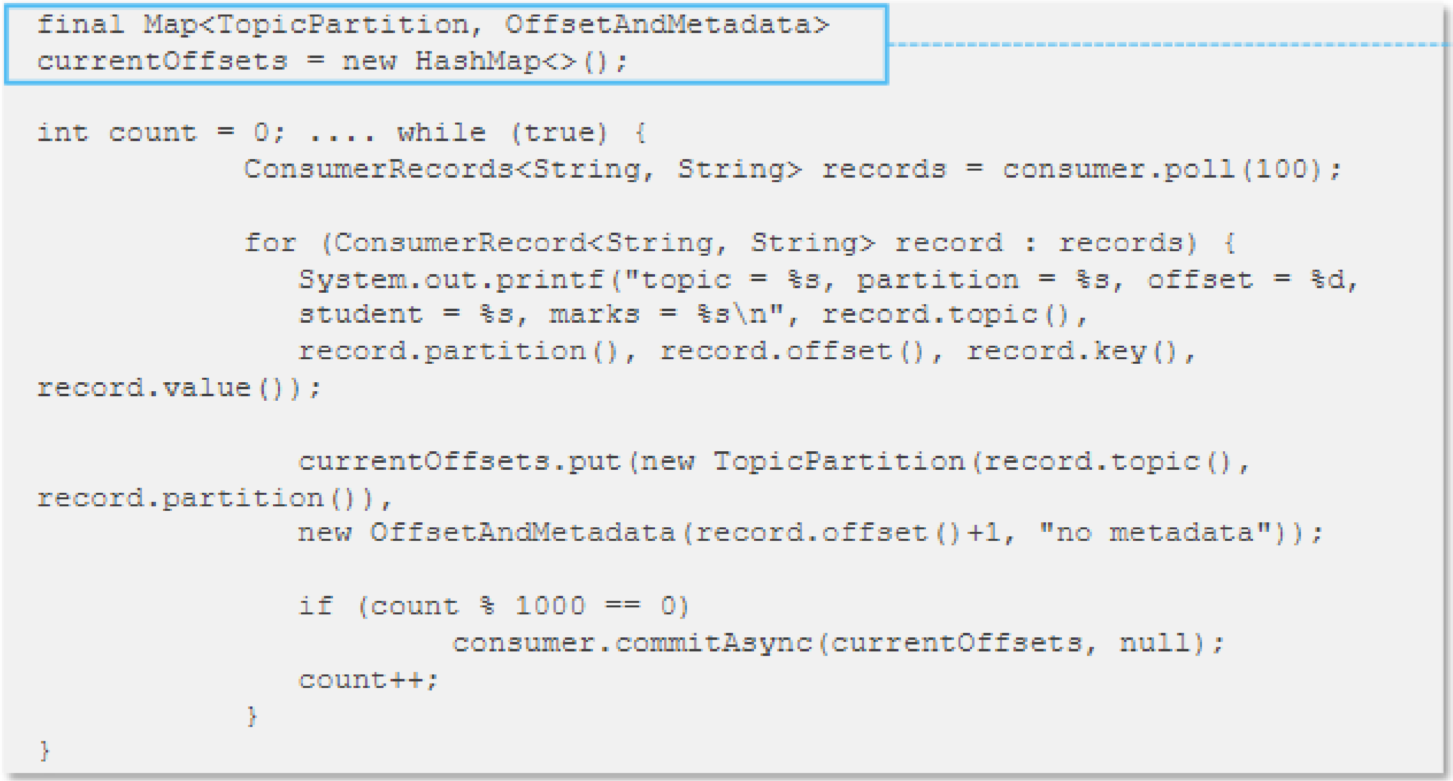

Automatic Commit

Commit Current Offset

Asynchronous Commit

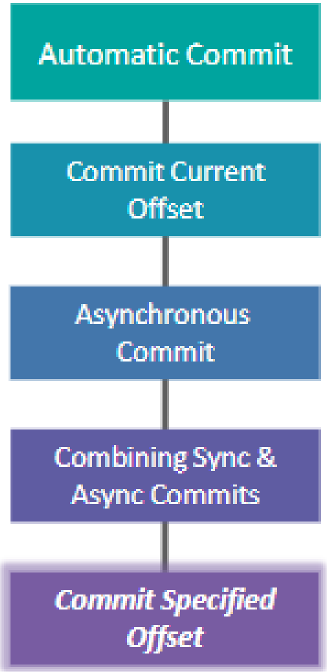
Combining Sync & Async Commits

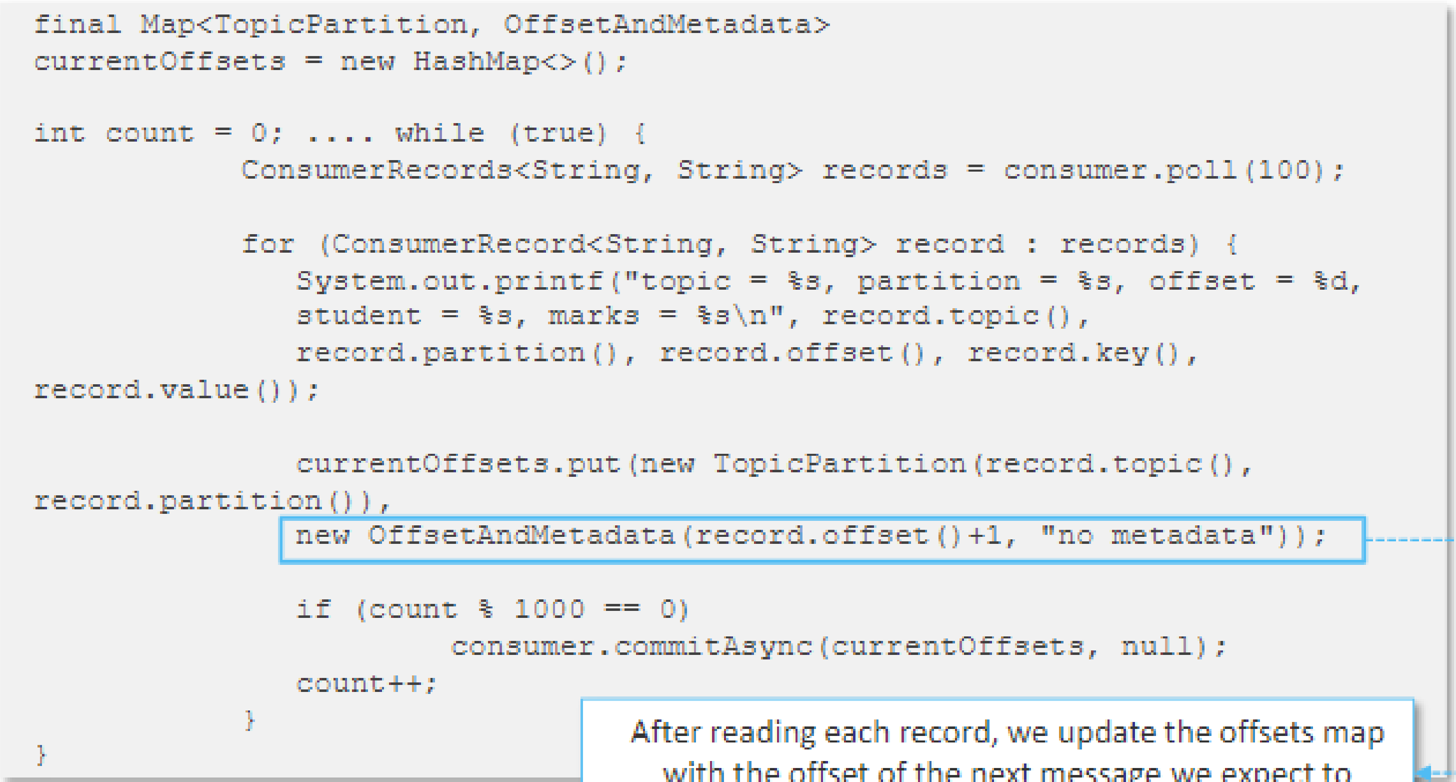
Commit Specified Offset


```
final Map<TopicPartition, OffsetAndMetadata>
currentOffsets = new HashMap<>();

int count = 0; .... while (true) {
            ConsumerRecords<String, String> records = consumer.poll(100);

            for (ConsumerRecord<String, String> record : records) {
               System.out.printf("topic = %s, partition = %s, offset = %d,
               student = %s, marks = %s\n", record.topic(),
               record.partition(), record.offset(), record.key(),
record.value());

               currentOffsets.put(new TopicPartition(record.topic(),
record.partition()),
               new OffsetAndMetadata(record.offset()+1, "no metadata"));

               if (count % 1000 == 0)
                        consumer.commitAsync(currentOffsets, null);
               count++;
            }
}
```

This is the map we will use to manually track offsets

Commit Specified Offset

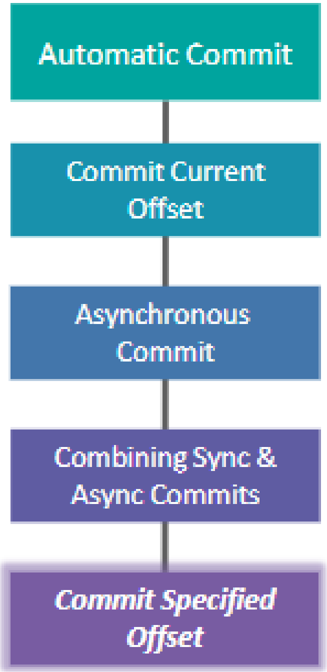




```
final Map<TopicPartition, OffsetAndMetadata>
currentOffsets = new HashMap<>();

int count = 0; .... while (true) {
            ConsumerRecords<String, String> records = consumer.poll(100);

            for (ConsumerRecord<String, String> record : records) {
               System.out.printf("topic = %s, partition = %s, offset = %d,
               student = %s, marks = %s\n", record.topic(),
               record.partition(), record.offset(), record.key(),
record.value());

               currentOffsets.put(new TopicPartition(record.topic(),
record.partition()),
               new OffsetAndMetadata(record.offset()+1, "no metadata"));

               if (count % 1000 == 0)
                        consumer.commitAsync(currentOffsets, null);
               count++;
            }
}
```

After reading each record, we update the offsets map with the offset of the next message we expect to process

Commit Specified Offset



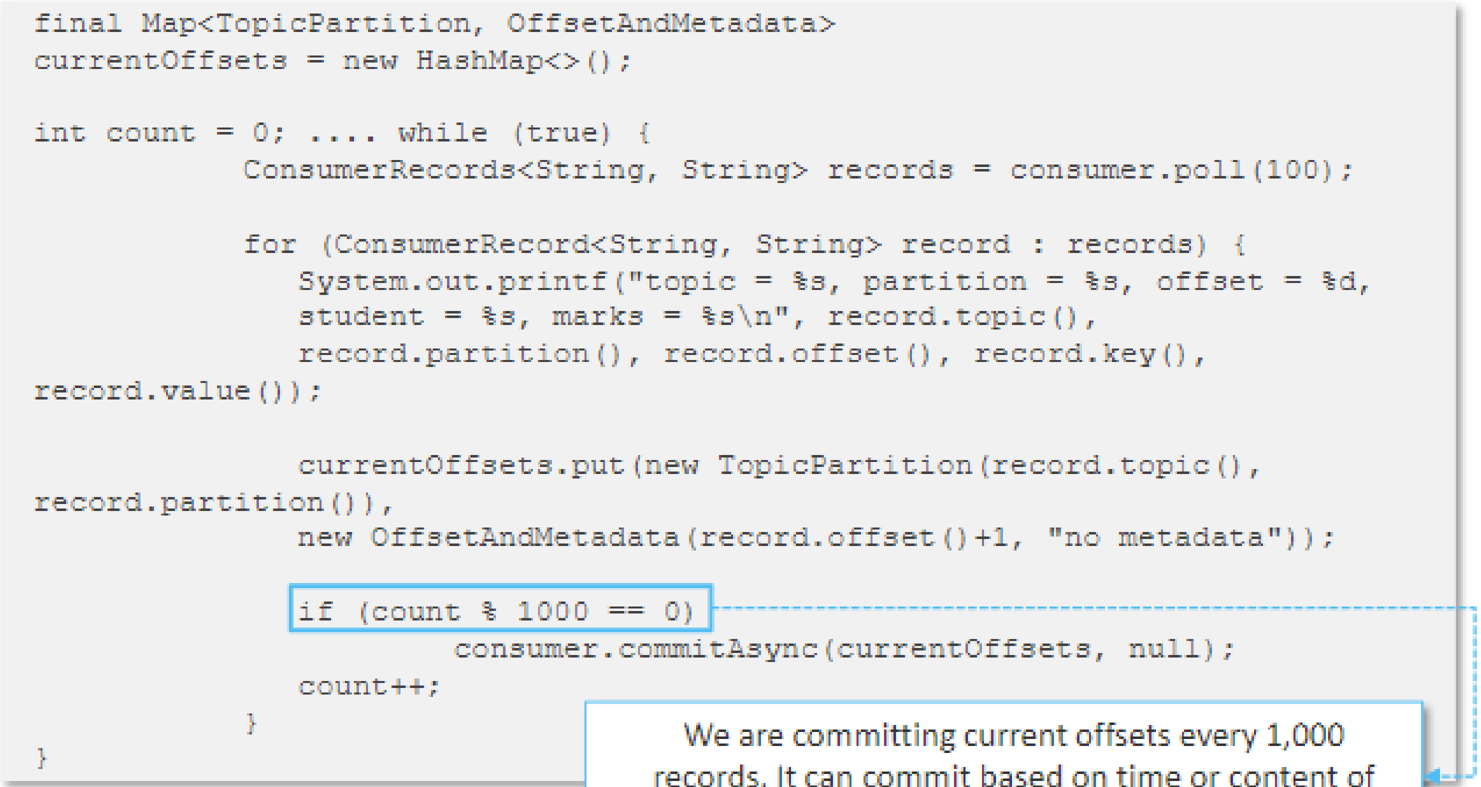
Automatic Commit

Commit Current Offset

Asynchronous Commit

Combining Sync & Async Commits

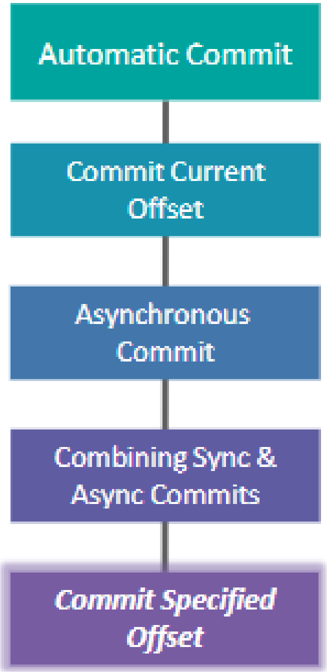
Commit Specified Offset



```
final Map<TopicPartition, OffsetAndMetadata>
currentOffsets = new HashMap<>();

int count = 0; .... while (true) {
           ConsumerRecords<String, String> records = consumer.poll(100);

           for (ConsumerRecord<String, String> record : records) {
              System.out.printf("topic = %s, partition = %s, offset = %d,
              student = %s, marks = %s\n", record.topic(),
              record.partition(), record.offset(), record.key(),
record.value());

              currentOffsets.put(new TopicPartition(record.topic(),
record.partition()),
              new OffsetAndMetadata(record.offset()+1, "no metadata"));

              if (count % 1000 == 0)
                     consumer.commitAsync(currentOffsets, null);
              count++;
           }
}
```

We are committing current offsets every 1,000 records. It can commit based on time or content of the records

Commit Specified Offset



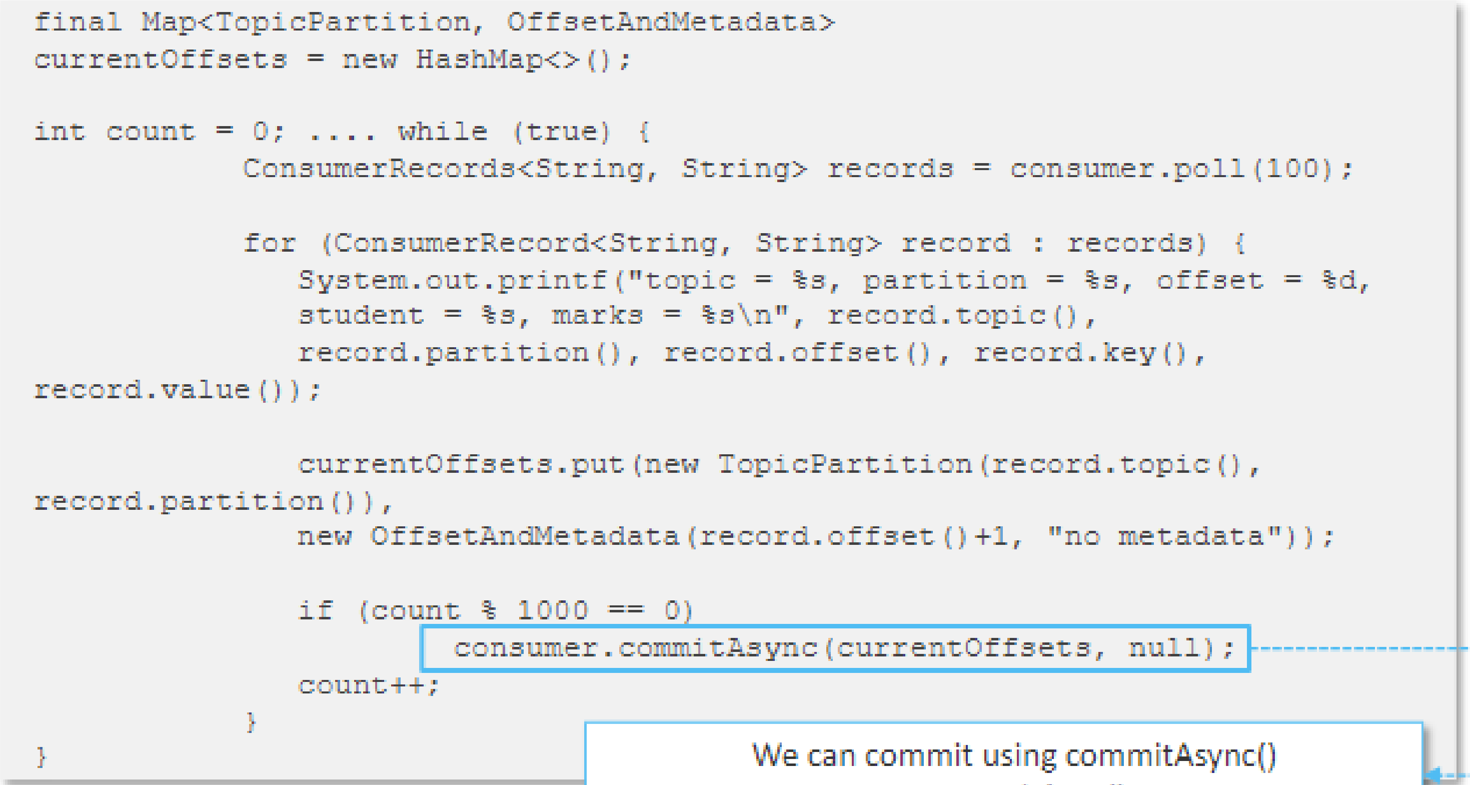
Automatic Commit

Commit Current Offset

Asynchronous Commit

Combining Sync & Async Commits

Commit Specified Offset



```
final Map<TopicPartition, OffsetAndMetadata>
currentOffsets = new HashMap<>();

int count = 0; .... while (true) {
            ConsumerRecords<String, String> records = consumer.poll(100);

            for (ConsumerRecord<String, String> record : records) {
               System.out.printf("topic = %s, partition = %s, offset = %d,
               student = %s, marks = %s\n", record.topic(),
               record.partition(), record.offset(), record.key(),
record.value());

               currentOffsets.put(new TopicPartition(record.topic(),
record.partition()),
               new OffsetAndMetadata(record.offset()+1, "no metadata"));

               if (count % 1000 == 0)
                     consumer.commitAsync(currentOffsets, null);
               count++;
            }
}
```

We can commit using commitAsync()
or commitSync()

Let’s see how to rebalance Listeners

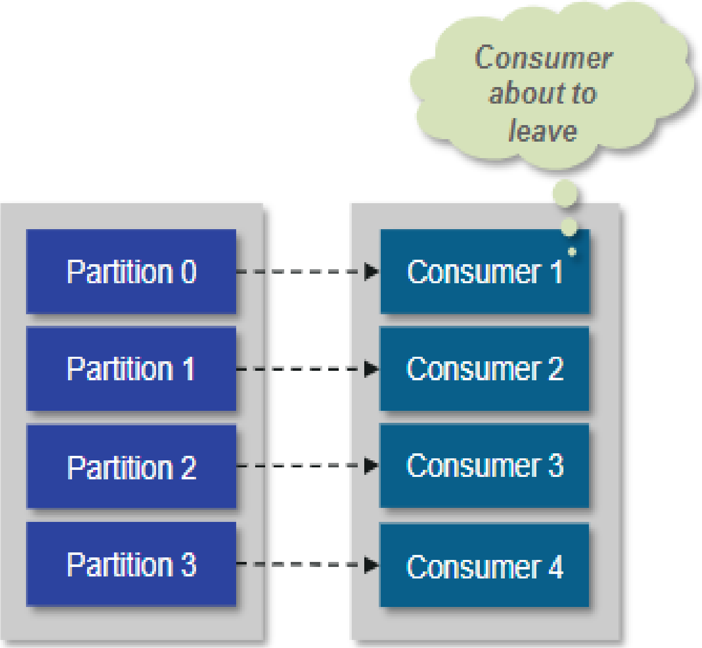
Rebalance Listeners

Consumer needs to do some cleanup work before exiting or before partition rebalancing

Before a consumer loses ownership of a partition, we need to commit offsets of the last event we've processed

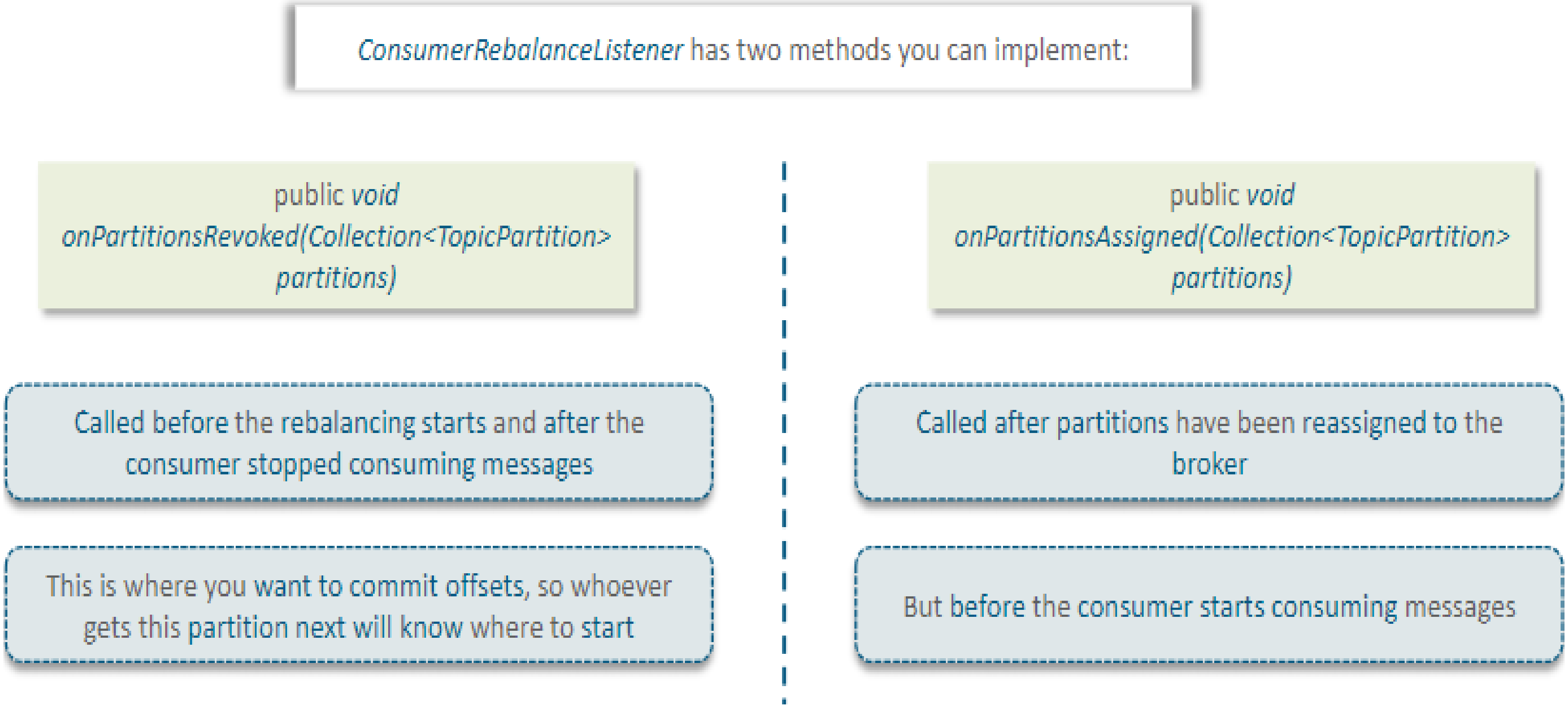
If consumer maintained a buffer with events, we need to process the accumulated events before losing ownership

The consumer API allows you to execute code when partitions are added or removed



☛ *You do this by passing a ConsumerRebalanceListener when calling the subscribe() method*

Rebalance Listeners : Methods



ConsumerRebalanceListener has two methods you can implement:

public *void onPartitionsRevoked(Collection<TopicPartition> partitions)*

Called before the rebalancing starts and after the consumer stopped consuming messages

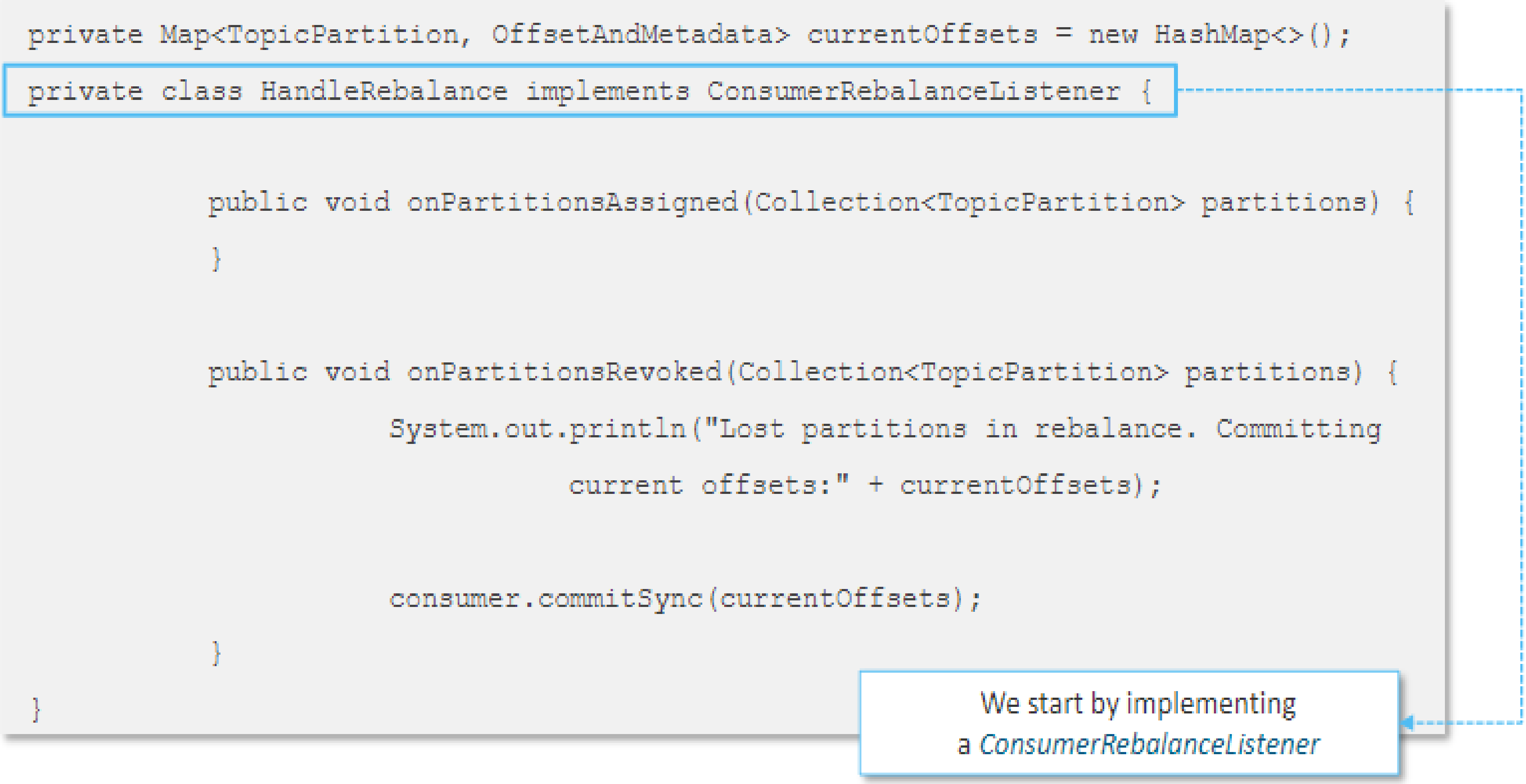
This is where you want to commit offsets, so whoever gets this partition next will know where to start

public *void onPartitionsAssigned(Collection<TopicPartition> partitions)*

Called after partitions have been reassigned to the broker

But before the consumer starts consuming messages

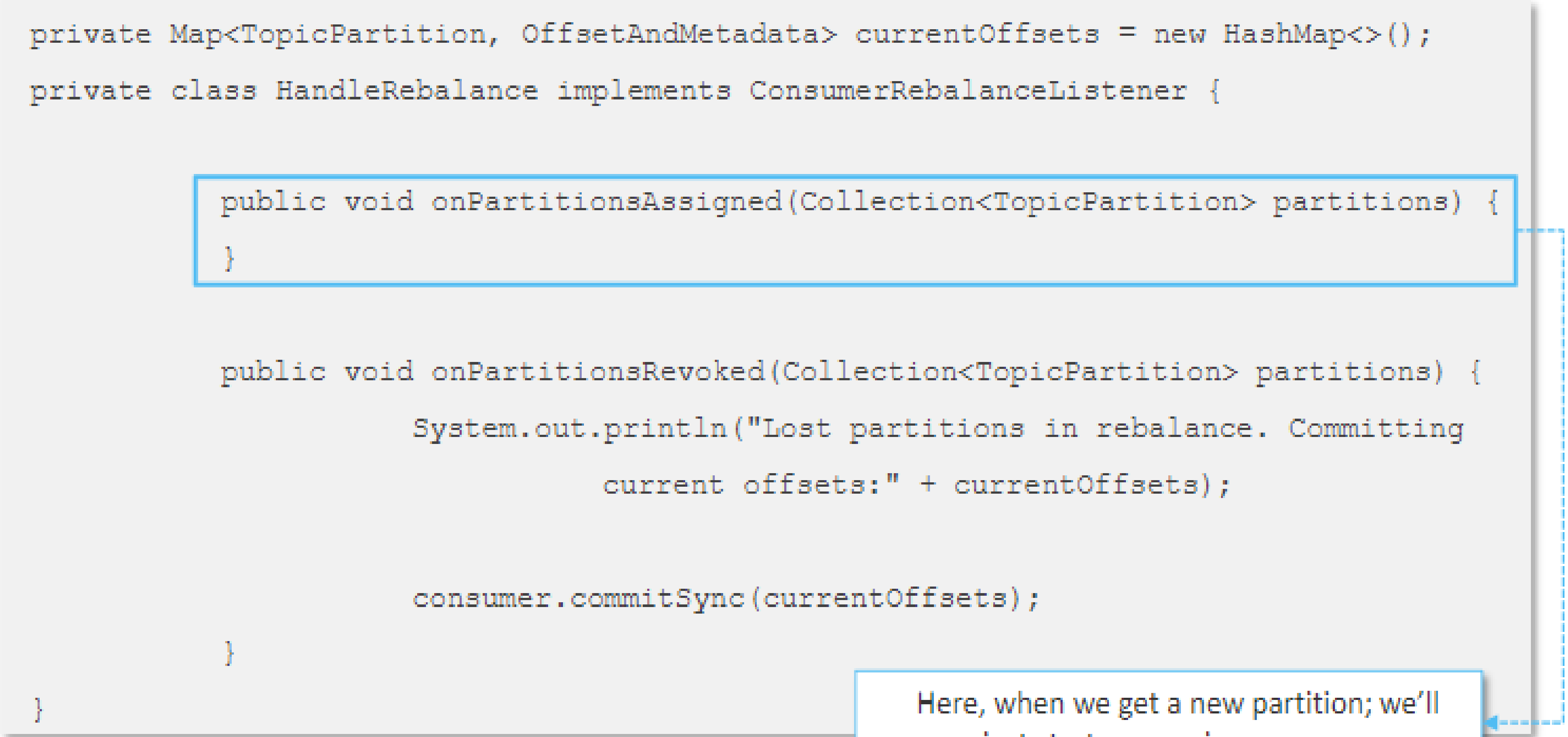
Rebalance Listeners : Sample Program



```
private Map<TopicPartition, OffsetAndMetadata> currentOffsets = new HashMap<>();
private class HandleRebalance implements ConsumerRebalanceListener {


          public void onPartitionsAssigned(Collection<TopicPartition> partitions) {
          }


          public void onPartitionsRevoked(Collection<TopicPartition> partitions) {
                    System.out.println("Lost partitions in rebalance. Committing
                              current offsets:" + currentOffsets);


                    consumer.commitSync(currentOffsets);
          }
}
```

We start by implementing
a *ConsumerRebalanceListener*

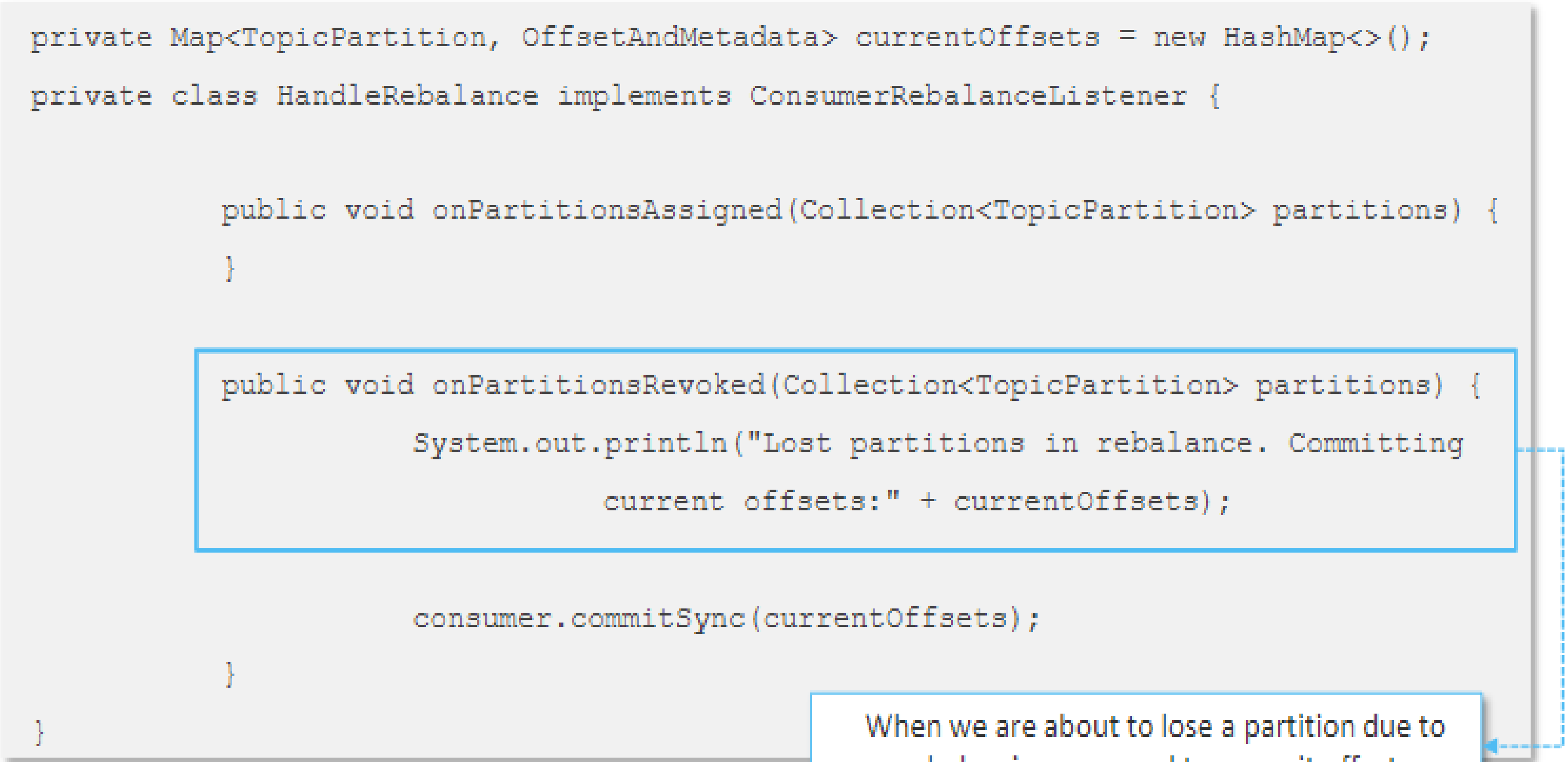
Rebalance Listeners : Sample Program



```
private Map<TopicPartition, OffsetAndMetadata> currentOffsets = new HashMap<>();
private class HandleRebalance implements ConsumerRebalanceListener {

     public void onPartitionsAssigned(Collection<TopicPartition> partitions) {
     }

     public void onPartitionsRevoked(Collection<TopicPartition> partitions) {
          System.out.println("Lost partitions in rebalance. Committing
                current offsets:" + currentOffsets);

          consumer.commitSync(currentOffsets);
     }
}
```

Here, when we get a new partition; we'll just start consuming messages

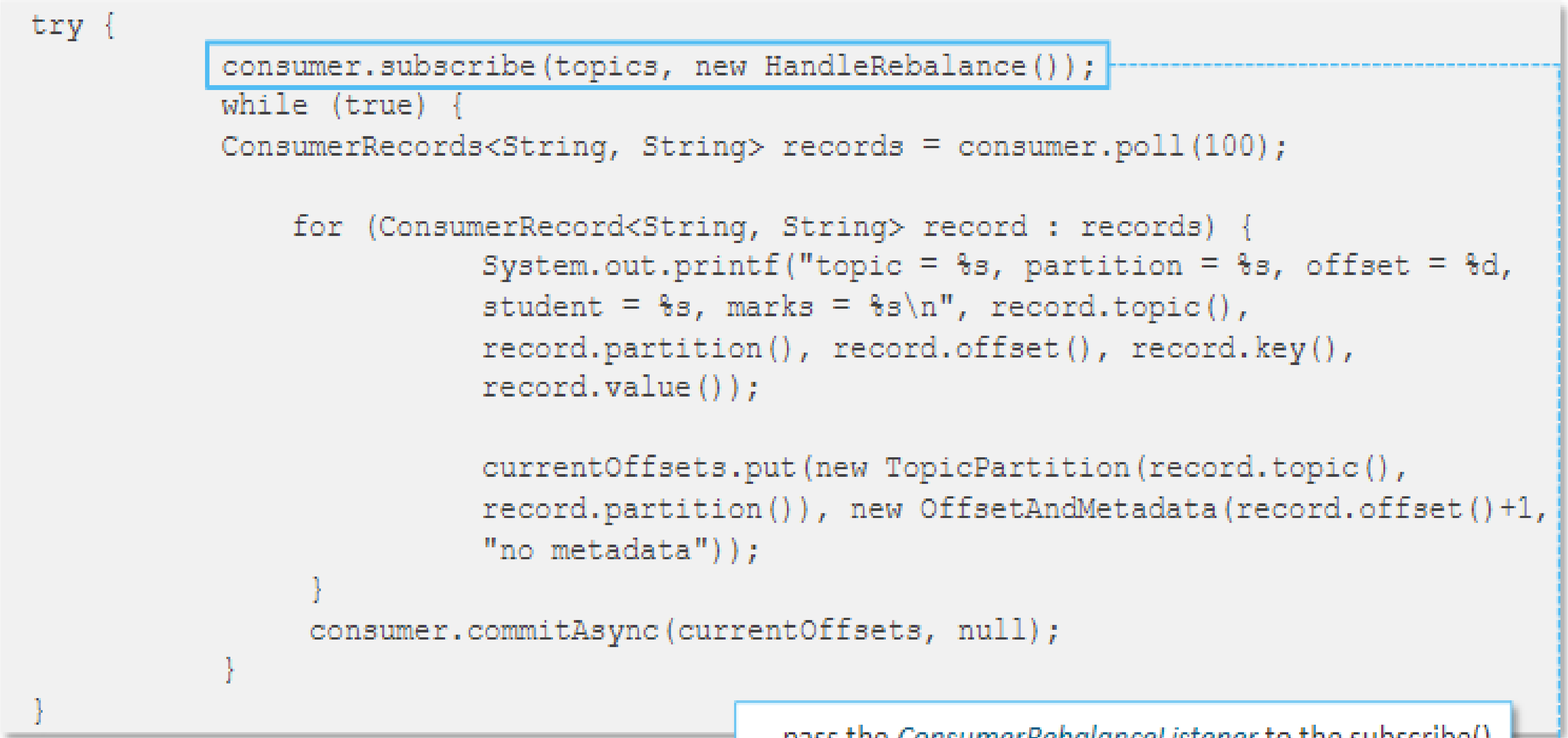
Rebalance Listeners : Sample Program



```
private Map<TopicPartition, OffsetAndMetadata> currentOffsets = new HashMap<>();
private class HandleRebalance implements ConsumerRebalanceListener {

          public void onPartitionsAssigned(Collection<TopicPartition> partitions) {
          }

          public void onPartitionsRevoked(Collection<TopicPartition> partitions) {
                    System.out.println("Lost partitions in rebalance. Committing
                              current offsets:" + currentOffsets);

                    consumer.commitSync(currentOffsets);
          }
}
```

When we are about to lose a partition due to rebalancing, we need to commit offsets

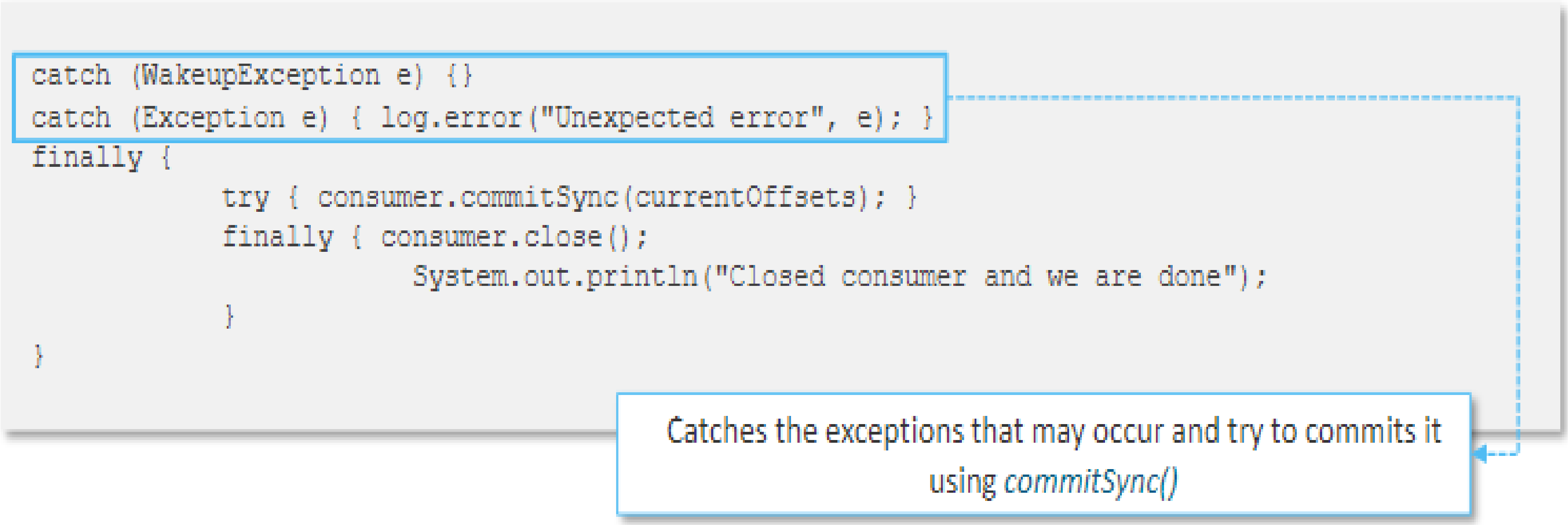
Rebalance Listeners : Sample Program



```
try {
          consumer.subscribe(topics, new HandleRebalance());
          while (true) {
          ConsumerRecords<String, String> records = consumer.poll(100);

              for (ConsumerRecord<String, String> record : records) {
                         System.out.printf("topic = %s, partition = %s, offset = %d,
                         student = %s, marks = %s\n", record.topic(),
                         record.partition(), record.offset(), record.key(),
                         record.value());

                         currentOffsets.put(new TopicPartition(record.topic(),
                         record.partition()), new OffsetAndMetadata(record.offset()+1,
                         "no metadata"));
               }
               consumer.commitAsync(currentOffsets, null);
          }
}
```

pass the *ConsumerRebalanceListener* to the subscribe() method so it will get invoked by the consumer

Rebalance Listeners : Sample Program



```
catch (WakeupException e) {}
catch (Exception e) { log.error("Unexpected error", e); }
finally {
            try { consumer.commitSync(currentOffsets); }
            finally { consumer.close();
                        System.out.println("Closed consumer and we are done");
            }
}
```

Catches the exceptions that may occur and try to commits it using *commitSync()*

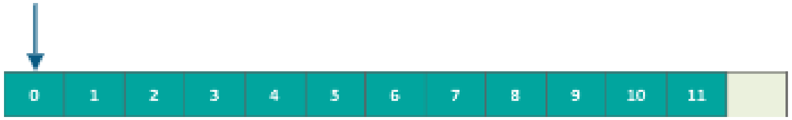
Consuming Records with Specific Offsets

Consuming Records with Specific Offsets

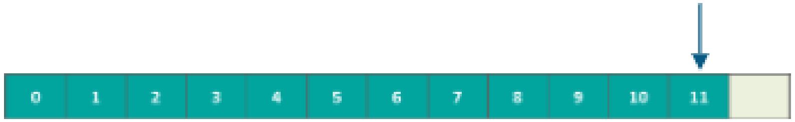
poll() starts consuming messages from the last committed offset in each partition and to proceed in processing all messages in sequence

But sometimes we want to start reading at a different offset:

If we want to start reading all messages from the beginning of the partition, we use *seekToBeginning(TopicPartition tp)*

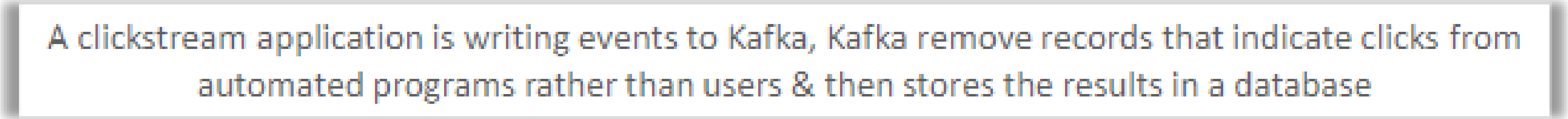


If we want to skip all the way to the end of the partition & consume only new messages, we use *seekToEnd(TopicPartition tp)*

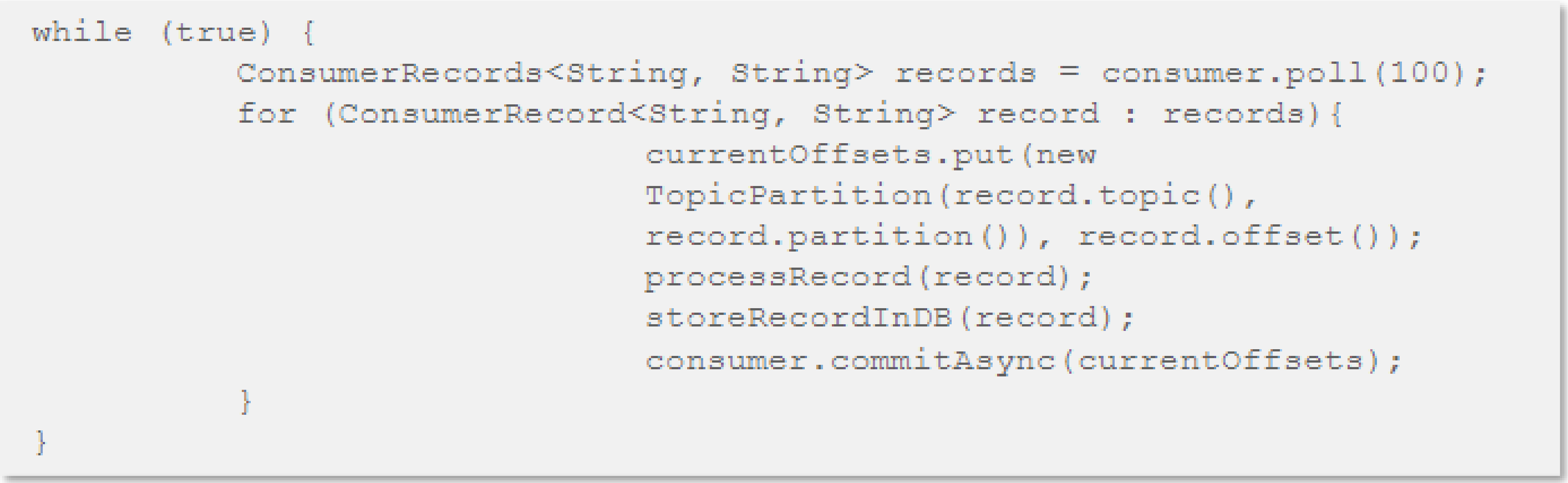


☛ *Perhaps a time-sensitive application that is falling behind will want to skip ahead to more relevant messages*

Consuming Records with Specific Offsets



A clickstream application is writing events to Kafka, Kafka remove records that indicate clicks from automated programs rather than users & then stores the results in a database

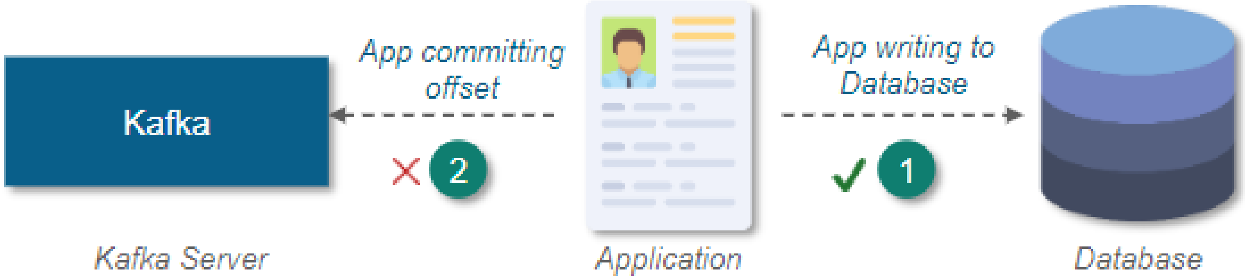


```
while (true) {
          ConsumerRecords<String, String> records = consumer.poll(100);
          for (ConsumerRecord<String, String> record : records){
                              currentOffsets.put(new
                              TopicPartition(record.topic(),
                              record.partition()), record.offset());
                              processRecord(record);
                              storeRecordInDB(record);
                              consumer.commitAsync(currentOffsets);
          }
}
```

☛ *We don't want to lose any data, nor do we want to store the same results in the database twice*

Consuming Records with Specific Offsets

If we commit offsets after processing each record, but still there are chances that application will crash after the record was stored in the database, before committing the offsets



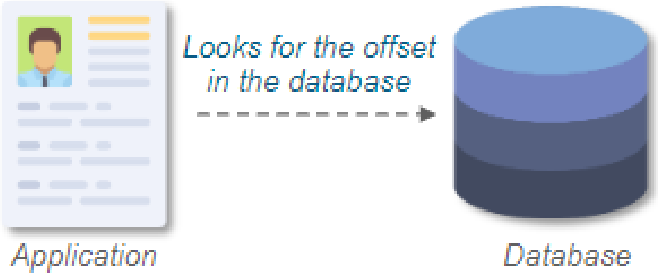
- This causes the record to be processed again and the database to contain duplicates
- To avoid this, we need to store both the record and the offset in one atomic action

☛ *Either both the record and the offset are committed, or neither of them are committed.*

Consuming Records with Specific Offsets

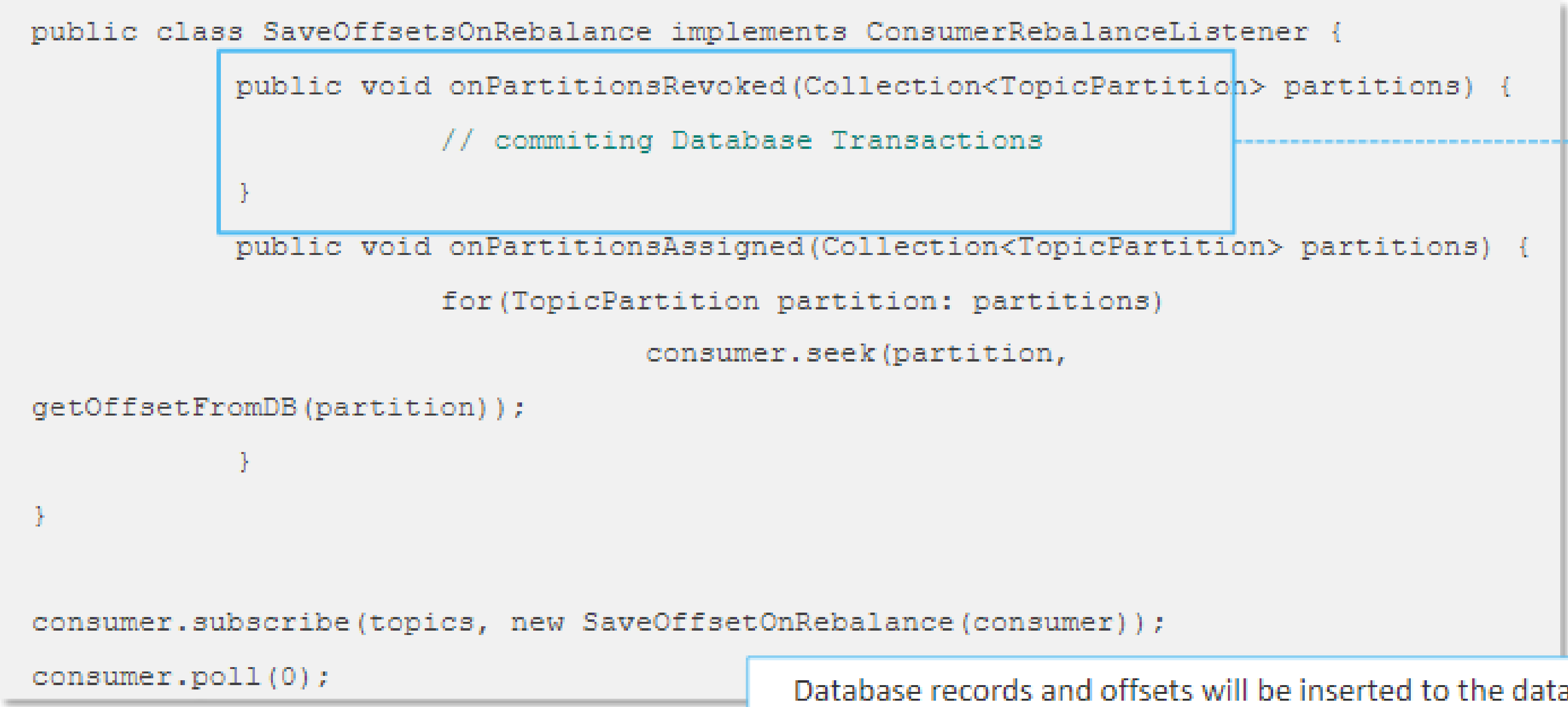
Only problem is, if the record is stored in a database and not in Kafka, how will our consumer know where to start reading?

- *seek()* is used
- When the consumer starts with new partitions, it can look up the offset in the database and *seek()* to that location



☛ *We use ConsumerRebalanceLister and seek() to make sure we start processing at the offsets stored in the database*

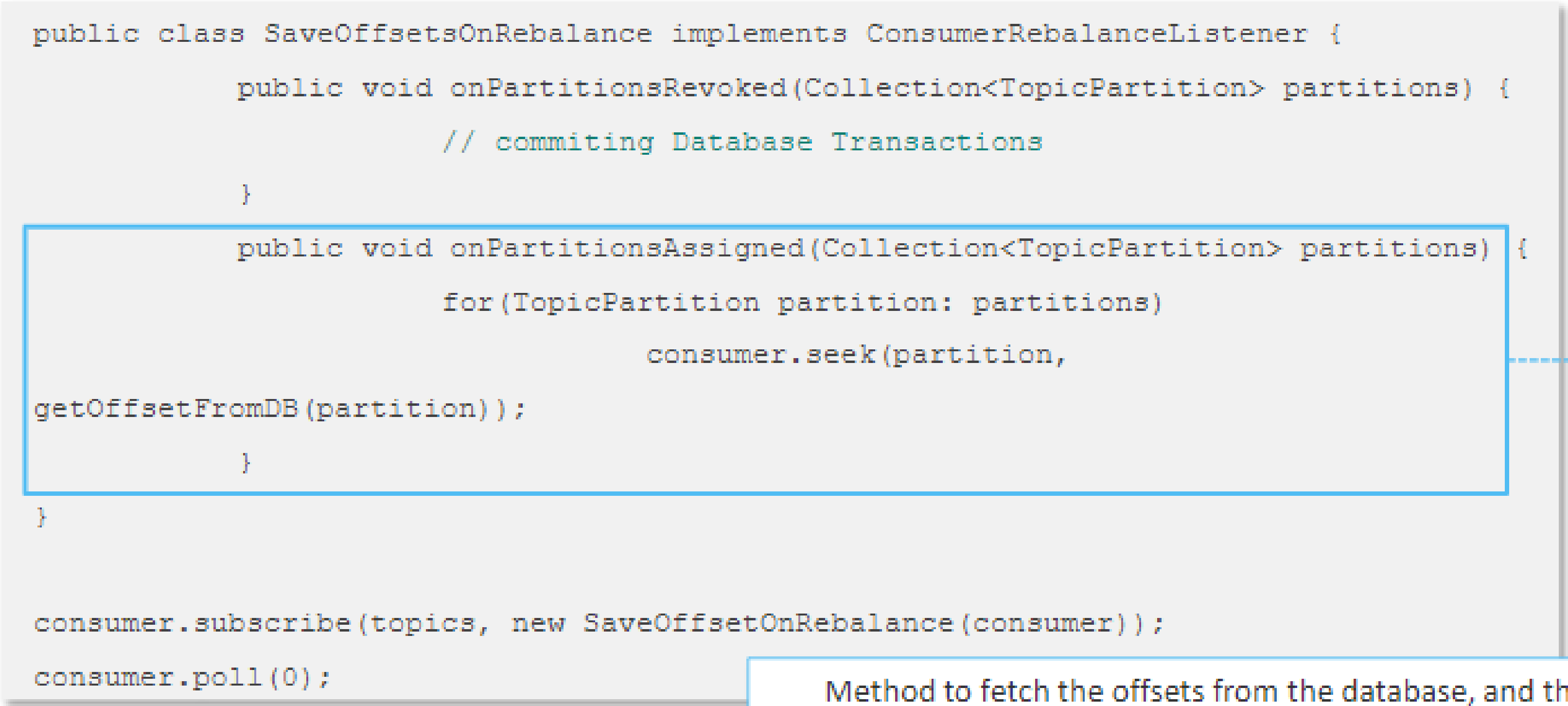
Consuming Records with Specific Offsets



```
public class SaveOffsetsOnRebalance implements ConsumerRebalanceListener {
            public void onPartitionsRevoked(Collection<TopicPartition> partitions) {
                        // commiting Database Transactions
            }
            public void onPartitionsAssigned(Collection<TopicPartition> partitions) {
                        for(TopicPartition partition: partitions)
                                    consumer.seek(partition,
getOffsetFromDB(partition));
            }
}

consumer.subscribe(topics, new SaveOffsetOnRebalance(consumer));
consumer.poll(0);
```

Database records and offsets will be inserted to the database as we process the records

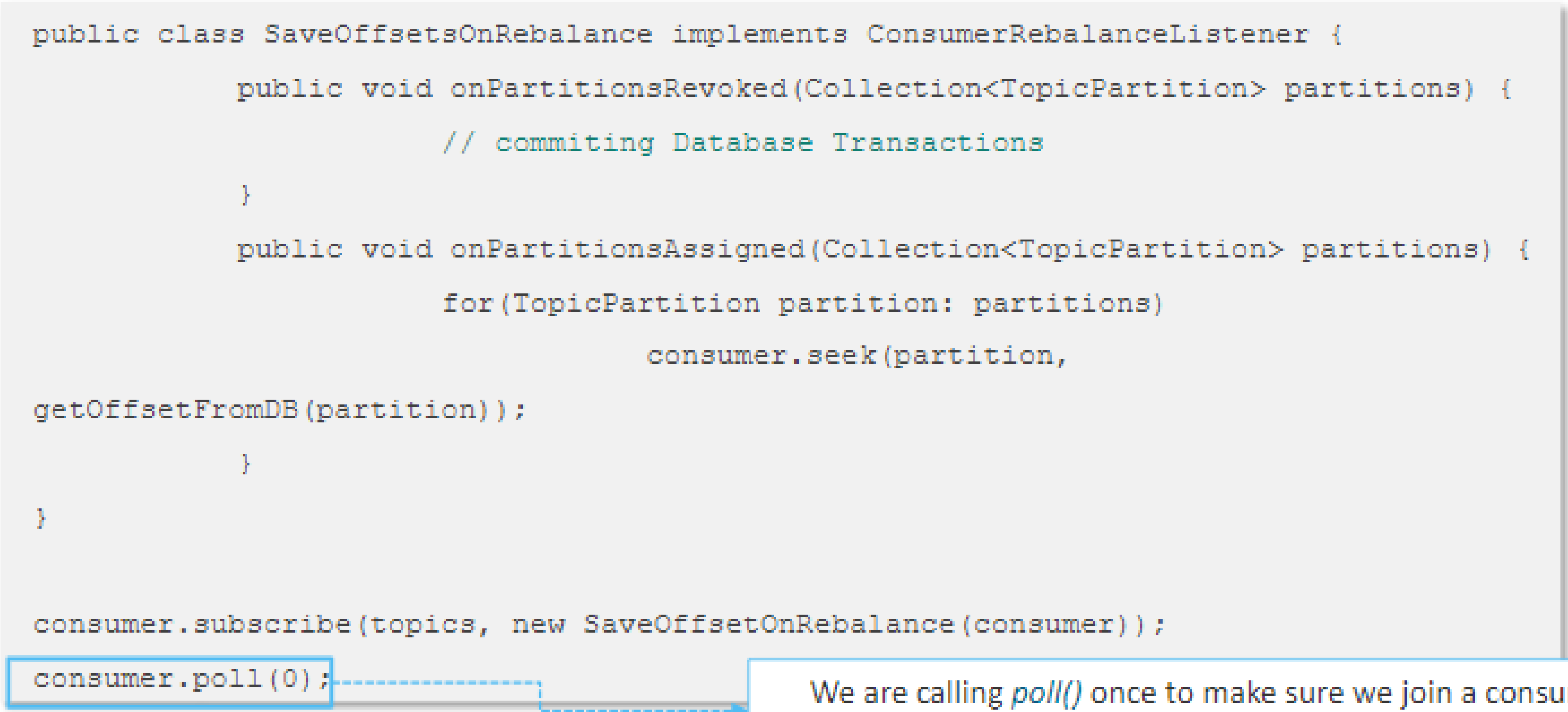
Consuming Records with Specific Offsets



```
public class SaveOffsetsOnRebalance implements ConsumerRebalanceListener {
            public void onPartitionsRevoked(Collection<TopicPartition> partitions) {
                        // commiting Database Transactions
            }
            public void onPartitionsAssigned(Collection<TopicPartition> partitions) {
                        for(TopicPartition partition: partitions)
                                    consumer.seek(partition,
getOffsetFromDB(partition));
            }
}

consumer.subscribe(topics, new SaveOffsetOnRebalance(consumer));
consumer.poll(0);
```

Method to fetch the offsets from the database, and then we seek() to those records

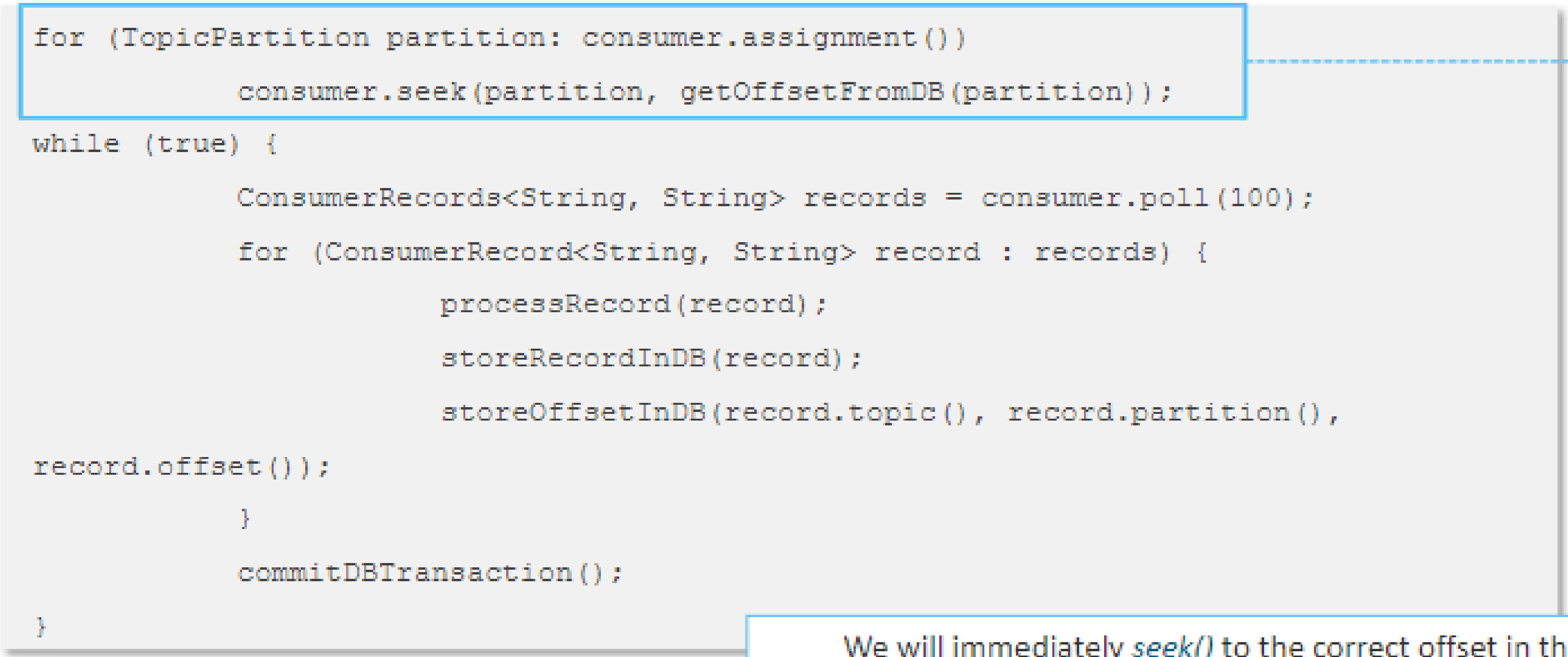
Consuming Records with Specific Offsets



```
public class SaveOffsetsOnRebalance implements ConsumerRebalanceListener {
            public void onPartitionsRevoked(Collection<TopicPartition> partitions) {
                        // commiting Database Transactions
            }
            public void onPartitionsAssigned(Collection<TopicPartition> partitions) {
                        for(TopicPartition partition: partitions)
                                    consumer.seek(partition,
getOffsetFromDB(partition));
            }
}

consumer.subscribe(topics, new SaveOffsetOnRebalance(consumer));
consumer.poll(0);
```

We are calling *poll()* once to make sure we join a consumer group and get assigned partitions

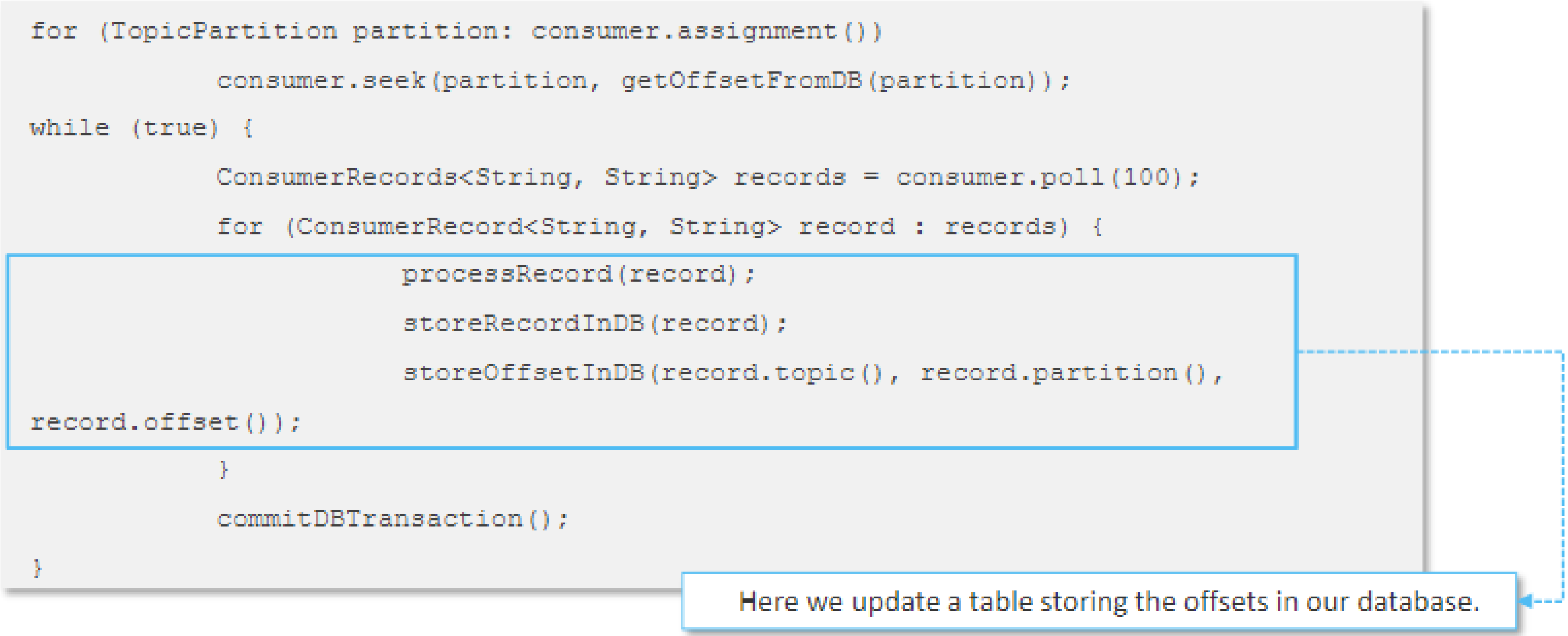
Consuming Records with Specific Offsets



```
for (TopicPartition partition: consumer.assignment())
            consumer.seek(partition, getOffsetFromDB(partition));
while (true) {
            ConsumerRecords<String, String> records = consumer.poll(100);
            for (ConsumerRecord<String, String> record : records) {
                        processRecord(record);
                        storeRecordInDB(record);
                        storeOffsetInDB(record.topic(), record.partition(),
record.offset());
            }
            commitDBTransaction();
}
```

We will immediately *seek()* to the correct offset in the partitions we are assigned to

Consuming Records with Specific Offsets



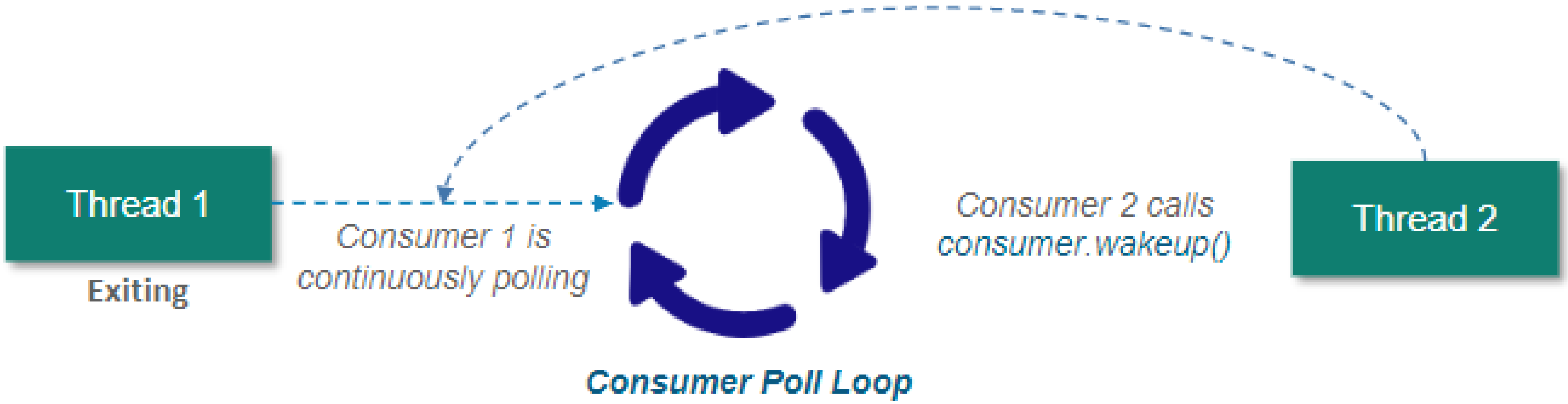
```
for (TopicPartition partition: consumer.assignment())
            consumer.seek(partition, getOffsetFromDB(partition));
while (true) {
            ConsumerRecords<String, String> records = consumer.poll(100);
            for (ConsumerRecord<String, String> record : records) {
                        processRecord(record);
                        storeRecordInDB(record);
                        storeOffsetInDB(record.topic(), record.partition(),
record.offset());
            }
            commitDBTransaction();
}
```

Here we update a table storing the offsets in our database.

How do we exit cleanly from Poll Loop?

Exiting from Poll Loop

Consumer polls in an infinite loop and it should exit the loop cleanly


Thread 1
Exiting
Consumer 1 is continuously polling
Consumer 2 calls consumer.wakeup()
Thread 2
Consumer Poll Loop

- When you decide to exit the poll loop, you will need another thread to call *consumer.wakeup()*
- If you are running the consumer loop in the main thread, this can be done from *ShutdownHook*

☛ *consumer.wakeup() is the only consumer method that is safe to call from a different thread*

Exiting from Poll Loop

Calling *wakeup* will cause *poll()* to exit with *WakeupException*

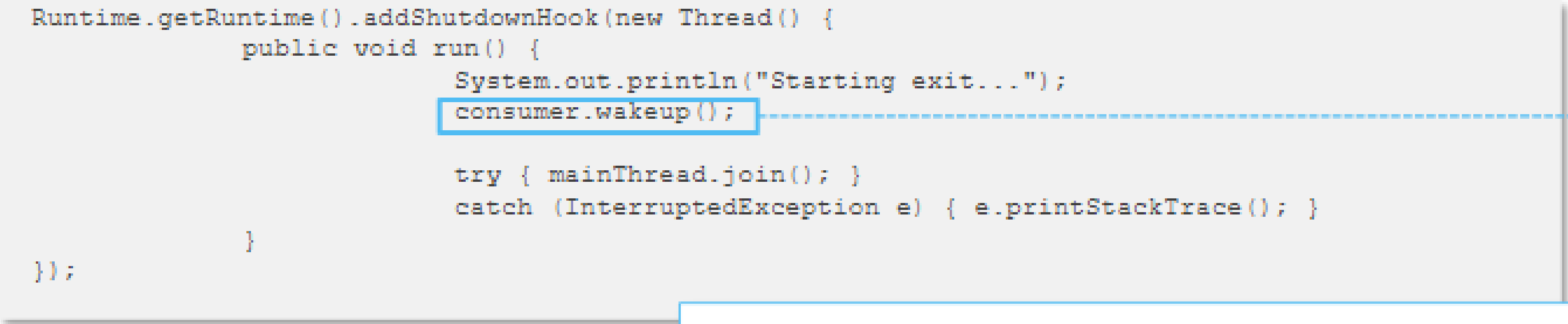
If *consumer.wakeup()* was called while the thread was not waiting on poll, the exception will be thrown on the next *poll()*

WakeupException doesn't need to be handled, but before exiting the thread, you must call *consumer.close()*

Closing the consumer will commit offsets and will notify the group coordinator that the consumer is leaving

☛ *The consumer coordinator will trigger rebalancing immediately & before session times out, partitions will be assigned to another consumer in the group*

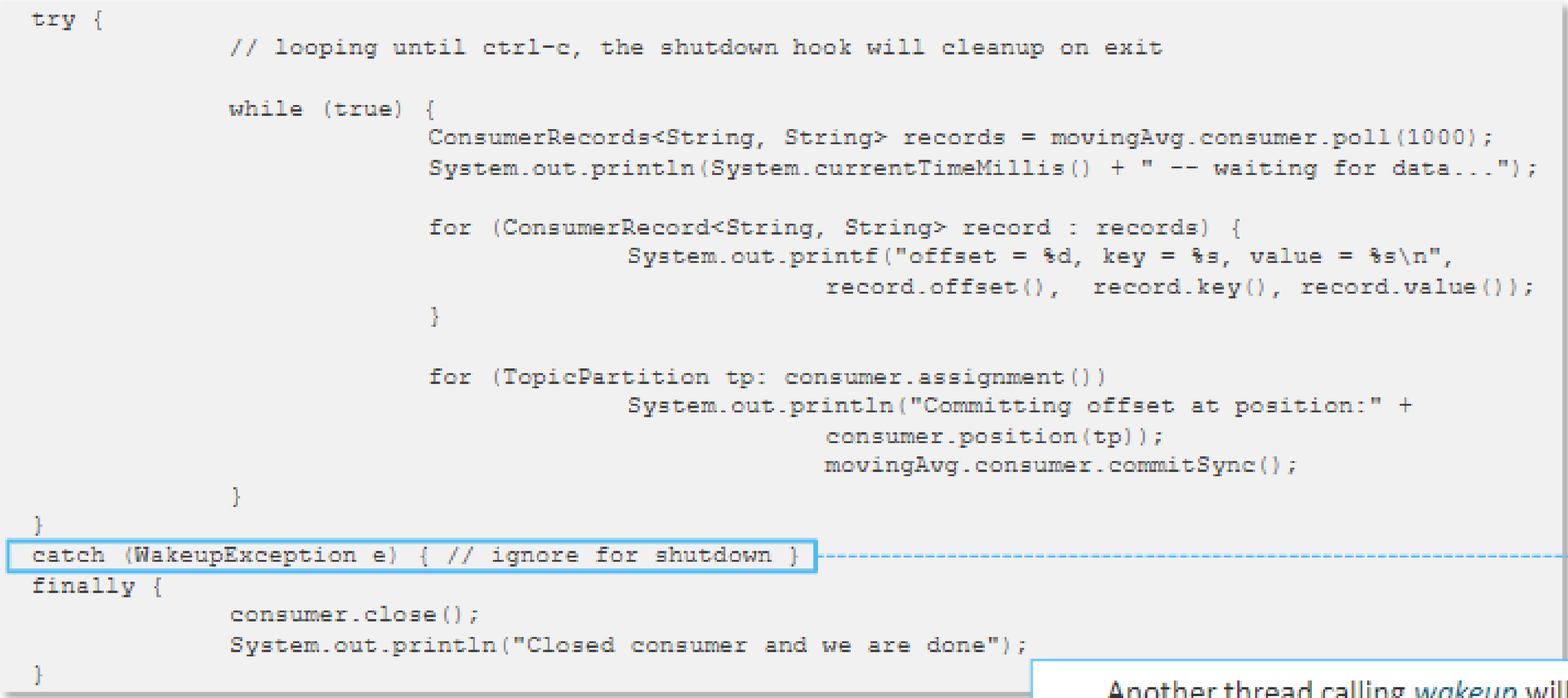
Exiting from Poll Loop



```
Runtime.getRuntime().addShutdownHook(new Thread() {
             public void run() {
                           System.out.println("Starting exit...");
                           consumer.wakeup();

                           try { mainThread.join(); }
                           catch (InterruptedException e) { e.printStackTrace(); }
             }
});
```

ShutdownHook runs in a seperate thread, so the only safe action we can take is to call wakeup to break out of the poll loop

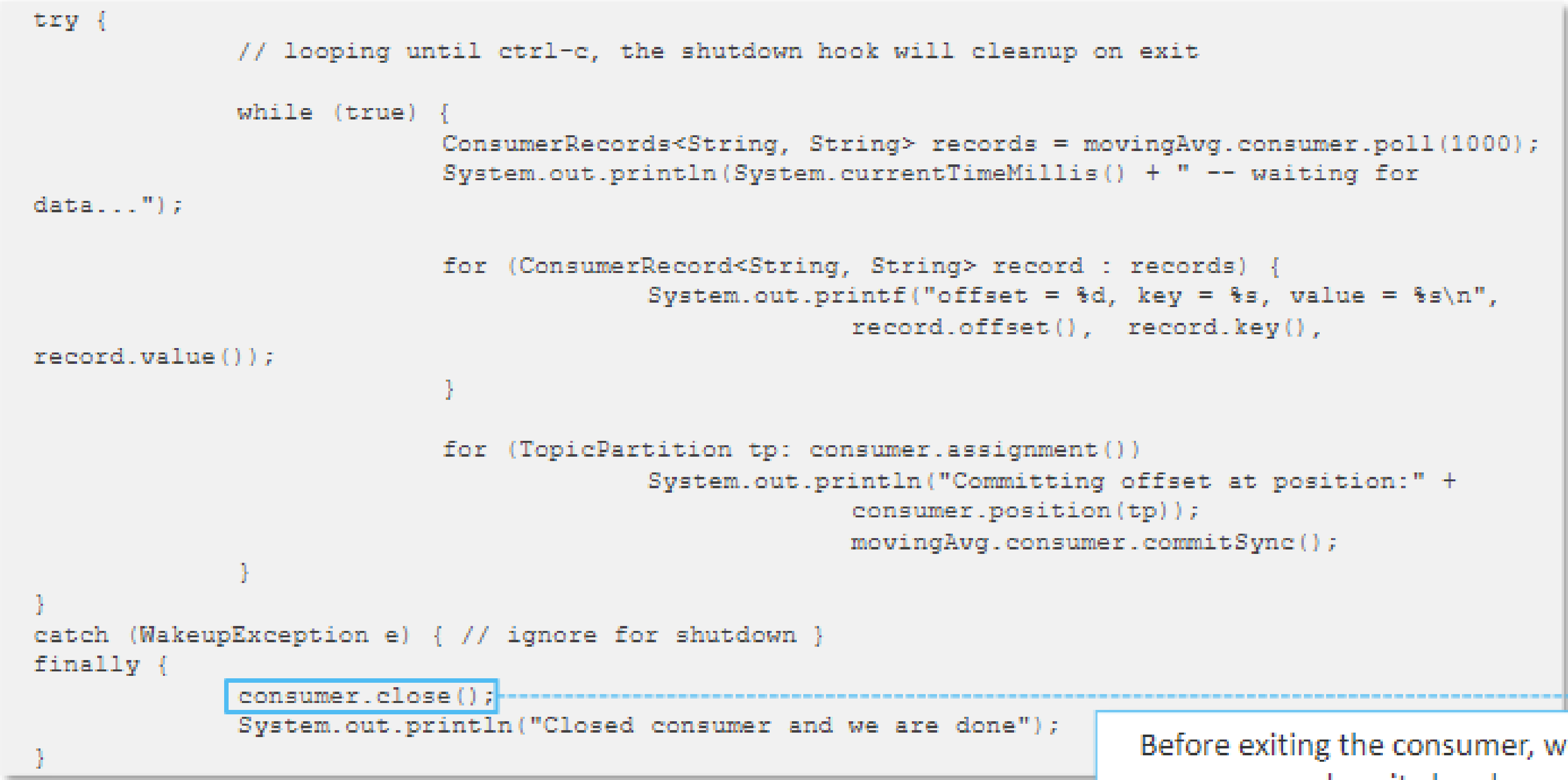
Exiting from Poll Loop



```
try {
          // looping until ctrl-c, the shutdown hook will cleanup on exit

          while (true) {
                    ConsumerRecords<String, String> records = movingAvg.consumer.poll(1000);
                    System.out.println(System.currentTimeMillis() + " -- waiting for data...");

                    for (ConsumerRecord<String, String> record : records) {
                              System.out.printf("offset = %d, key = %s, value = %s\n",
                                        record.offset(),  record.key(), record.value());
                    }

                    for (TopicPartition tp: consumer.assignment())
                              System.out.println("Committing offset at position:" +
                                        consumer.position(tp));
                                        movingAvg.consumer.commitSync();
          }
}
catch (WakeupException e) { // ignore for shutdown }
finally {
          consumer.close();
          System.out.println("Closed consumer and we are done");
}
```

Another thread calling *wakeup* will cause poll to throw a *WakeupException*

Exiting from Poll Loop



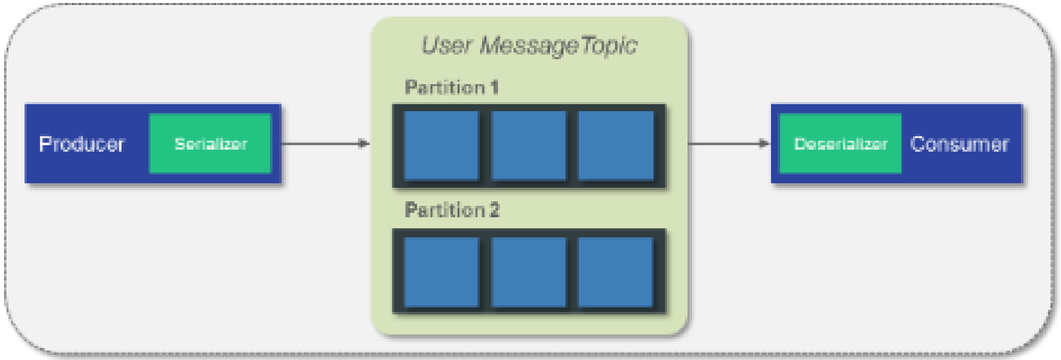
```
try {
          // looping until ctrl-c, the shutdown hook will cleanup on exit

          while (true) {
                    ConsumerRecords<String, String> records = movingAvg.consumer.poll(1000);
                    System.out.println(System.currentTimeMillis() + " -- waiting for
data...");

                    for (ConsumerRecord<String, String> record : records) {
                              System.out.printf("offset = %d, key = %s, value = %s\n",
                                        record.offset(),  record.key(),
record.value());
                    }

                    for (TopicPartition tp: consumer.assignment())
                              System.out.println("Committing offset at position:" +
                                        consumer.position(tp));
                                        movingAvg.consumer.commitSync();
          }
}
catch (WakeupException e) { // ignore for shutdown }
finally {
          consumer.close();
          System.out.println("Closed consumer and we are done");
}
```

Before exiting the consumer, we should close it cleanly

Deserializers for Consumers

Deserializers

Kafka consumers require *deserializers* to convert *byte arrays* recieved from Kafka into *Java objects*

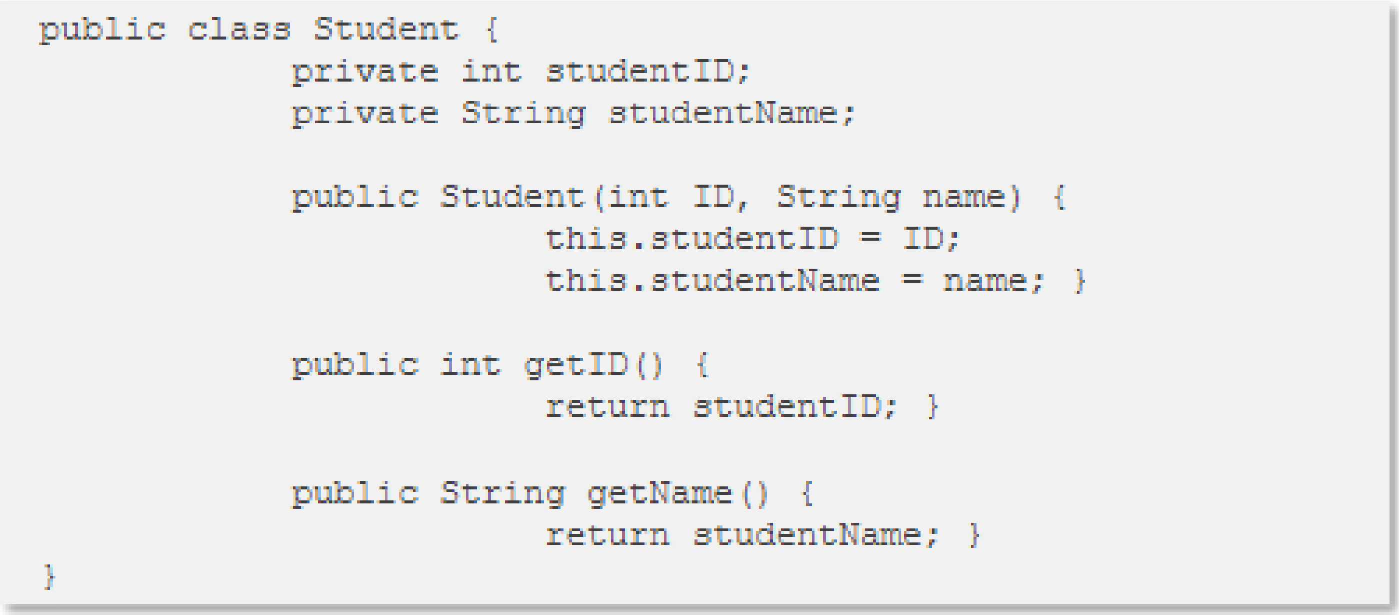


- We can create custom deserializers for own objects
- The serializer used to produce events to Kafka must match the deserializer that will be used when consuming events

☛ *We need to make sure which serializers were used to write into each topic & make sure the same deserializers is used to interpret*

Custom Deserializers

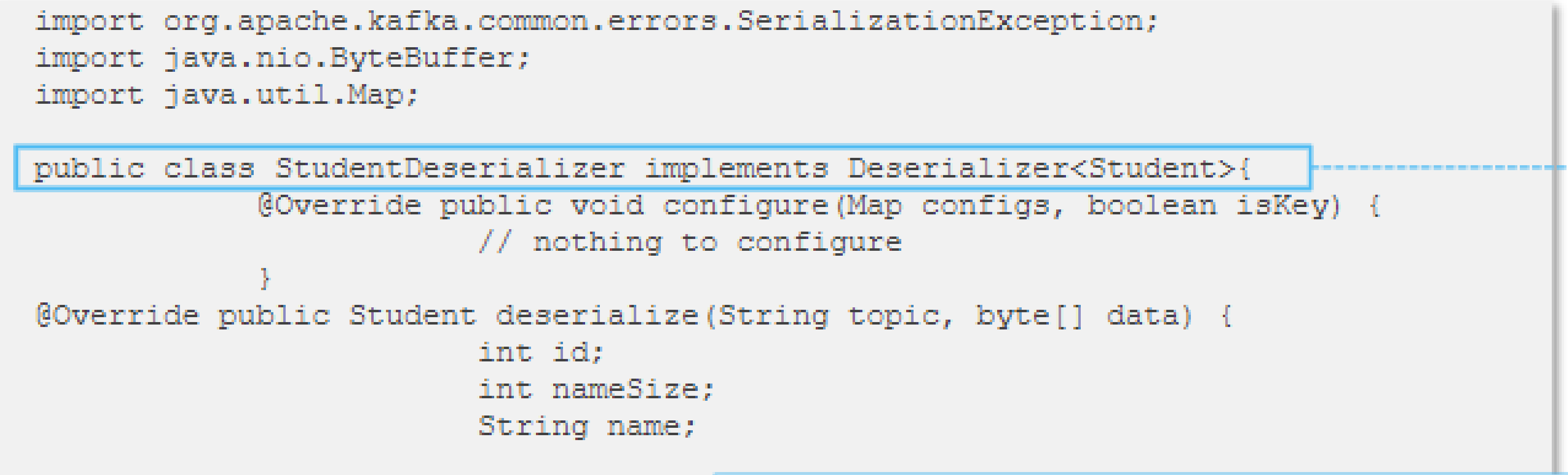
Creating a custom deserializer for this student class, which we used in producer



```
public class Student {
            private int studentID;
            private String studentName;

            public Student(int ID, String name) {
                        this.studentID = ID;
                        this.studentName = name; }

            public int getID() {
                        return studentID; }

            public String getName() {
                        return studentName; }
}
```

Custom Deserializers

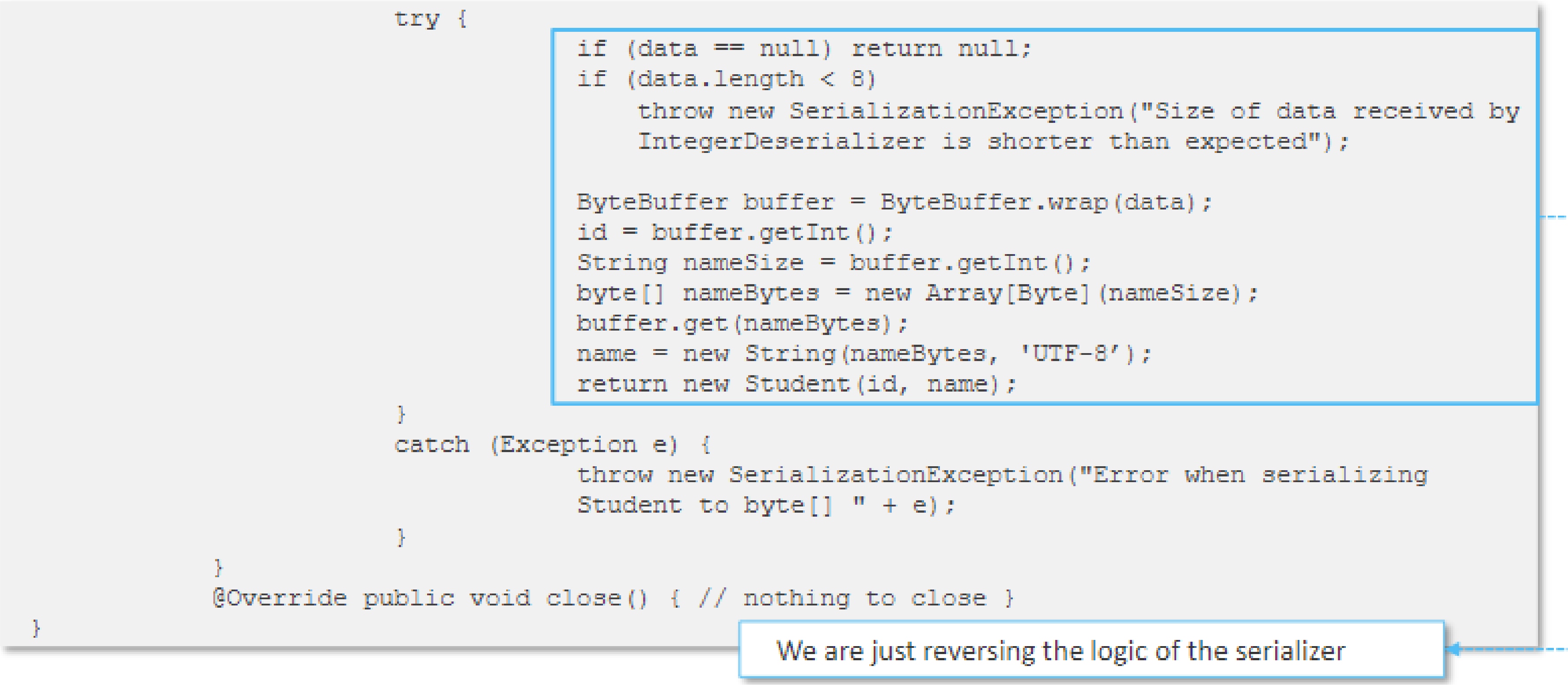
Creating a custom deserializer for student class



```
import org.apache.kafka.common.errors.SerializationException;
import java.nio.ByteBuffer;
import java.util.Map;

public class StudentDeserializer implements Deserializer<Student>{
            @Override public void configure(Map configs, boolean isKey) {
                        // nothing to configure
            }
@Override public Student deserialize(String topic, byte[] data) {
                        int id;
                        int nameSize;
                        String name;
```

The consumer needs the implementation of the Student class, & both, the class and & serializer need to match on the producing and consuming applications

Custom Deserializers



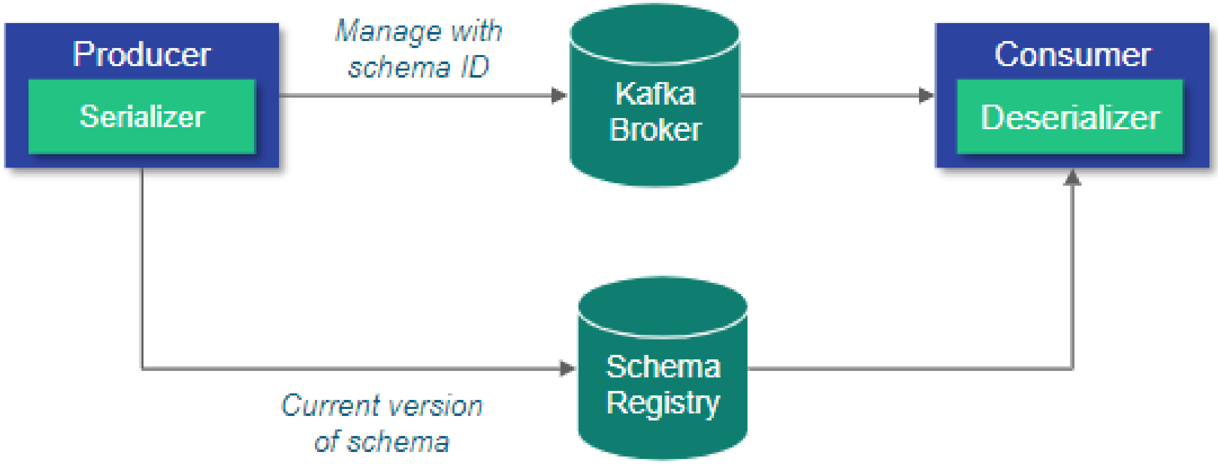
```
                        try {
                                  if (data == null) return null;
                                  if (data.length < 8)
                                      throw new SerializationException("Size of data received by
                                      IntegerDeserializer is shorter than expected");

                                  ByteBuffer buffer = ByteBuffer.wrap(data);
                                  id = buffer.getInt();
                                  String nameSize = buffer.getInt();
                                  byte[] nameBytes = new Array[Byte](nameSize);
                                  buffer.get(nameBytes);
                                  name = new String(nameBytes, 'UTF-8');
                                  return new Student(id, name);
                        }
                        catch (Exception e) {
                                  throw new SerializationException("Error when serializing
                                  Student to byte[] " + e);
                        }
              }
              @Override public void close() { // nothing to close }
}
```

We are just reversing the logic of the serializer

Avro Deserialization with Kafka Consumer

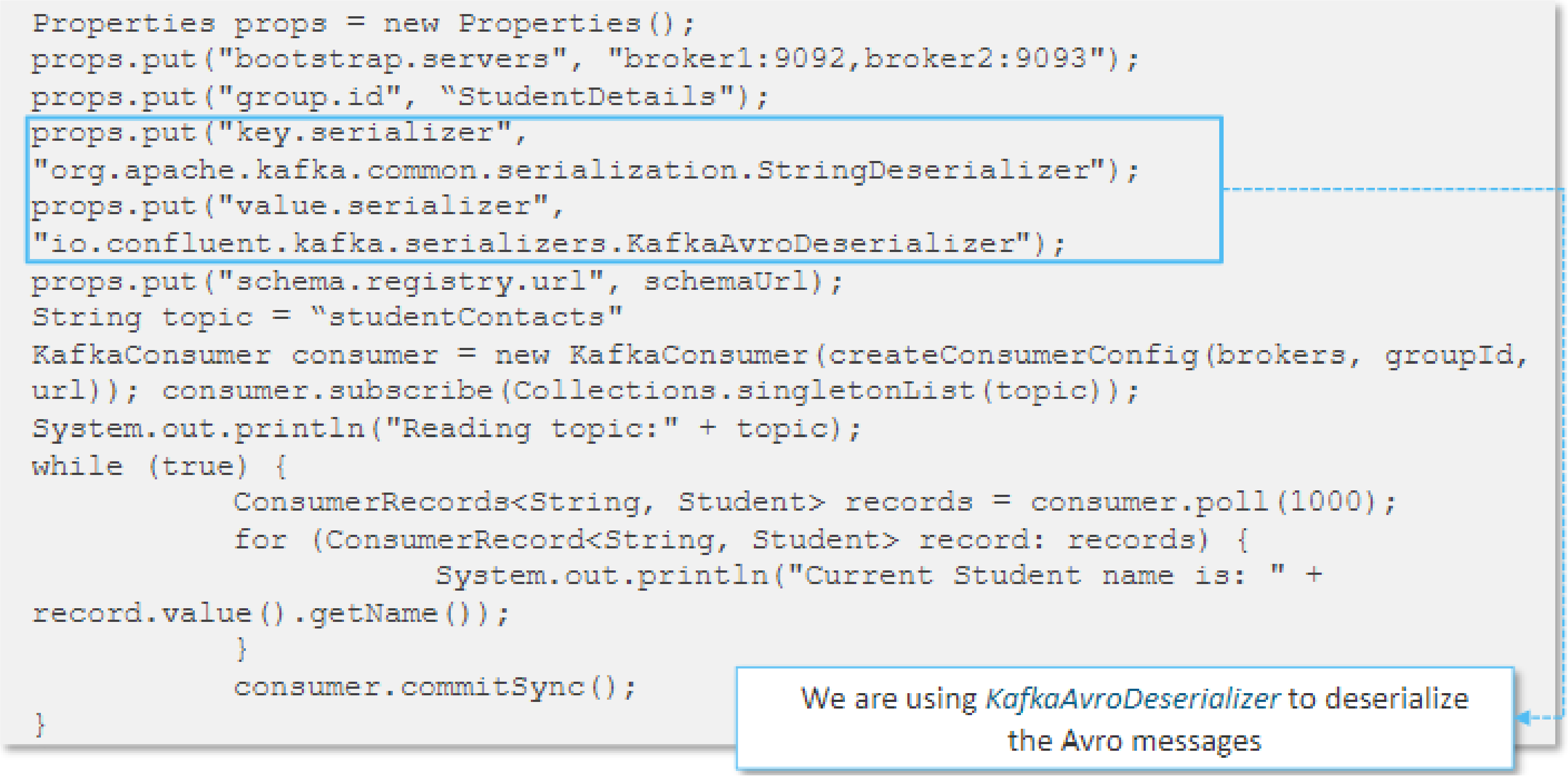
Using Avro Deserialization with Kafka Consumer

AvroSerializer makes sure that all the data written to a specific topic is compatible with the schema of the topic



☛ *Any errors in compatibility—on the producer or the consumer side—will be caught easily with an appropriate error message*

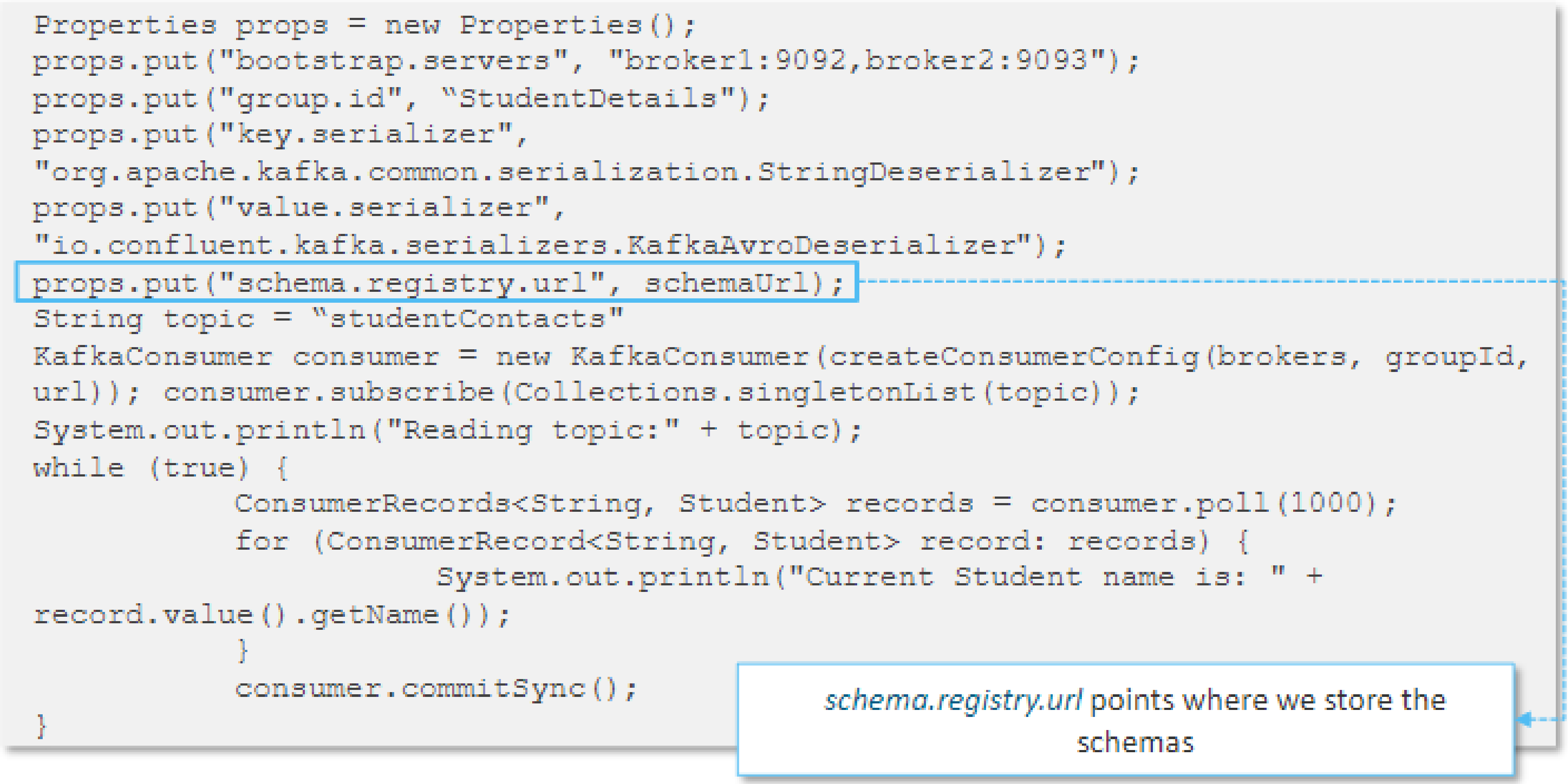
Using Avro Deserialization with Kafka Consumer



```
Properties props = new Properties();
props.put("bootstrap.servers", "broker1:9092,broker2:9093");
props.put("group.id", “StudentDetails");
props.put("key.serializer",
"org.apache.kafka.common.serialization.StringDeserializer");
props.put("value.serializer",
"io.confluent.kafka.serializers.KafkaAvroDeserializer");
props.put("schema.registry.url", schemaUrl);
String topic = “studentContacts"
KafkaConsumer consumer = new KafkaConsumer(createConsumerConfig(brokers, groupId,
url)); consumer.subscribe(Collections.singletonList(topic));
System.out.println("Reading topic:" + topic);
while (true) {
           ConsumerRecords<String, Student> records = consumer.poll(1000);
           for (ConsumerRecord<String, Student> record: records) {
                      System.out.println("Current Student name is: " +
record.value().getName());
           }
           consumer.commitSync();
}
```

We are using *KafkaAvroDeserializer* to deserialize the Avro messages

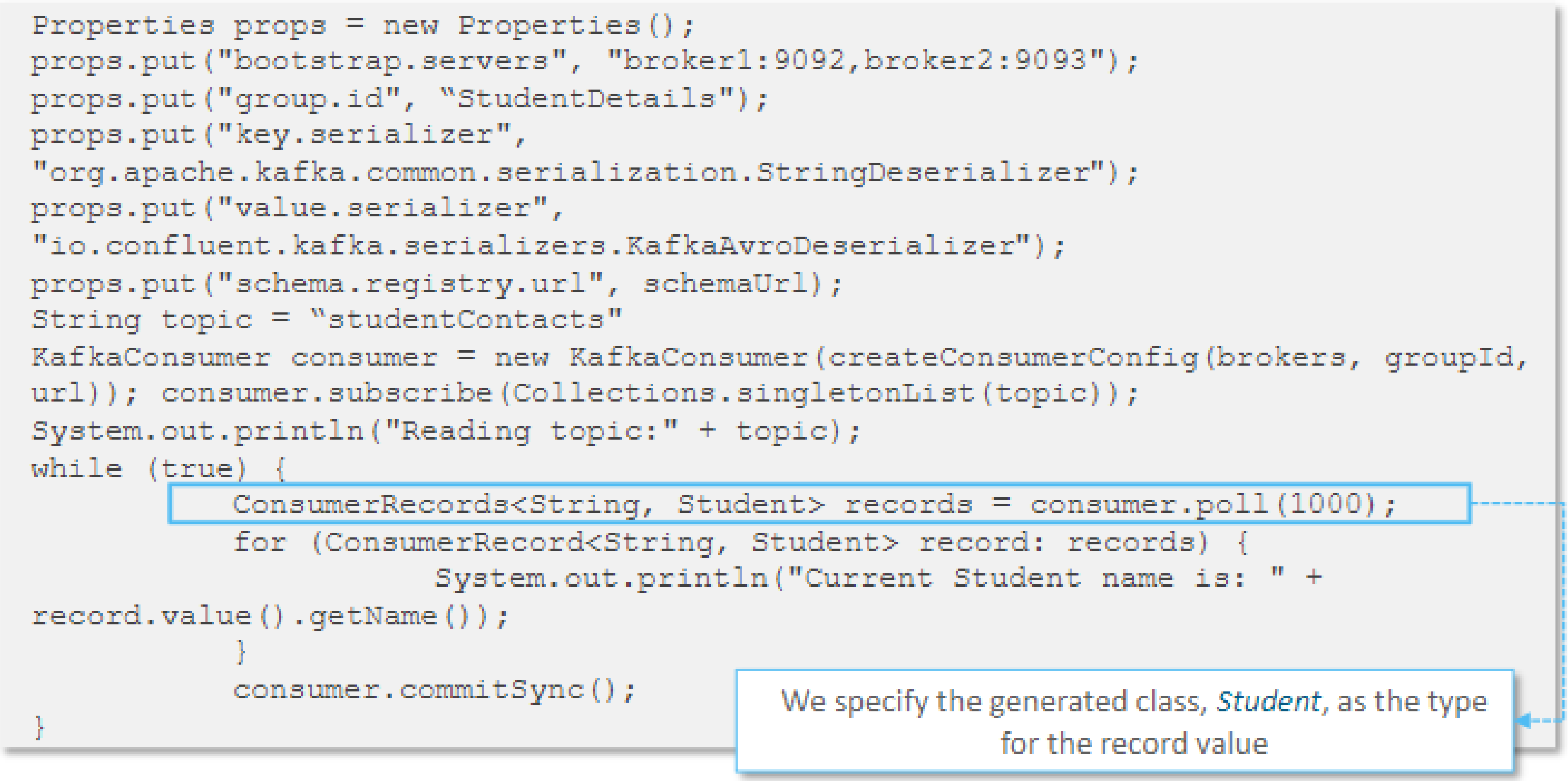
Using Avro Deserialization with Kafka Consumer



```
Properties props = new Properties();
props.put("bootstrap.servers", "broker1:9092,broker2:9093");
props.put("group.id", “StudentDetails");
props.put("key.serializer",
"org.apache.kafka.common.serialization.StringDeserializer");
props.put("value.serializer",
"io.confluent.kafka.serializers.KafkaAvroDeserializer");
props.put("schema.registry.url", schemaUrl);
String topic = “studentContacts"
KafkaConsumer consumer = new KafkaConsumer(createConsumerConfig(brokers, groupId,
url)); consumer.subscribe(Collections.singletonList(topic));
System.out.println("Reading topic:" + topic);
while (true) {
           ConsumerRecords<String, Student> records = consumer.poll(1000);
           for (ConsumerRecord<String, Student> record: records) {
                      System.out.println("Current Student name is: " +
record.value().getName());
           }
           consumer.commitSync();
}
```

schema.registry.url points where we store the schemas

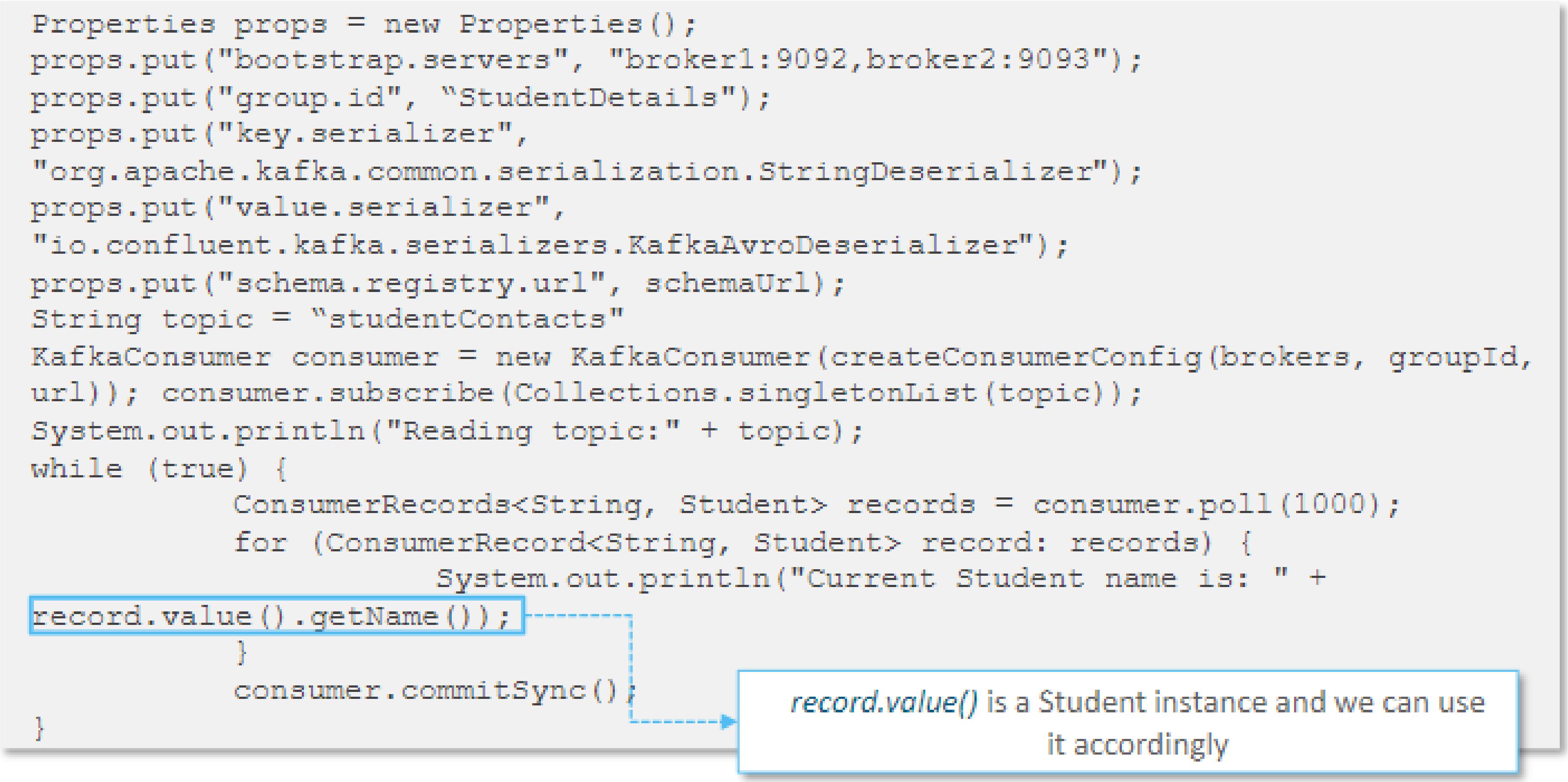
Using Avro Deserialization with Kafka Consumer



```
Properties props = new Properties();
props.put("bootstrap.servers", "broker1:9092,broker2:9093");
props.put("group.id", “StudentDetails");
props.put("key.serializer",
"org.apache.kafka.common.serialization.StringDeserializer");
props.put("value.serializer",
"io.confluent.kafka.serializers.KafkaAvroDeserializer");
props.put("schema.registry.url", schemaUrl);
String topic = “studentContacts"
KafkaConsumer consumer = new KafkaConsumer(createConsumerConfig(brokers, groupId,
url)); consumer.subscribe(Collections.singletonList(topic));
System.out.println("Reading topic:" + topic);
while (true) {
           ConsumerRecords<String, Student> records = consumer.poll(1000);
           for (ConsumerRecord<String, Student> record: records) {
                      System.out.println("Current Student name is: " +
record.value().getName());
           }
           consumer.commitSync();
}
```

We specify the generated class, *Student*, as the type for the record value

Using Avro Deserialization with Kafka Consumer



```
Properties props = new Properties();
props.put("bootstrap.servers", "broker1:9092,broker2:9093");
props.put("group.id", “StudentDetails");
props.put("key.serializer",
"org.apache.kafka.common.serialization.StringDeserializer");
props.put("value.serializer",
"io.confluent.kafka.serializers.KafkaAvroDeserializer");
props.put("schema.registry.url", schemaUrl);
String topic = “studentContacts"
KafkaConsumer consumer = new KafkaConsumer(createConsumerConfig(brokers, groupId,
url)); consumer.subscribe(Collections.singletonList(topic));
System.out.println("Reading topic:" + topic);
while (true) {
           ConsumerRecords<String, Student> records = consumer.poll(1000);
           for (ConsumerRecord<String, Student> record: records) {
                      System.out.println("Current Student name is: " +
record.value().getName());
           }
           consumer.commitSync();
}
```

record.value() is a Student instance and we can use it accordingly

TEKCRUX
We ReDefine IT