Spark

Lightning-Fast Cluster Computing

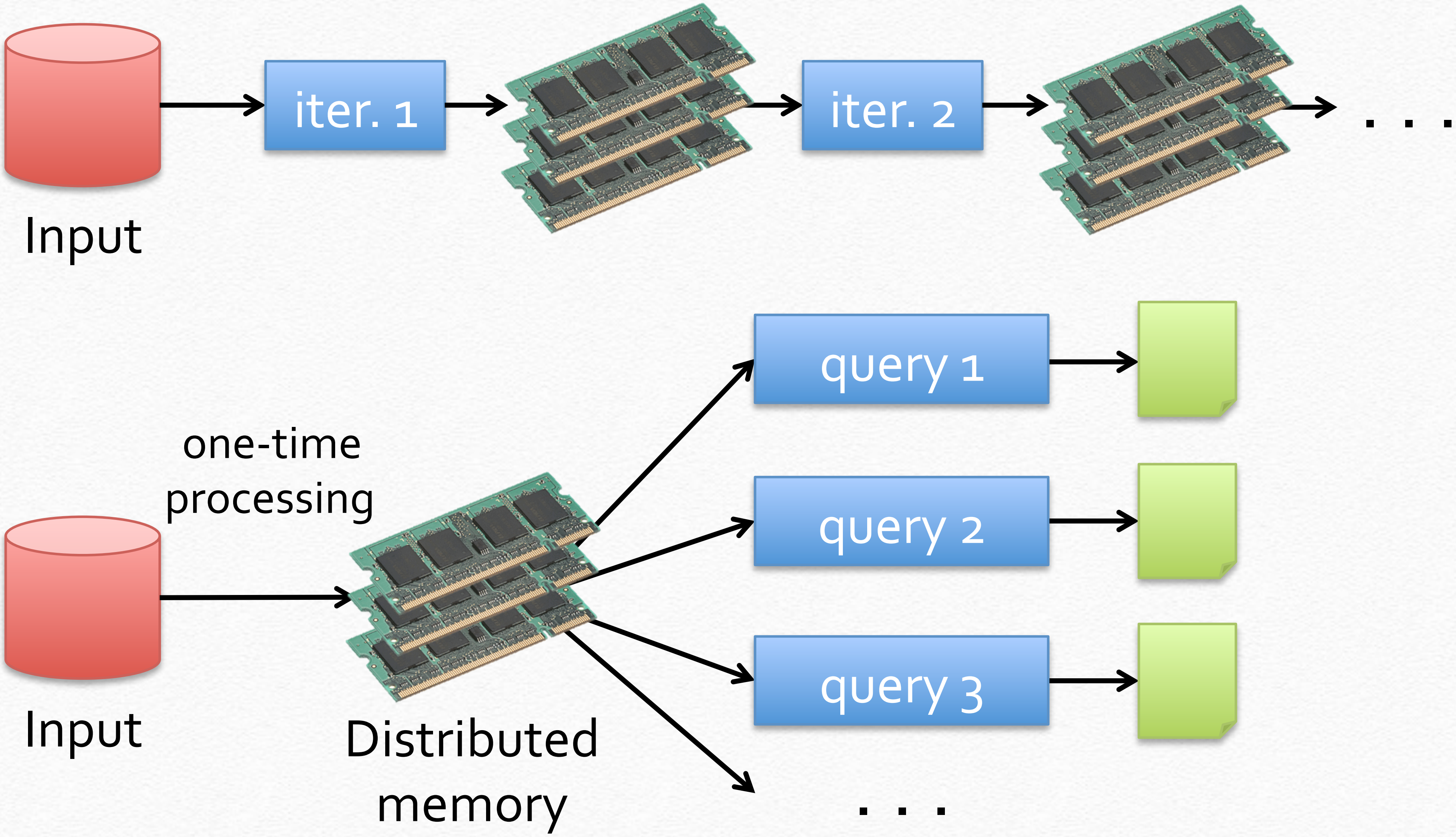
DataFrame and Streaming

13/05/2019 - Big Data 2019

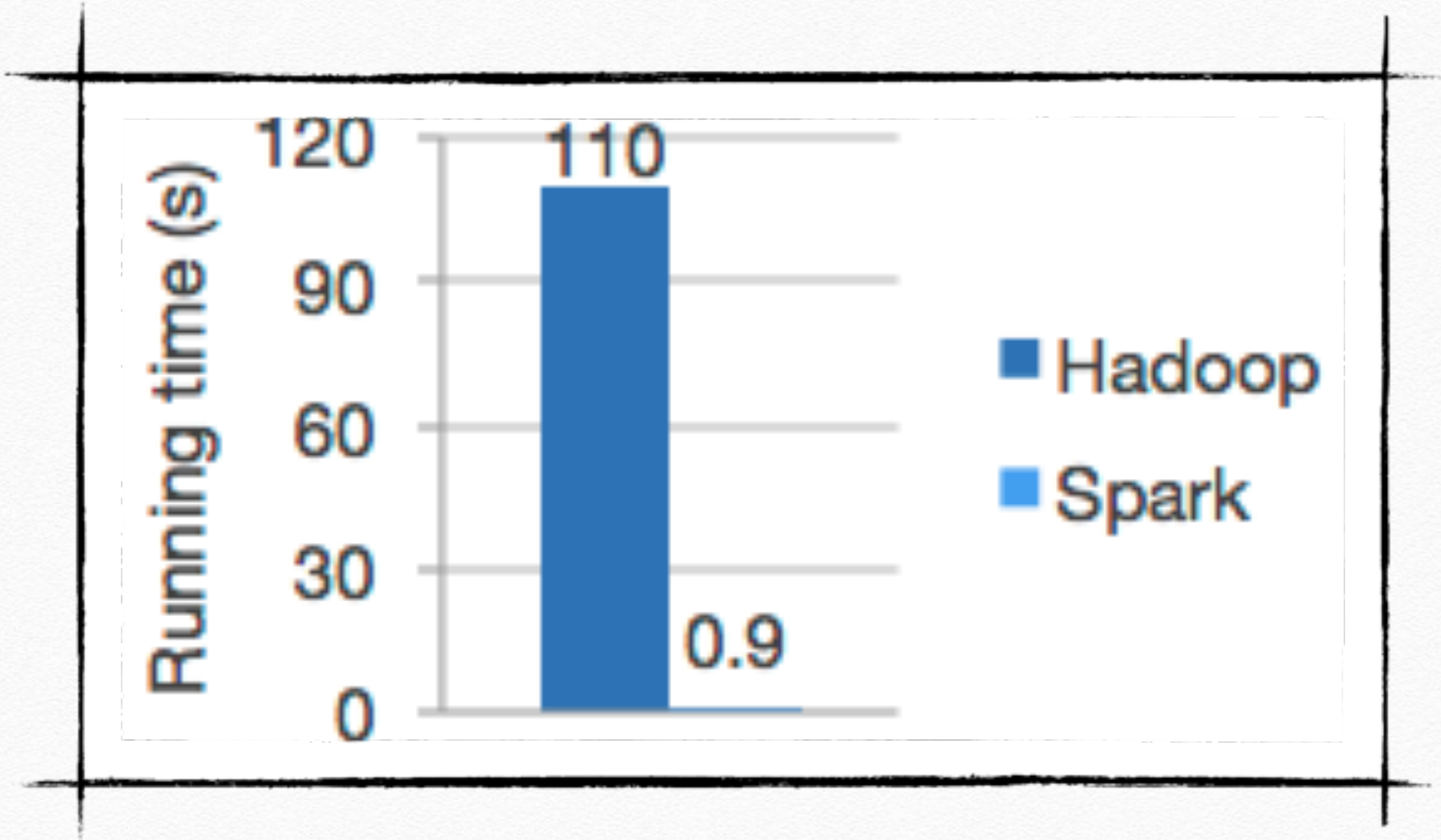
Apache Spark

- **Not** a modified version of **Hadoop**
- Separate, fast, **MapReduce-like** engine
  - ☑ **In-memory** data storage for very fast iterative queries
  - ☑ General **execution graphs** and powerful optimizations
  - ☑ Up to **40x faster** than Hadoop
- **Compatible** with Hadoop's storage APIs
  - ☑ Can read/write to any Hadoop-supported system, including HDFS, HBase, SequenceFiles, etc

Apache Spark



Apache Spark



10-100× faster than network and disk

Spark

Users

CONVIVA

foursquare

quantiFind

KLOUT

YAHOO! RESEARCH

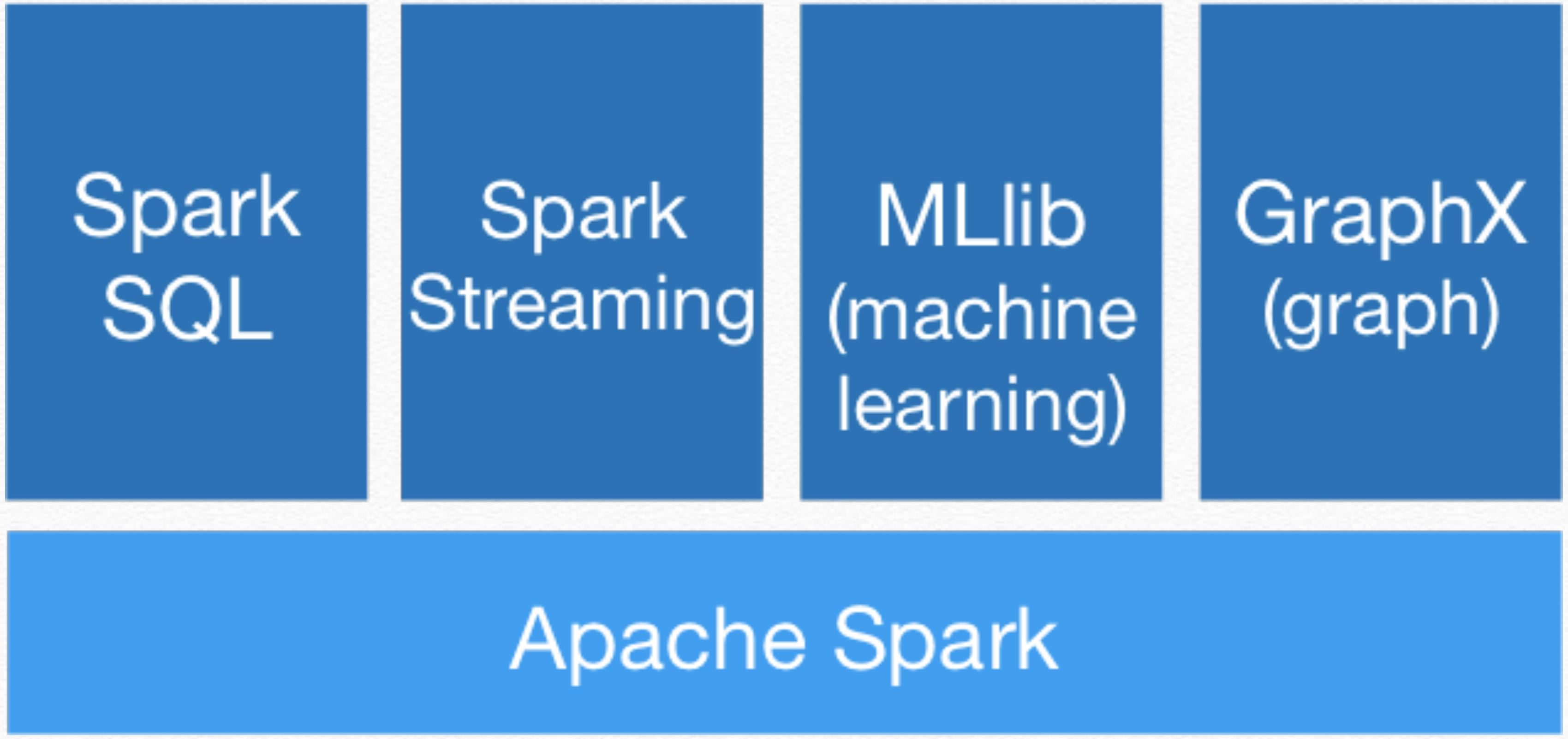
University of California Berkeley

PRINCETON UNIVERSITY

UCSF

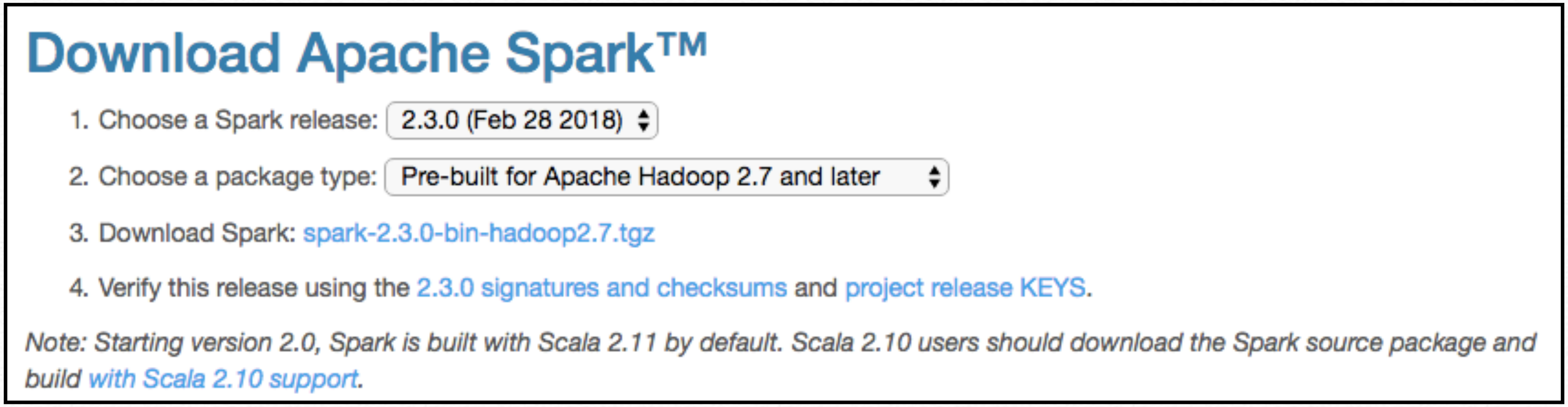
Spark

Combine SQL, streaming, and complex analytics



Spark Configuration

- Download a **binary release** of **apache Spark:**
- **spark-2.3.0-bin-hadoop2.7.tgz**



Download Apache Spark™

1. Choose a Spark release: 2.3.0 (Feb 28 2018)
2. Choose a package type: Pre-built for Apache Hadoop 2.7 and later
3. Download Spark: spark-2.3.0-bin-hadoop2.7.tgz
4. Verify this release using the 2.3.0 signatures and checksums and project release KEYS.

Note: Starting version 2.0, Spark is built with Scala 2.11 by default. Scala 2.10 users should download the Spark source package and build with Scala 2.10 support.

Spark Running

- Running Spark Shell [**scala**]:

```
$:~spark-*/bin/spark-shell
```

- Running Spark Shell [**python**]:

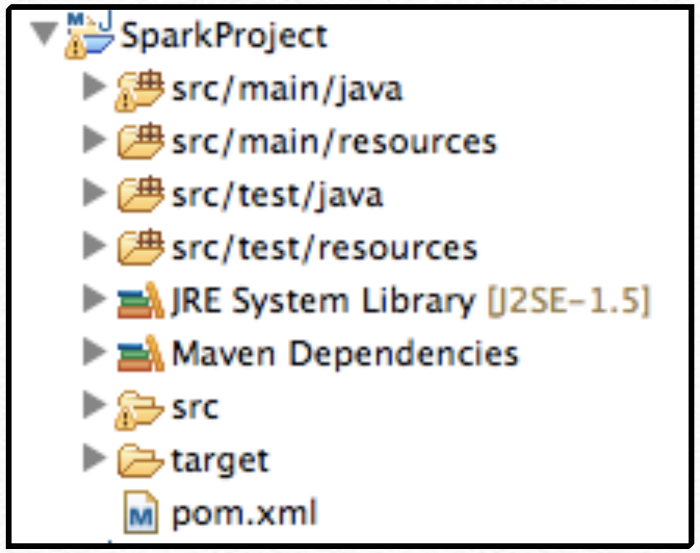
```
$:~spark-*/bin/pyspark
```

- **Spark Shell - Scala**

```
Welcome to
     ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 1.3.1
      /_/

Using Scala version 2.10.4 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_05)

Type in expressions to have them evaluated.

scala>
```

Spark Self-contained applications

- **Maven Project**



```
▼ SparkProject
    ▶ src/main/java
    ▶ src/main/resources
    ▶ src/test/java
    ▶ src/test/resources
    ▶ JRE System Library [J2SE-1.5]
    ▶ Maven Dependencies
    ▶ src
    ▶ target
      pom.xml
```

- **pom.xml**

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>edu.berkeley</groupId>
  <artifactId>simple-project</artifactId>
  <version>1.0</version>
  <dependencies>
    <dependency> <!-- Spark dependency -->
      <groupId>org.apache.spark</groupId>
      <artifactId>spark-core_2.10</artifactId>
      <version>1.6.1</version>
    </dependency>

    <dependency>
            <groupId>org.apache.spark</groupId>
            <artifactId>spark-streaming_2.10</artifactId>
            <version>1.6.1</version>
        </dependency>
        <dependency>
            <groupId>org.apache.spark</groupId>
            <artifactId>spark-sql_2.10</artifactId>
            <version>1.6.1</version>
        </dependency>
  </dependencies>
</project>
```

Spark Self-contained applications

- **SimpleApp.java**

create logData: an Object like [line1, line2, line3, ...]
sopra la panca la capra campa, sotto la panca la capra crepa

Lines with **a**: 1, lines with **b**: 0

Spark Self-contained applications

- **Java Spark API**

```
import org.apache.spark.api.java.*;
import org.apache.spark.SparkConf;
import org.apache.spark.api.java.function.Function;
public class SimpleApp {
  public static void main(String[] args) {
    String logFile = “data/simpleLog.txt”; // you can use “YOUR_SPARK_HOME/README.md”
    SparkConf conf = new SparkConf().setAppName("Simple Application");
    JavaSparkContext sc = new JavaSparkContext(conf);
    JavaRDD<String> logData = sc.textFile(logFile).cache();
    long numAs = logData.filter(new Function<String, Boolean>() {
      public Boolean call(String s) { return s.contains("a"); }
    }).count();
    long numBs = logData.filter(new Function<String, Boolean>() {
      public Boolean call(String s) { return s.contains("b"); }
    }).count();
    System.out.println("Lines with a: " + numAs + ", lines with b: " + numBs);
  }
}
```

Spark Self-contained applications

- **Java Spark API: configuration of Spark application**

```
String logFile = “data/simpleLog.txt”;

SparkConf conf = new SparkConf().setAppName("Simple Application");

JavaSparkContext sc = new JavaSparkContext(conf);

JavaRDD<String> logData = sc.textFile(logFile).cache();
```

Resilient Distributed Datasets (RDD) is a fundamental data structure of Spark. It is an immutable distributed collection of objects.

Spark Self-contained applications

- **Java Spark API: Spark actions**

```
long numAs = logData.filter(new Function<String, Boolean>() {
  public Boolean call(String s) { return s.contains("a"); }
}).count();
long numBs = logData.filter(new Function<String, Boolean>() {
  public Boolean call(String s) { return s.contains("b"); }
}).count();
System.out.println("Lines with a: " + numAs + ", lines with b: " + numBs);
```

Spark Running - standalone

- Running Java Spark applications:

```
$:~spark-*/bin/spark-submit

               --class "SimpleApp"

               --master local[4]

               SparkProject-1.0.jar
```

local to run locally with one thread, or local[N] to run locally with N threads.

- **output [terminal] - using simpleLog.txt**

```
Lines with a: 1, Lines with b: 0
```

Spark Running - standalone

- Running Java Spark applications:

```
$:~spark-*/bin/spark-submit
               --class "SimpleApp"
               --master local[4]
               SparkProject-1.0.jar
```

local to run locally with one thread, or local[N] to run locally with N threads.

- **output [terminal] - using README.md**

```
Lines with a: 46, Lines with b: 23
```

Spark Running - YARN

- Running Java Spark applications:

```
$:~spark-*/bin/spark-submit

               --class "SimpleApp"

               --master yarn

               SparkProject-1.0.jar
```

The --master option allows to specify the master URL for a distributed cluster

- **output [terminal] - using README.md**

```
Lines with a: 46, Lines with b: 23
```

Spark Running - AWS

- Look at

http://docs.aws.amazon.com/emr/latest/ReleaseGuide/emr-spark-launch.html

Lambda Expressions

- You're usually trying to pass functionality as an argument to another method, such as what action should be taken when someone clicks a button.
- Lambda expressions enable you to do this, to treat functionality as method argument, or code as data

Lambda Expressions

- Suppose that you are creating a social networking application.

- You want to create a feature that enables an administrator to perform **any kind of action**, such as *sending a message, on members of the social networking application that satisfy certain criteria*.

- Suppose that members of this social networking application are represented by the following **Person** class:

```
public class Person {

    public enum Sex {
        MALE, FEMALE
    }

    String name;
    LocalDate birthday;
    Sex gender;
    String emailAddress;

    public int getAge() {
        // ...
    }

    public void printPerson() {
        // ...
    }
}
```

Lambda Expressions

- Suppose that the members of your social networking application are stored in a List<Person> instance

- Approach 1: Create Methods That Search for Members That Match One Characteristic

```
public static void printPersonsOlderThan(List<Person> roster, int age)
{
    for (Person p : roster) {
        if (p.getAge() >= age) {
            p.printPerson();
        }
    }
}
```

Lambda Expressions

- Suppose that the members of your social networking application are stored in a List<Person> instance

- Approach 2: Create More Generalized Search Methods

```
public static void printPersonsWithinAgeRange(List<Person> roster,
                                              int low, int high) {
    for (Person p : roster) {
        if (low <= p.getAge() && p.getAge() < high) {
            p.printPerson();
        }
    }
}
```

Lambda Expressions

- Approach 3: Specify Search Criteria Code in a Local Class

```
public static void printPersons(List<Person> roster,
                                CheckPerson tester) {
    for (Person p : roster) {
        if (tester.test(p)) {
            p.printPerson();
        }
    }
}

interface CheckPerson {
    boolean test(Person p);
}

class CheckPersonEligibleForSelectiveService implements CheckPerson {
    public boolean test(Person p) {
        return p.gender == Person.Sex.MALE &&
            p.getAge() >= 18 &&
            p.getAge() <= 25;
    }
}
```

Lambda Expressions

- Approach 4: Specify Search Criteria Code in an Anonymous Class

```
printPersons(
    roster,
    new CheckPerson() {
        public boolean test(Person p) {
            return p.getGender() == Person.Sex.MALE
                && p.getAge() >= 18
                && p.getAge() <= 25;
        }
    }
);
```

Lambda Expressions

- Approach 5: Specify Search Criteria Code with a Lambda Expression

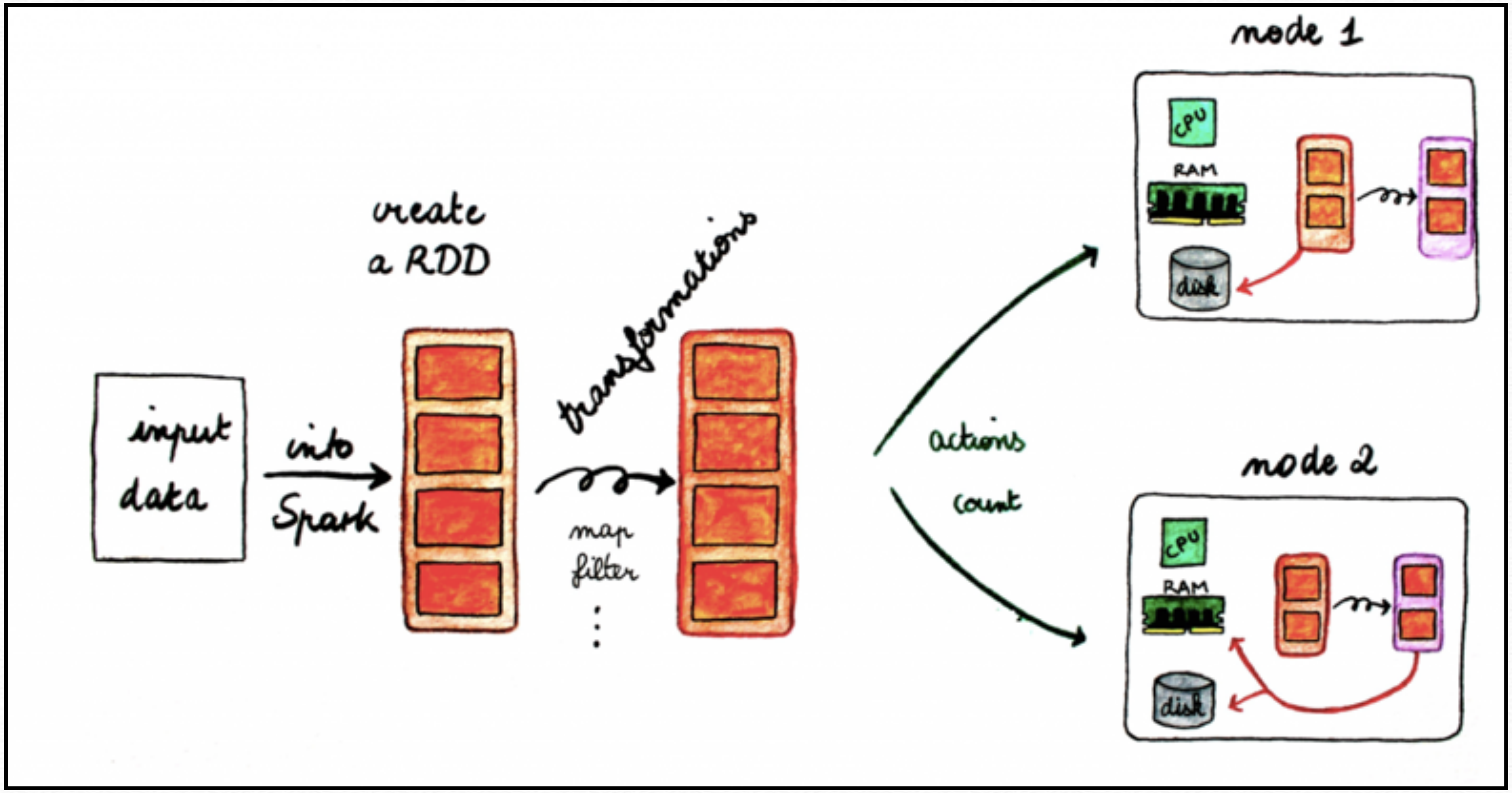
```
printPersons(
    roster,
    (Person p) -> p.getGender() == Person.Sex.MALE
        && p.getAge() >= 18
        && p.getAge() <= 25
);
```

Exercises

- http://dia.uniroma3.it/~dvr/es_4.zip

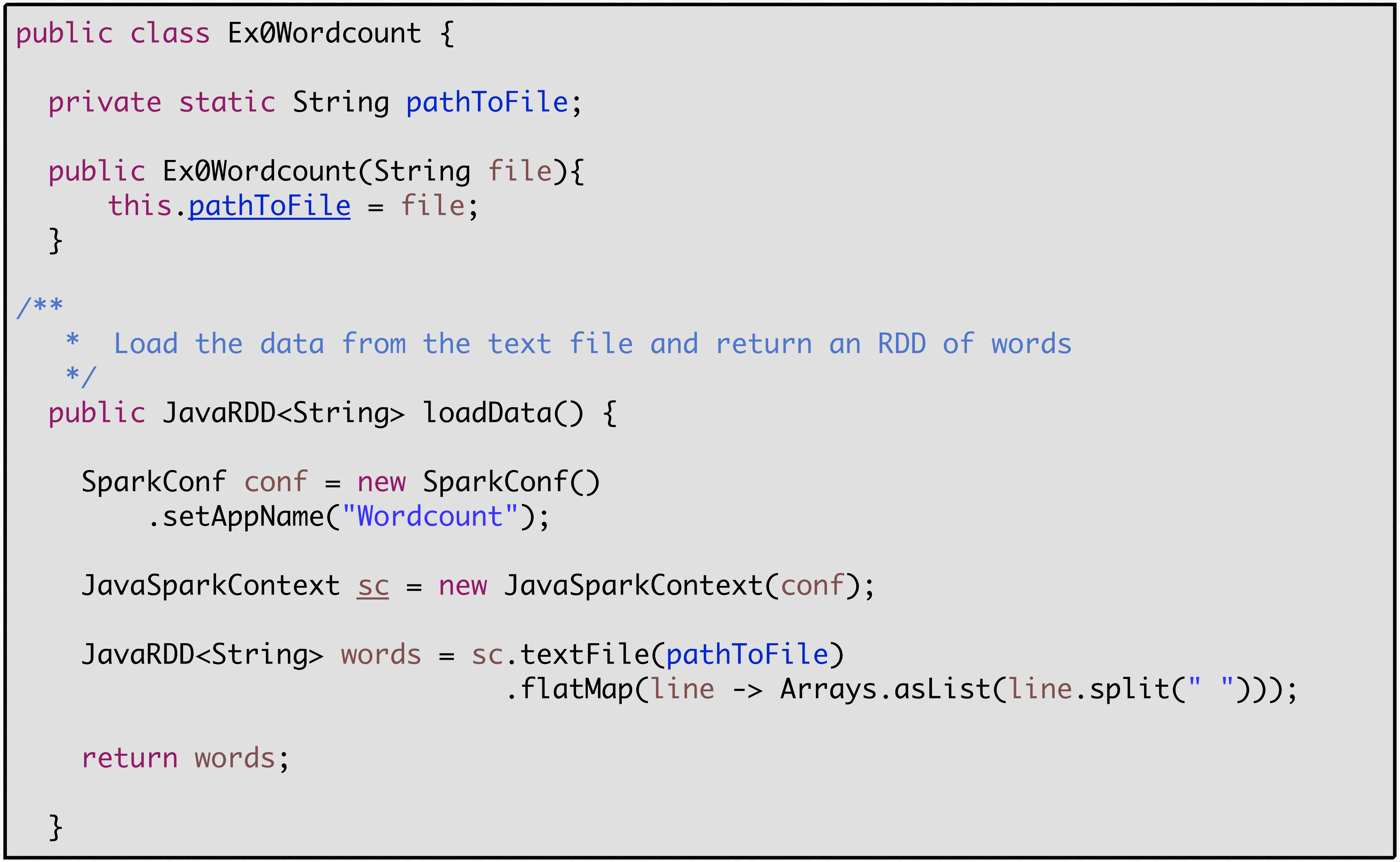
Exercises

- **SPARK Core API**



Exercises

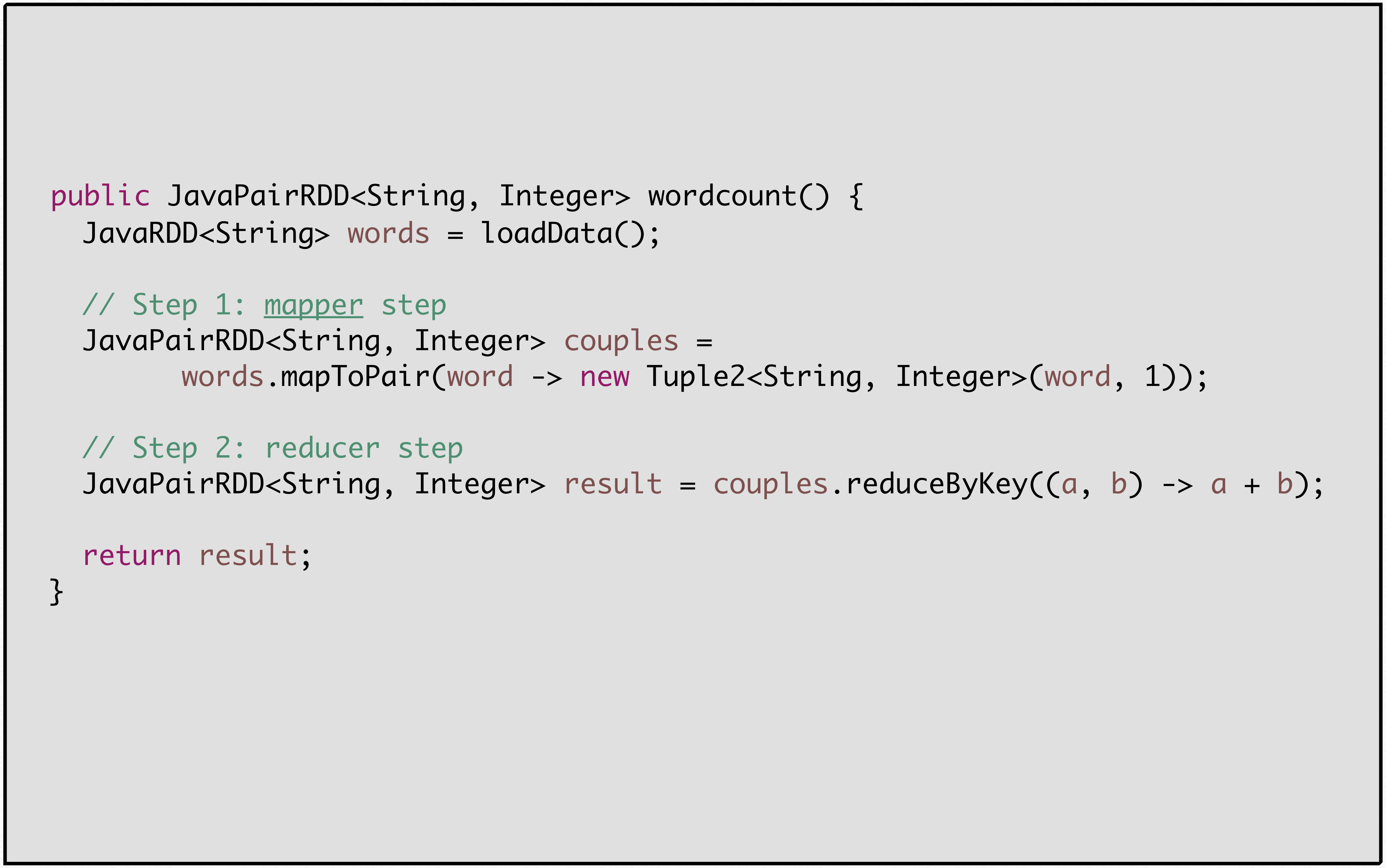
SPARK Core API: Word Count



```
public class Ex0Wordcount {

  private static String pathToFile;

  public Ex0Wordcount(String file){
     this.pathToFile = file;
  }

/**
   *  Load the data from the text file and return an RDD of words
   */
  public JavaRDD<String> loadData() {

    SparkConf conf = new SparkConf()
        .setAppName("Wordcount");

    JavaSparkContext sc = new JavaSparkContext(conf);

    JavaRDD<String> words = sc.textFile(pathToFile)
                                          .flatMap(line -> Arrays.asList(line.split(" ")));

    return words;

  }
```

Exercises

SPARK Core API: Word Count



```java
public JavaPairRDD<String, Integer> wordcount() {
  JavaRDD<String> words = loadData();

  // Step 1: mapper step
  JavaPairRDD<String, Integer> couples =
        words.mapToPair(word -> new Tuple2<String, Integer>(word, 1));

  // Step 2: reducer step
  JavaPairRDD<String, Integer> result = couples.reduceByKey((a, b) -> a + b);

  return result;
}
```

Exercises

SPARK Core API: Word Count

```
/**
   *  Now just keep the word which appear strictly more than x times!
   */


  public JavaPairRDD<String, Integer> filterOnWordcount(int x) {
    JavaPairRDD<String, Integer> wordcounts = wordcount();

    JavaPairRDD<String, Integer> filtered =
                             wordcounts.filter(couple -> couple._2() > x);

    return filtered;

 }
```

Exercises

SPARK Core API: Word Count

```
public static void main(String[] args) {
      // TODO Auto-generated method stub

      if (args.length < 1) {
            System.err.println("Usage: Ex0Wordcount <filetxt>");
            System.exit(1);
          }

      Ex0Wordcount wc = new Ex0Wordcount(args[0]);

      System.out.println("wordcount: "+wc.wordcount().toString());

  }
```

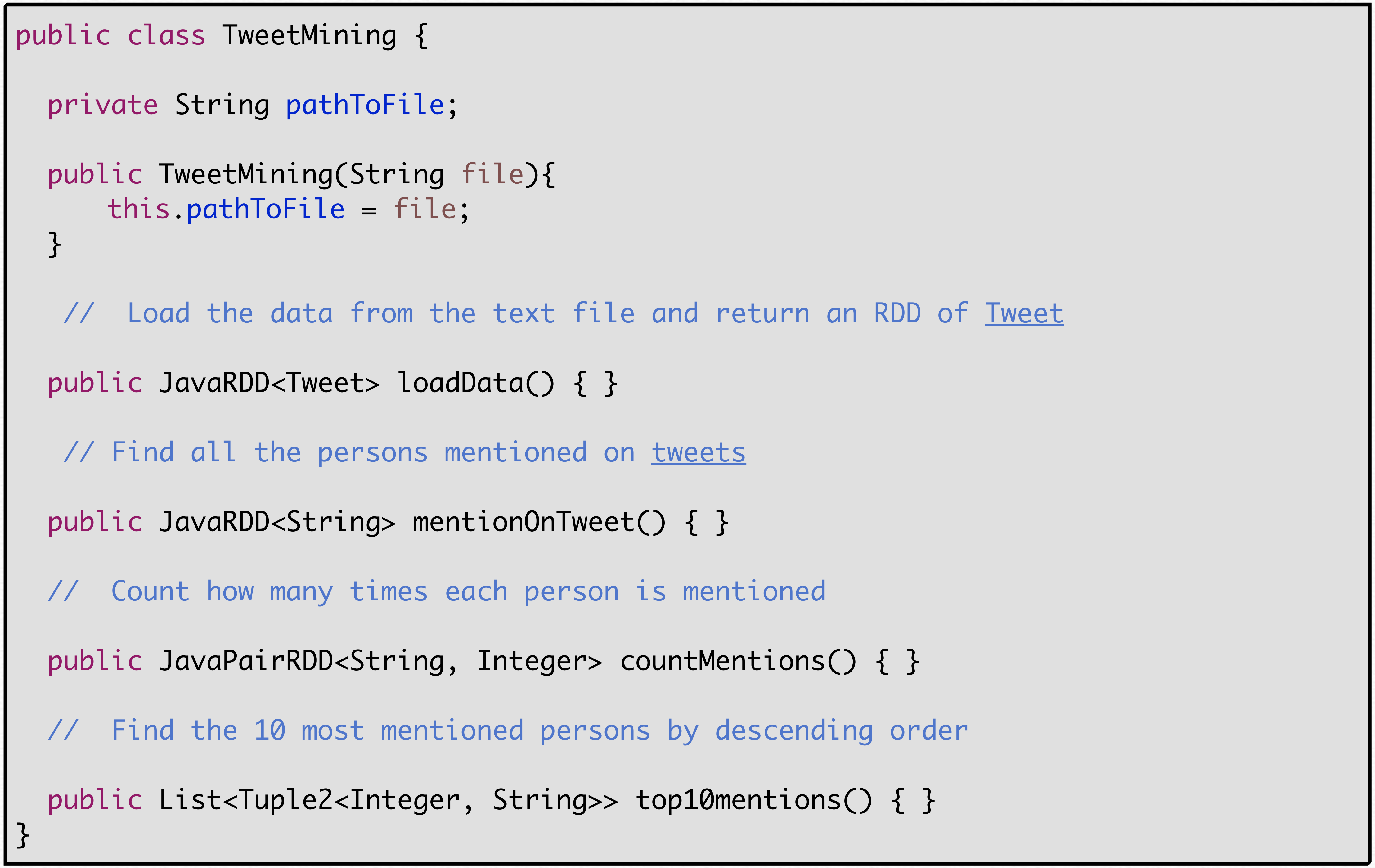
Exercises

SPARK Core API: Tweet Mining

```
Now we use a dataset with 8198 tweets.
Here an example of a tweet (json):

   {"id":"572692378957430785",
    "user":"Srkian_nishu :)",
    "text":"@always_nidhi @YouTube no i dnt understand bt i
             loved of this mve is rocking",
    "place":"Orissa",
    “country":"India"
    }

   We want to make some computations on the tweets:
   - Find all the persons mentioned on tweets
   - Count how many times each person is mentioned
   - Find the 10 most mentioned persons by descending order
```

Exercises

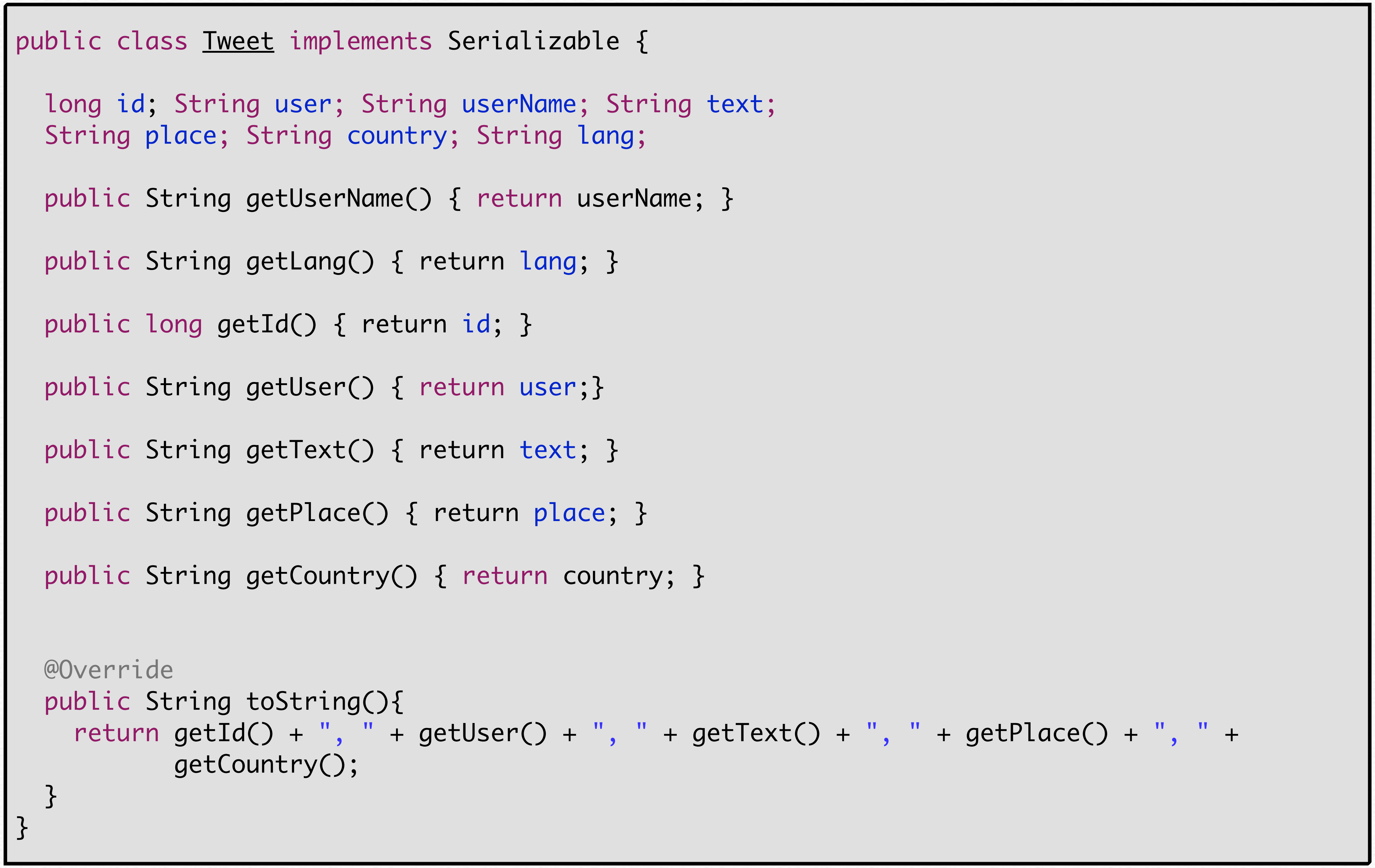
SPARK Core API: Tweet Mining



```
public class TweetMining {

  private String pathToFile;

  public TweetMining(String file){
      this.pathToFile = file;
  }

   //  Load the data from the text file and return an RDD of Tweet

  public JavaRDD<Tweet> loadData() { }

   // Find all the persons mentioned on tweets

  public JavaRDD<String> mentionOnTweet() { }

  //  Count how many times each person is mentioned

  public JavaPairRDD<String, Integer> countMentions() { }

  //  Find the 10 most mentioned persons by descending order

  public List<Tuple2<Integer, String>> top10mentions() { }
}
```

Exercises

SPARK Core API: Tweet Mining



```
public class Tweet implements Serializable {

  long id; String user; String userName; String text;
  String place; String country; String lang;

  public String getUserName() { return userName; }

  public String getLang() { return lang; }

  public long getId() { return id; }

  public String getUser() { return user;}

  public String getText() { return text; }

  public String getPlace() { return place; }

  public String getCountry() { return country; }


  @Override
  public String toString(){
    return getId() + ", " + getUser() + ", " + getText() + ", " + getPlace() + ", " +
           getCountry();
  }
}
```

Exercises

SPARK Core API: Tweet Mining

```java
import com.fasterxml.jackson.databind.ObjectMapper;

public class Parse {

  public static Tweet parseJsonToTweet(String jsonLine) {

    ObjectMapper objectMapper = new ObjectMapper();
    Tweet tweet = null;

    try {
      tweet = objectMapper.readValue(jsonLine, Tweet.class);
    } catch (IOException e) {
      e.printStackTrace();
    }
    return tweet;
  }

}
```

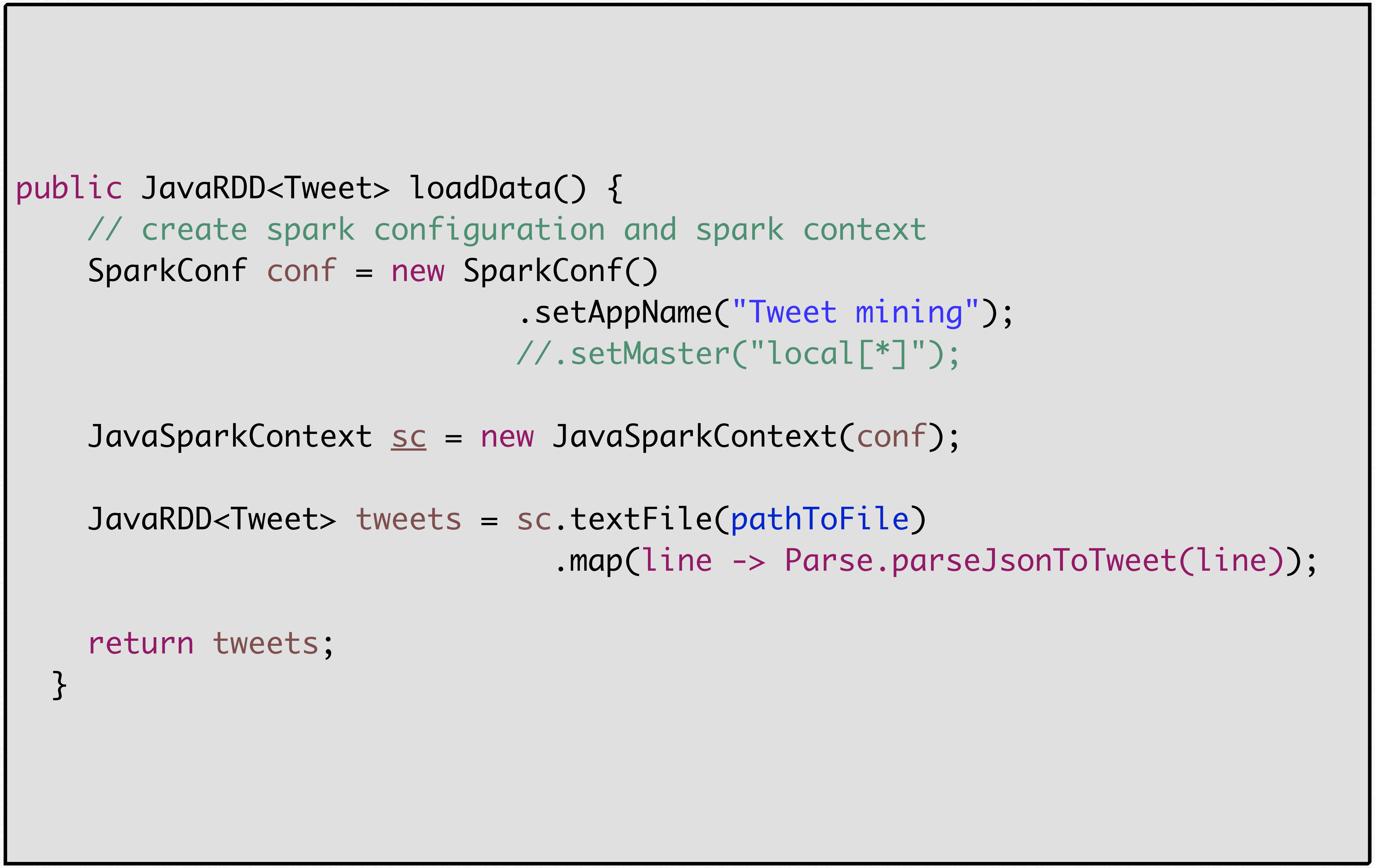
Exercises

SPARK Core API: Tweet Mining (Java 1.7 or later)

```
public JavaRDD<Tweet> loadData() {
    // create spark configuration and spark context
    SparkConf conf = new SparkConf()
                            .setAppName("Tweet mining");
                            //.setMaster("local[*]");

    JavaSparkContext sc = new JavaSparkContext(conf);

    JavaRDD<Tweet> tweets = sc.textFile(pathToFile)
                              .map(new Function<String, Tweet>() {

                          @Override
                          public Tweet call(String line) throws Exception
{
                              return Parse.parseJsonToTweet(line);
                          }
                       });

    return tweets;
  }
```

Exercises

SPARK Core API: Tweet Mining (LAMBDA Java 1.8)



```java
public JavaRDD<Tweet> loadData() {
    // create spark configuration and spark context
    SparkConf conf = new SparkConf()
                            .setAppName("Tweet mining");
                            //.setMaster("local[*]");

    JavaSparkContext sc = new JavaSparkContext(conf);

    JavaRDD<Tweet> tweets = sc.textFile(pathToFile)
                              .map(line -> Parse.parseJsonToTweet(line));

    return tweets;
  }
```

Exercises

SPARK Core API: Tweet Mining (Java 1.7 or later)

```
  - Find all the persons mentioned on tweets

public JavaRDD<String> mentionOnTweet() {
    JavaRDD<Tweet> tweets = loadData();

    JavaRDD<String> mentions = tweets.flatMap(new FlatMapFunction<Tweet,
String>() {
       @Override
       public Iterable<String> call(Tweet tweet) throws Exception {
          return Arrays.asList(tweet.getText().split(" "));
       }
   })
        .filter(new Function<String, Boolean>() {
                        @Override
                        public Boolean call(String word) throws Exception {
                        return word.startsWith("@") && word.length() > 1;
                                   }
                                });

    System.out.println("mentions.count() " + mentions.count());
    return mentions;
  }
```

Exercises

SPARK Core API: Tweet Mining (Java 1.8)

```
    - Find all the persons mentioned on tweets


public JavaRDD<String> mentionOnTweet() {
    JavaRDD<Tweet> tweets = loadData();

     JavaRDD<String> mentions =
          tweets.flatMap(tweet -> Arrays.asList(tweet.getText()
                .split(" ")))
                .filter(word -> word.startsWith("@") && word.length() > 1);

    System.out.println("mentions.count() " + mentions.count());
    return mentions;

  }
```

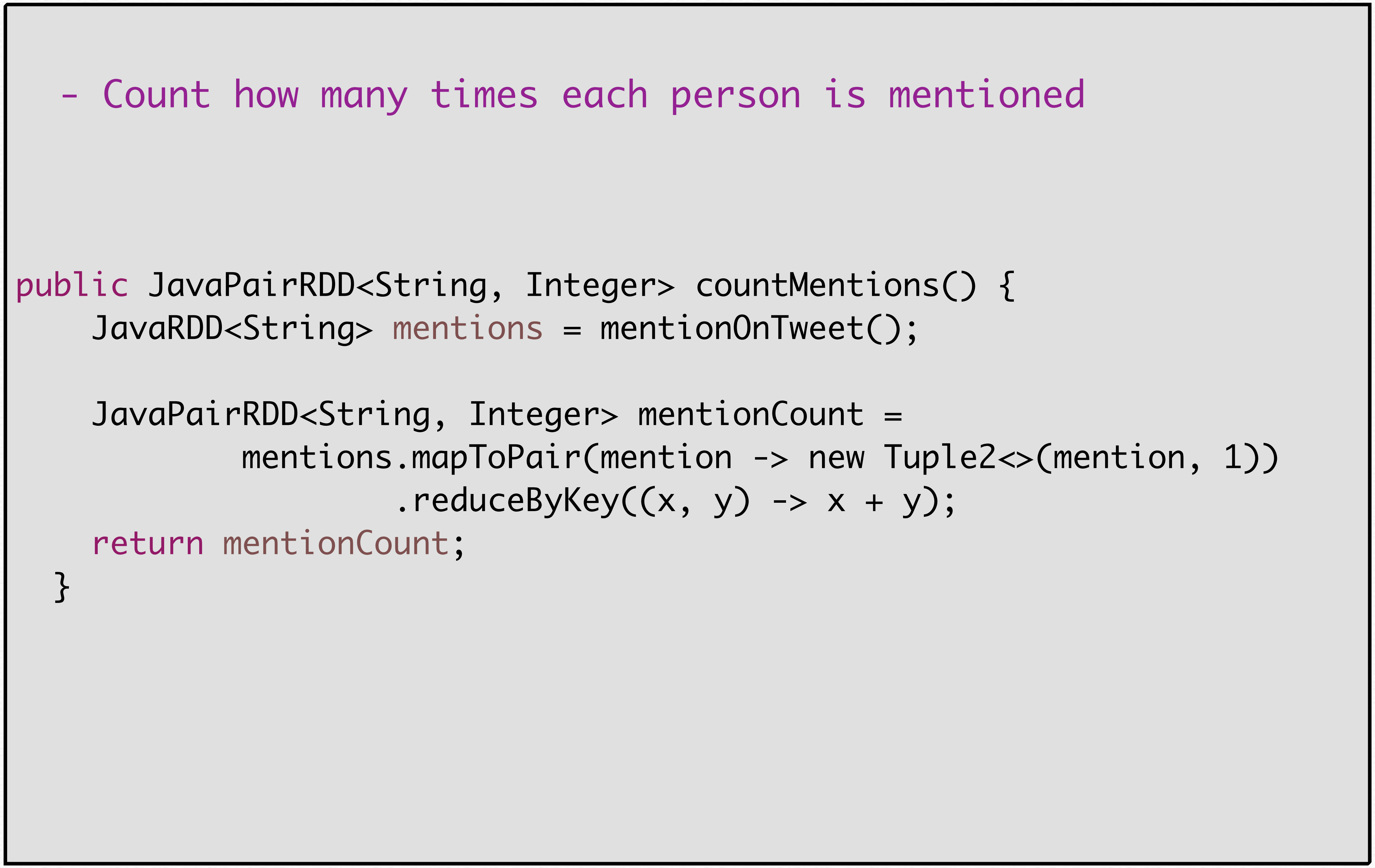
Exercises

SPARK Core API: Tweet Mining (Java 1.7 or later)

```
   - Count how many times each person is mentioned

public JavaPairRDD<String, Integer> countMentions() {
    JavaRDD<String> mentions = mentionOnTweet();

    JavaPairRDD<String, Integer> mentionCount = mentions.mapToPair(new
PairFunction<String, String, Integer>() {
        @Override
        public Tuple2<String, Integer> call(String mention) throws Exception {
            return new Tuple2<>(mention, 1);
        }
    })
         .reduceByKey(new Function2<Integer, Integer, Integer>() {
               @Override
               public Integer call(Integer x, Integer y) throws Exception {
                      return x + y;
                                                        }
                                                    });

    return mentionCount;
  }
```

Exercises

SPARK Core API: Tweet Mining (Java 1.8)



```
  - Count how many times each person is mentioned



public JavaPairRDD<String, Integer> countMentions() {
    JavaRDD<String> mentions = mentionOnTweet();

    JavaPairRDD<String, Integer> mentionCount =
            mentions.mapToPair(mention -> new Tuple2<>(mention, 1))
                    .reduceByKey((x, y) -> x + y);
    return mentionCount;
  }
```

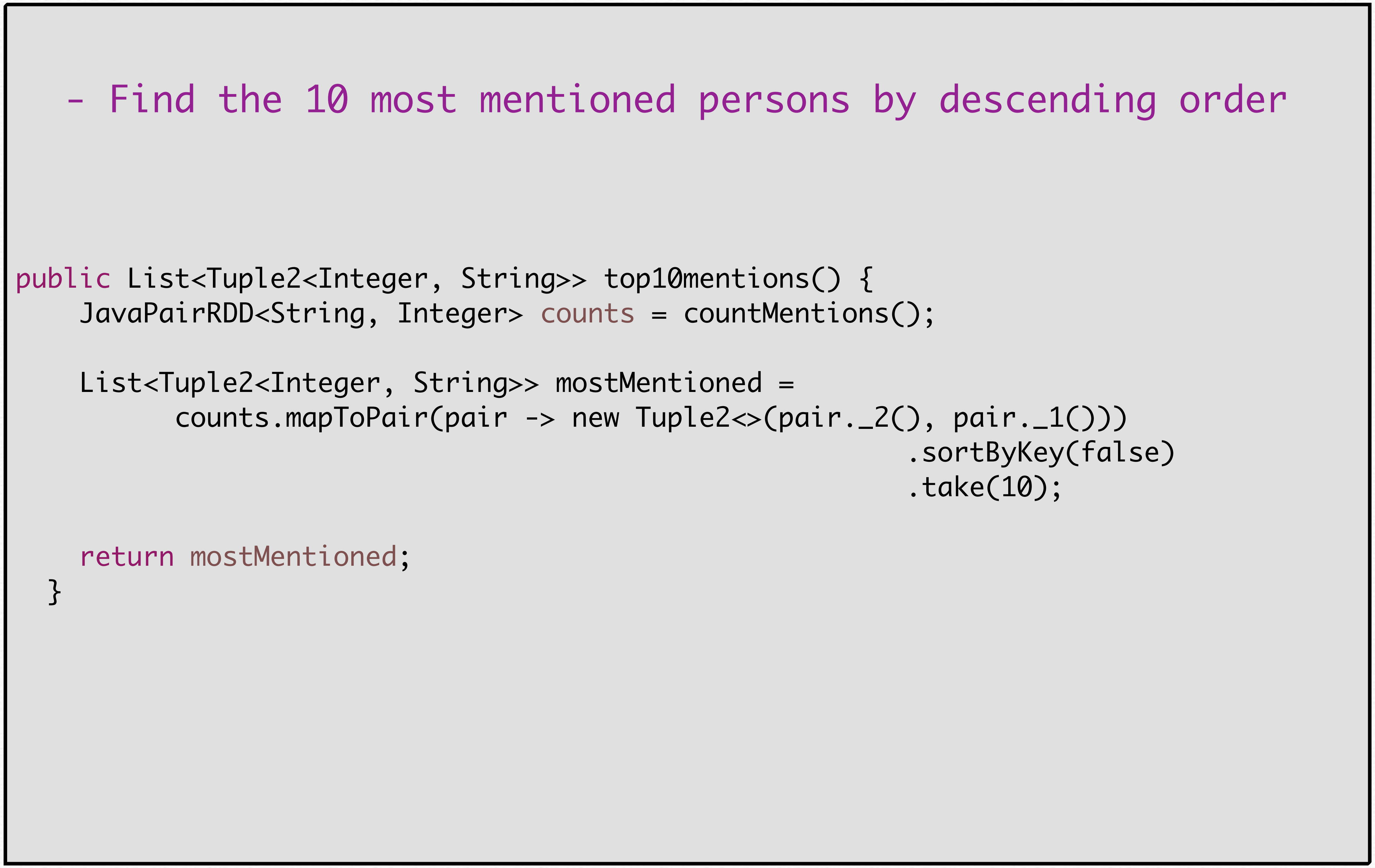
Exercises

- **SPARK Core API: Tweet Mining (Java 1.7 or later)**

```
  - Find the 10 most mentioned persons by descending order


public List<Tuple2<Integer, String>> top10mentions() {
    JavaPairRDD<String, Integer> counts = countMentions();

    List<Tuple2<Integer, String>> mostMentioned =
         counts.mapToPair(new PairFunction<Tuple2<String, Integer>, Integer,
String>() {
                                @Override
       public Tuple2<Integer, String> call(Tuple2<String, Integer> pair) throws
Exception {
          return new Tuple2<>(pair._2(), pair._1());
       }
    })
                                                        .sortByKey(false)
                                                        .take(10);


    return mostMentioned;
  }
```

Exercises

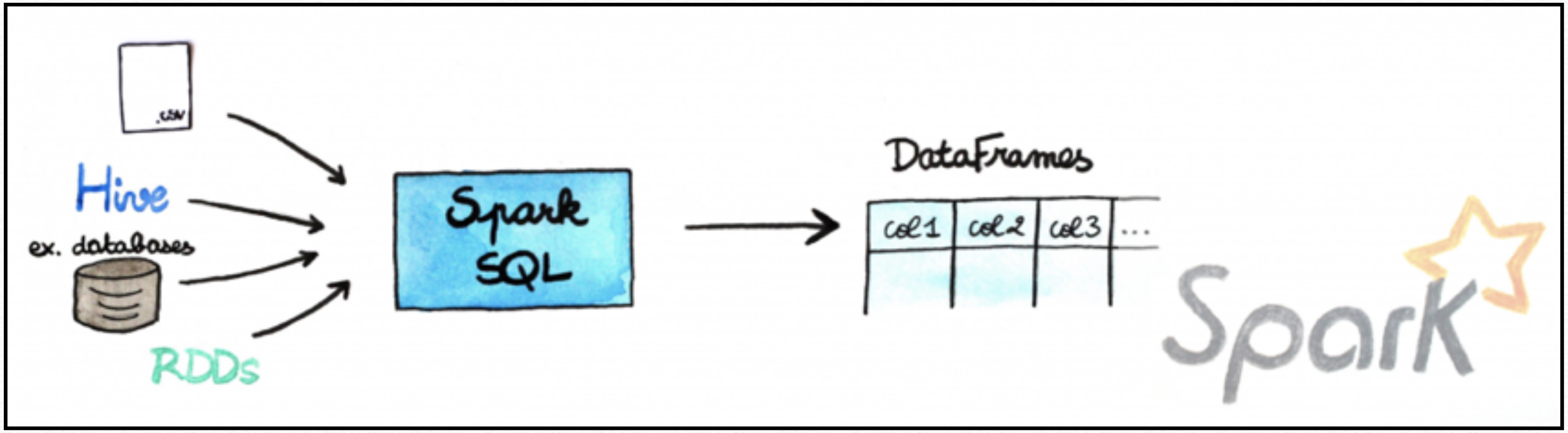
SPARK Core API: Tweet Mining (Java 1.8)



```
   - Find the 10 most mentioned persons by descending order



public List<Tuple2<Integer, String>> top10mentions() {
    JavaPairRDD<String, Integer> counts = countMentions();

    List<Tuple2<Integer, String>> mostMentioned =
          counts.mapToPair(pair -> new Tuple2<>(pair._2(), pair._1()))
                                                        .sortByKey(false)
                                                        .take(10);

    return mostMentioned;
  }
```

Exercises

- **SPARK SQL (DataFrame)**



Exercises

```
Michael, 29
Andy, 30
Justin, 19
```

SPARK DataFrame: SparkSQL

```java
public class SparkSQL {
  public static class Person implements Serializable {
    private String name;
    private int age;

    public String getName() {
      return name;
    }

    public void setName(String name) {
      this.name = name;
    }

    public int getAge() {
      return age;
    }

    public void setAge(int age) {
      this.age = age;
    }
  }
```

Exercises

SPARK DataFrame: SparkSQL

```
public static void main(String[] args) throws Exception {

       if (args.length < 2) {
           System.err.println("Usage: JavaSparkSQL <filetxt> <filejson>");
           System.exit(1);
         }

    SparkConf sparkConf = new SparkConf().setAppName("JavaSparkSQL");
    JavaSparkContext ctx = new JavaSparkContext(sparkConf);
    SQLContext sqlContext = new SQLContext(ctx);
```

```
Michael, 29
Andy, 30
Justin, 19
```

```
{"name":"Michael"}
{"name":"Andy", "age":30}
{"name":"Justin", "age":19}
```

Exercises

Michael, 29
Andy, 30
Justin, 19

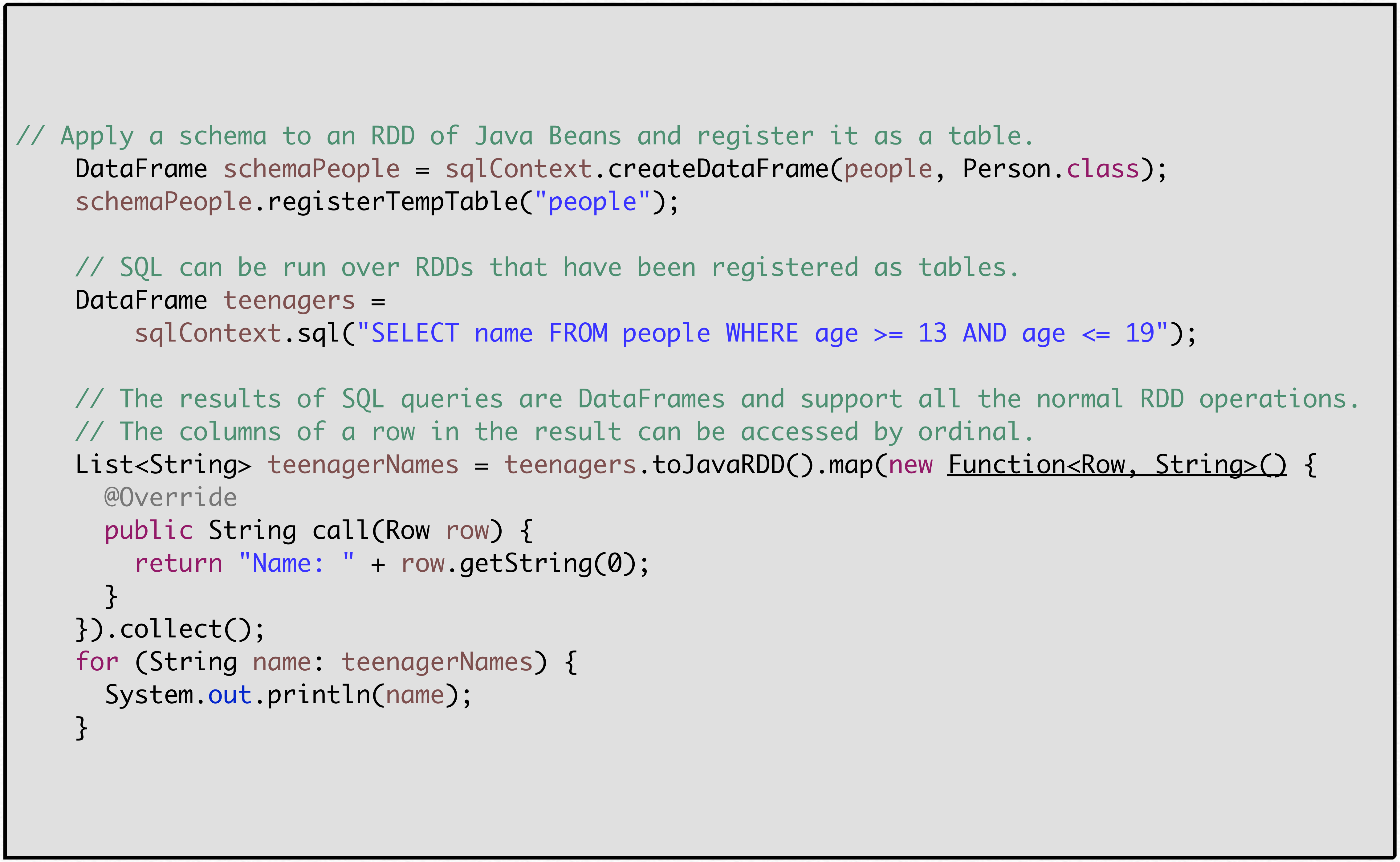
SPARK DataFrame: SparkSQL

```java
System.out.println("=== Data source: RDD ===");
    // Load a text file and convert each line to a Java Bean.
    JavaRDD<Person> people = ctx.textFile(args[0]).map(
      new Function<String, Person>() {
        @Override
        public Person call(String line) {
          String[] parts = line.split(",");

          Person person = new Person();
          person.setName(parts[0]);
          person.setAge(Integer.parseInt(parts[1].trim()));

          return person;
        }
      });
```

Exercises

Michael, 29
Andy, 30
Justin, 19

SPARK DataFrame: SparkSQL

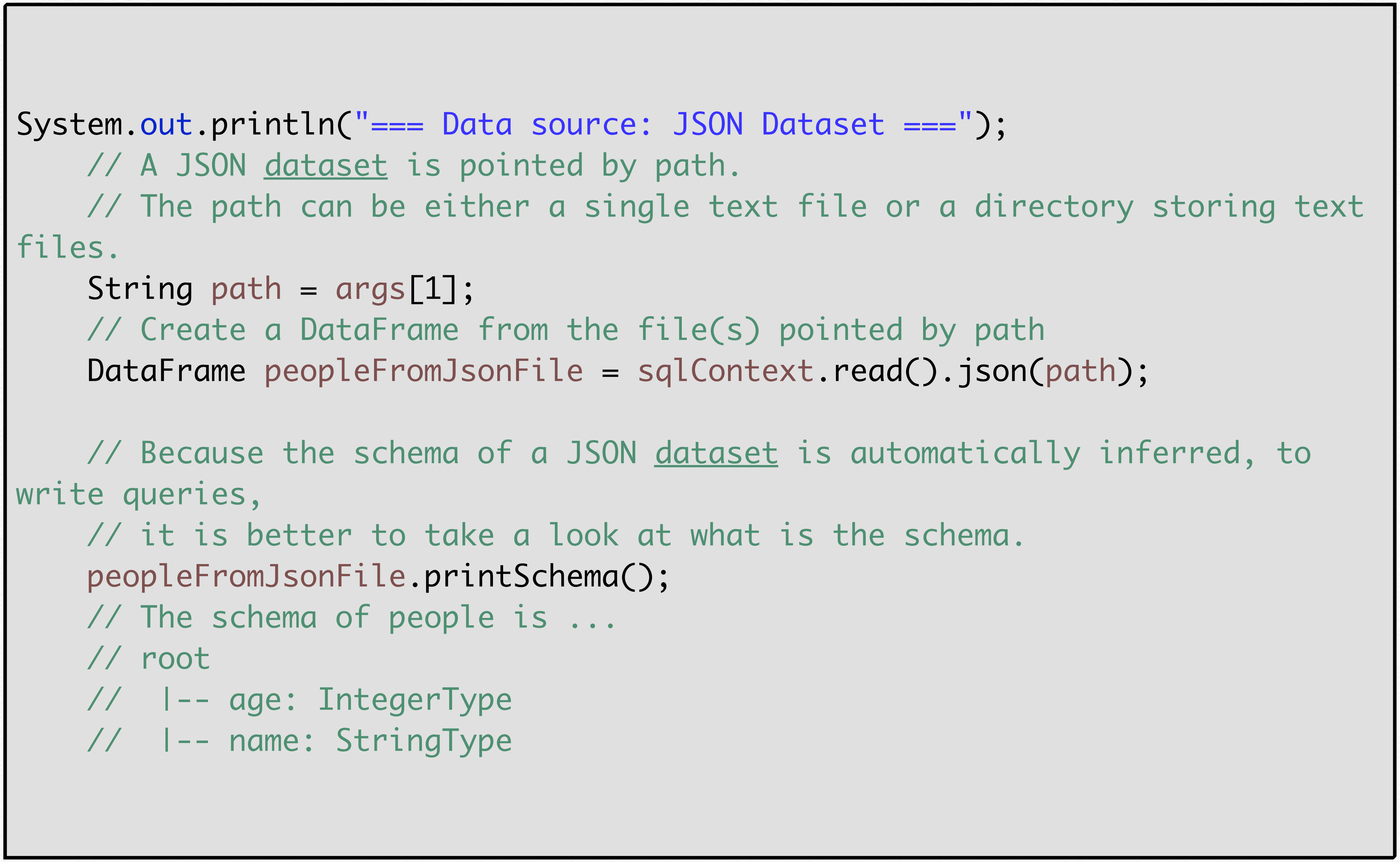


```
// Apply a schema to an RDD of Java Beans and register it as a table.
  DataFrame schemaPeople = sqlContext.createDataFrame(people, Person.class);
  schemaPeople.registerTempTable("people");

  // SQL can be run over RDDs that have been registered as tables.
  DataFrame teenagers =
    sqlContext.sql("SELECT name FROM people WHERE age >= 13 AND age <= 19");

  // The results of SQL queries are DataFrames and support all the normal RDD operations.
  // The columns of a row in the result can be accessed by ordinal.
  List<String> teenagerNames = teenagers.toJavaRDD().map(new Function<Row, String>() {
   @Override
   public String call(Row row) {
    return "Name: " + row.getString(0);
   }
  }).collect();
  for (String name: teenagerNames) {
   System.out.println(name);
  }
```

Exercises

{"name":"Michael"}
{"name":"Andy", "age":30}
{"name":"Justin", "age":19}

SPARK DataFrame: SparkSQL

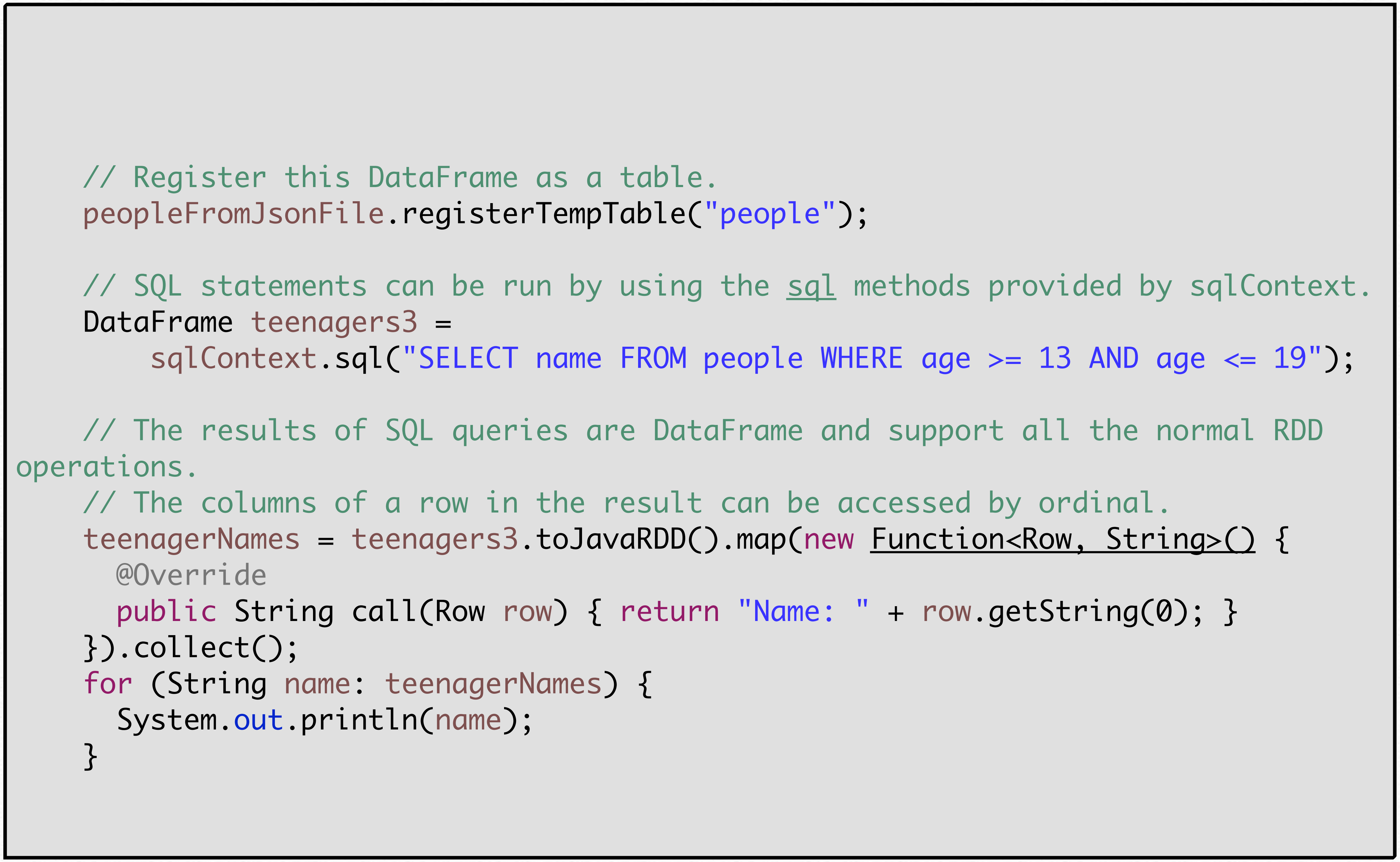


```
System.out.println("=== Data source: JSON Dataset ===");
    // A JSON dataset is pointed by path.
    // The path can be either a single text file or a directory storing text
files.
    String path = args[1];
    // Create a DataFrame from the file(s) pointed by path
    DataFrame peopleFromJsonFile = sqlContext.read().json(path);

    // Because the schema of a JSON dataset is automatically inferred, to
write queries,
    // it is better to take a look at what is the schema.
    peopleFromJsonFile.printSchema();
    // The schema of people is ...
    // root
    //  |-- age: IntegerType
    //  |-- name: StringType
```

Exercises

```
{"name":"Michael"}
{"name":"Andy", "age":30}
{"name":"Justin", "age":19}
```

SPARK DataFrame: SparkSQL

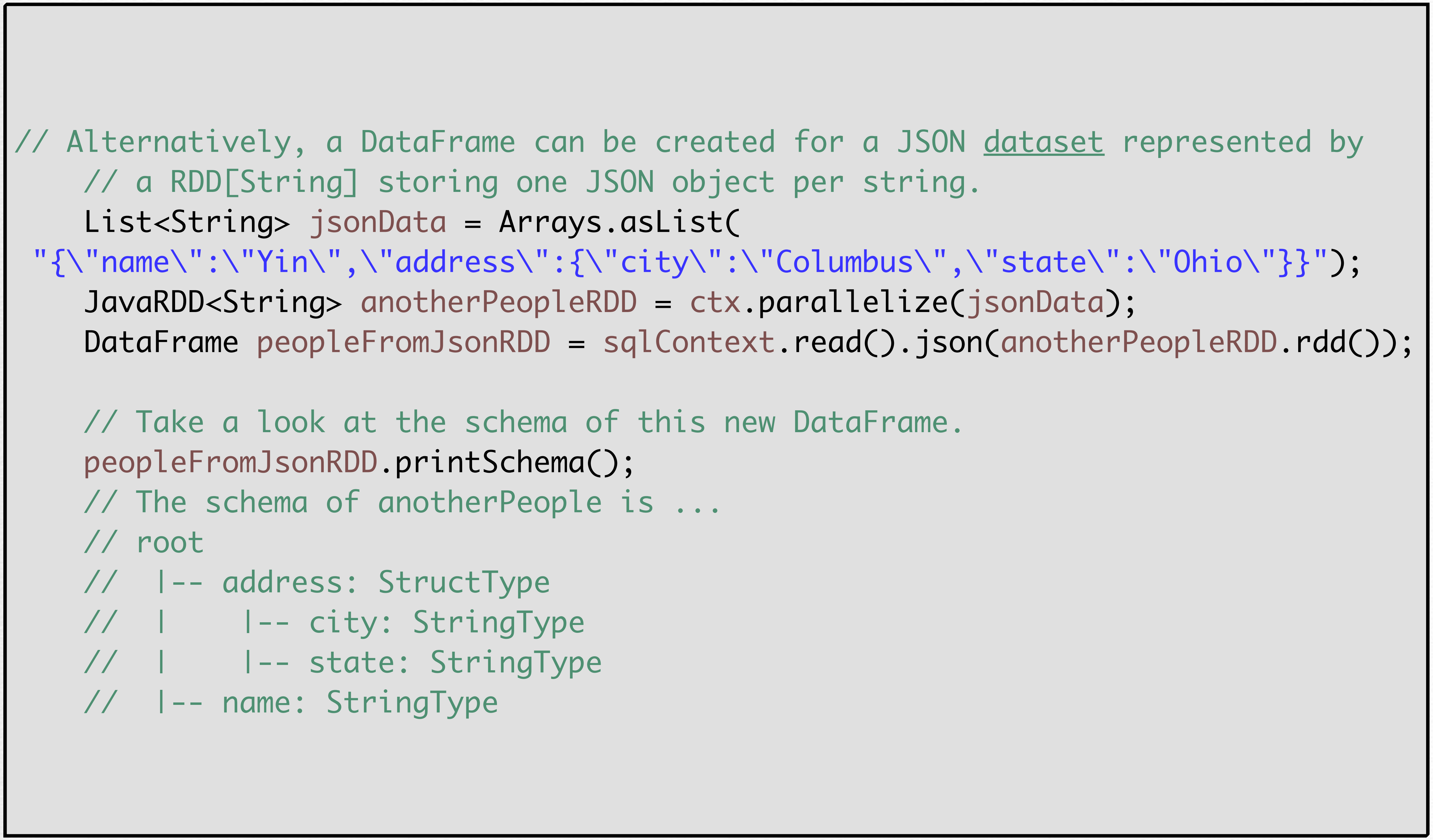


```
    // Register this DataFrame as a table.
    peopleFromJsonFile.registerTempTable("people");

    // SQL statements can be run by using the sql methods provided by sqlContext.
    DataFrame teenagers3 =
        sqlContext.sql("SELECT name FROM people WHERE age >= 13 AND age <= 19");

    // The results of SQL queries are DataFrame and support all the normal RDD
operations.
    // The columns of a row in the result can be accessed by ordinal.
    teenagerNames = teenagers3.toJavaRDD().map(new Function<Row, String>() {
      @Override
      public String call(Row row) { return "Name: " + row.getString(0); }
    }).collect();
    for (String name: teenagerNames) {
      System.out.println(name);
    }
```

Exercises

```
{"name":"Michael"}
{"name":"Andy", "age":30}
{"name":"Justin", "age":19}
```

SPARK DataFrame: SparkSQL

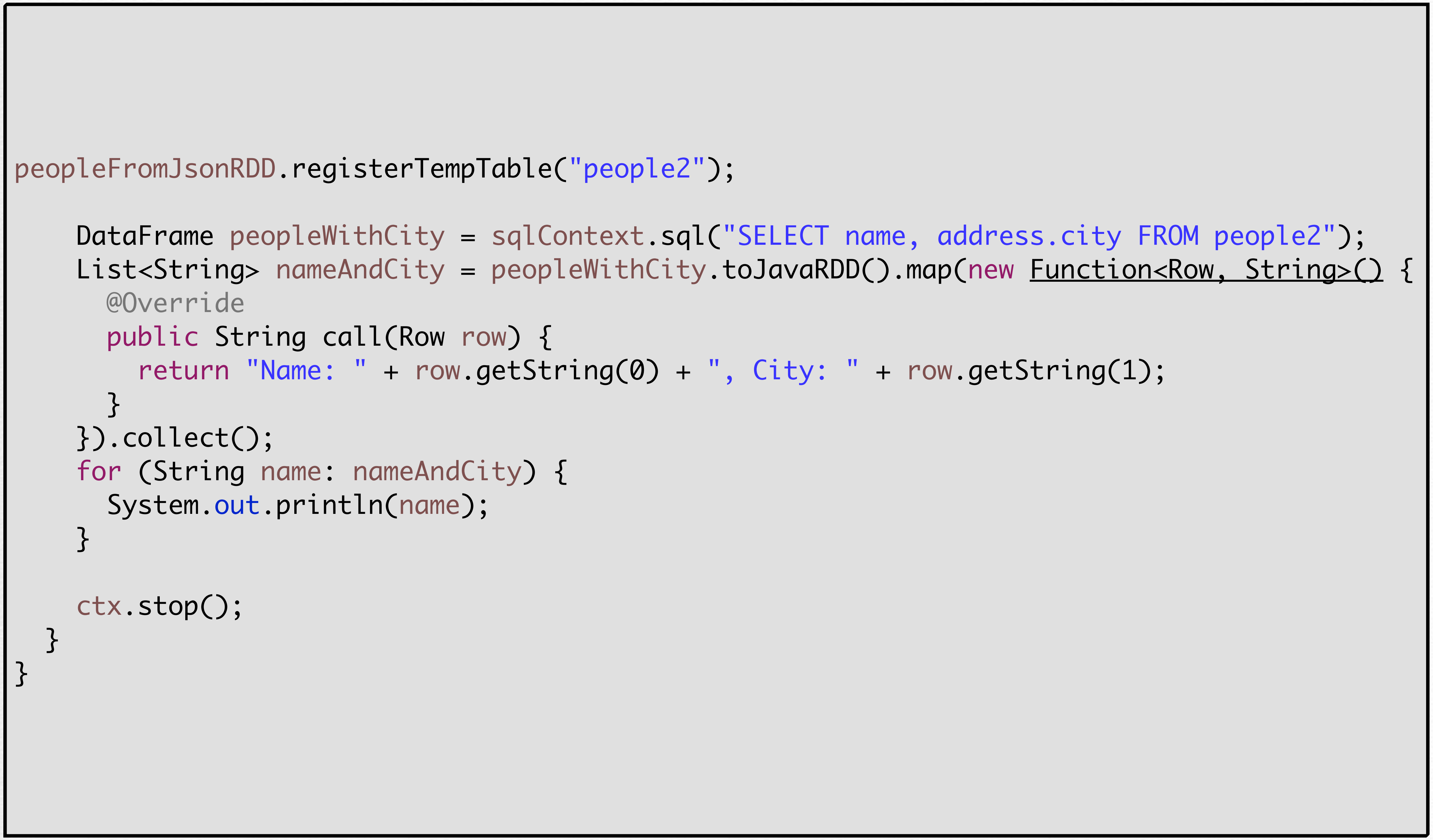


```
// Alternatively, a DataFrame can be created for a JSON dataset represented by
    // a RDD[String] storing one JSON object per string.
    List<String> jsonData = Arrays.asList(
 "{\"name\":\"Yin\",\"address\":{\"city\":\"Columbus\",\"state\":\"Ohio\"}}");
    JavaRDD<String> anotherPeopleRDD = ctx.parallelize(jsonData);
    DataFrame peopleFromJsonRDD = sqlContext.read().json(anotherPeopleRDD.rdd());

    // Take a look at the schema of this new DataFrame.
    peopleFromJsonRDD.printSchema();
    // The schema of anotherPeople is ...
    // root
    //  |-- address: StructType
    //  |     |-- city: StringType
    //  |     |-- state: StringType
    //  |-- name: StringType
```

Exercises

```
{"name":"Michael"}
{"name":"Andy", "age":30}
{"name":"Justin", "age":19}
```

SPARK DataFrame: SparkSQL



```
peopleFromJsonRDD.registerTempTable("people2");

    DataFrame peopleWithCity = sqlContext.sql("SELECT name, address.city FROM people2");
    List<String> nameAndCity = peopleWithCity.toJavaRDD().map(new Function<Row, String>() {
      @Override
      public String call(Row row) {
        return "Name: " + row.getString(0) + ", City: " + row.getString(1);
      }
    }).collect();
    for (String name: nameAndCity) {
      System.out.println(name);
    }

    ctx.stop();
  }
}
```

Exercises

SPARK DataFrame: DataFrameOnTweets

We use again a dataset with 8198 tweets.
Here an example of a tweet (json):

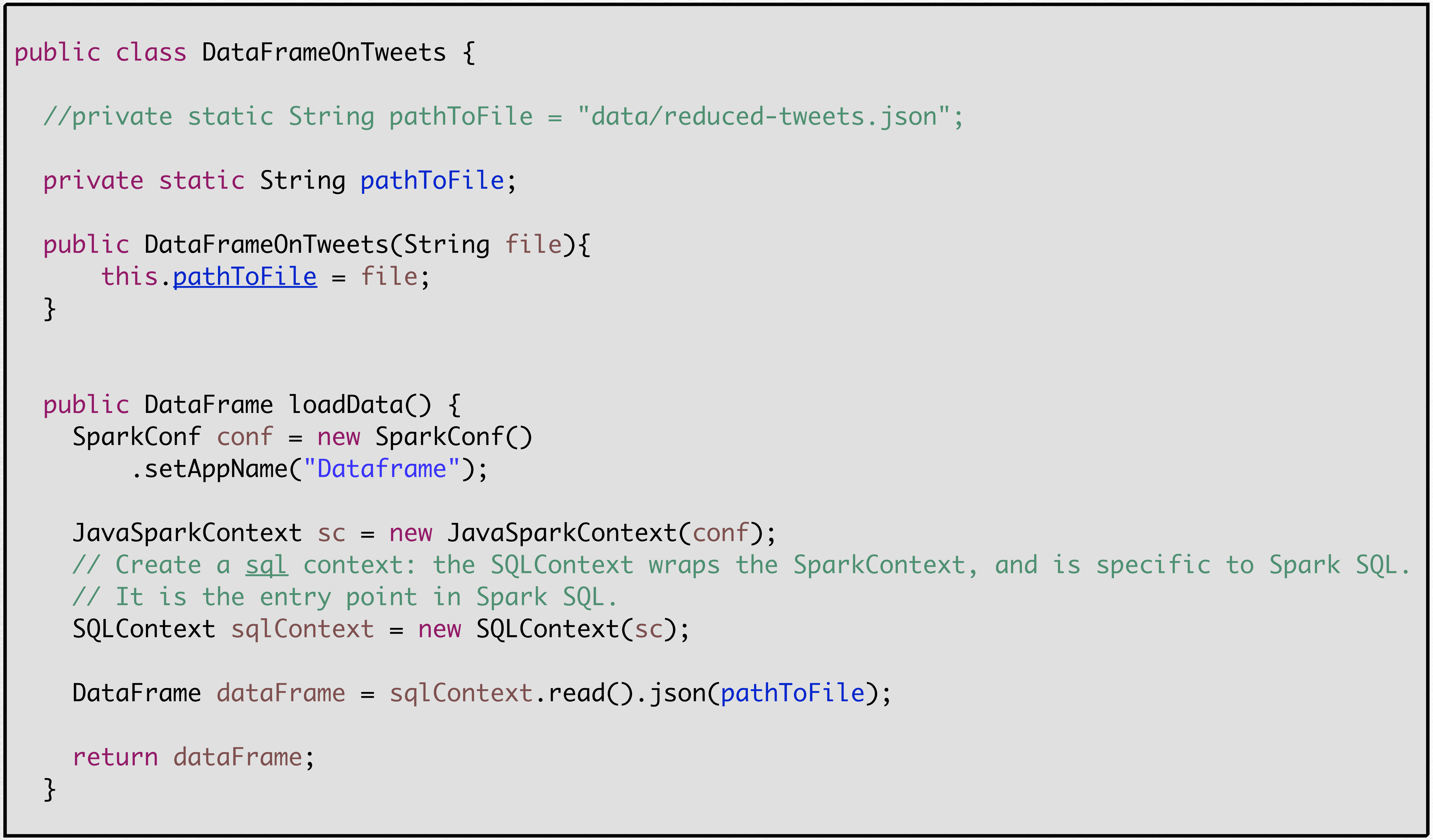
```
   {"id":"572692378957430785",
    "user":"Srkian_nishu :)",
    "text":"@always_nidhi @YouTube no i dnt understand bt i
             loved of this mve is rocking",
    "place":"Orissa",
    “country":"India"
    }
```

In the exercise we will create a dataframe with the content of a JSON file. We want to:

- print the dataframe
- print the schema of the dataframe
- find people who are located in Paris
- find the user who tweets the more

Exercises

SPARK DataFrame: DataFrameOnTweets



```java
public class DataFrameOnTweets {

  //private static String pathToFile = "data/reduced-tweets.json";

  private static String pathToFile;

  public DataFrameOnTweets(String file){
      this.pathToFile = file;
  }


  public DataFrame loadData() {
    SparkConf conf = new SparkConf()
        .setAppName("Dataframe");

    JavaSparkContext sc = new JavaSparkContext(conf);
    // Create a sql context: the SQLContext wraps the SparkContext, and is specific to Spark SQL.
    // It is the entry point in Spark SQL.
    SQLContext sqlContext = new SQLContext(sc);

    DataFrame dataFrame = sqlContext.read().json(pathToFile);

    return dataFrame;
  }
```

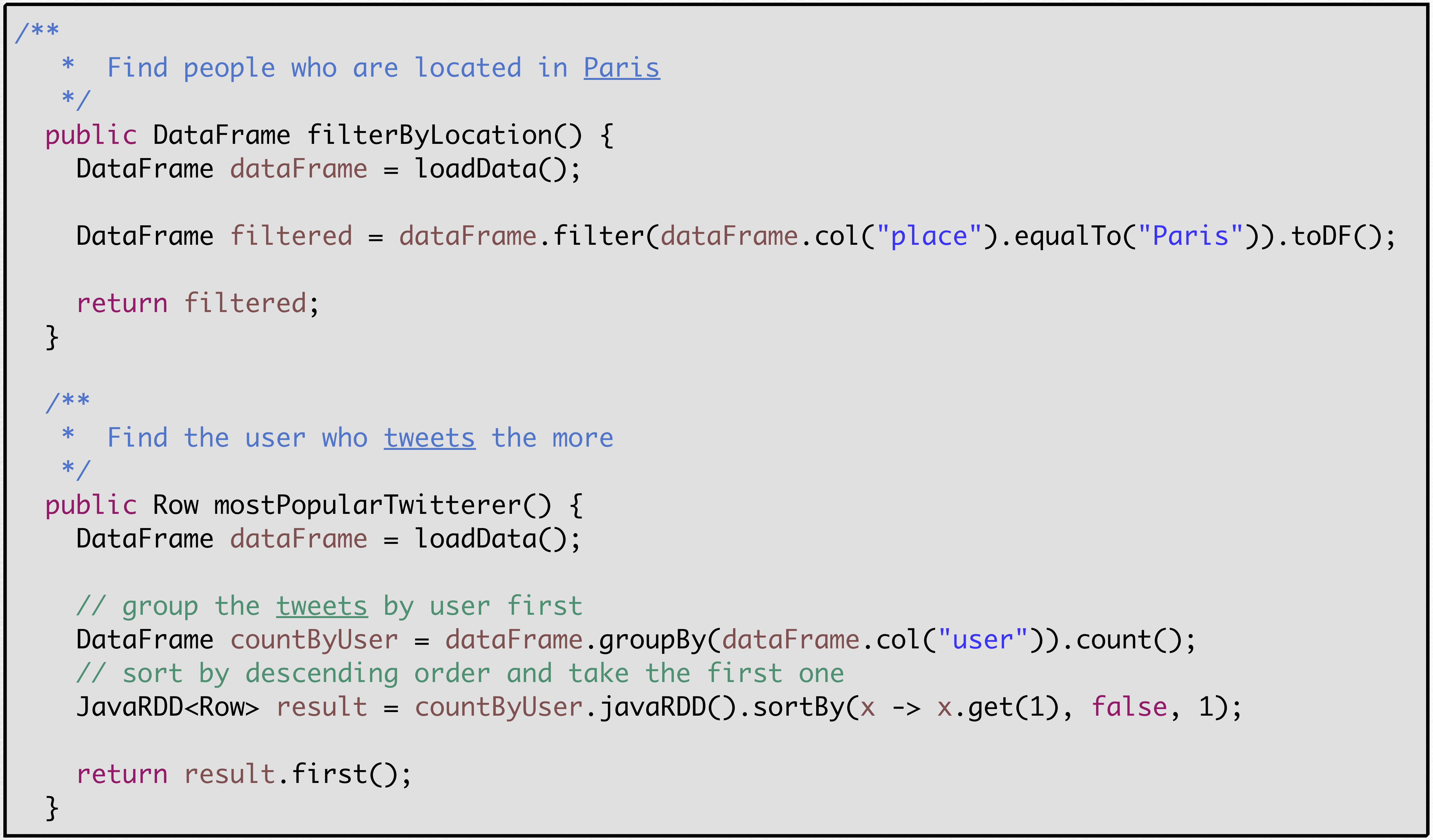
Exercises

SPARK DataFrame: DataFrameOnTweets

```
/**
   *  See how looks the dataframe
   */
  public void showDataFrame() {
    DataFrame dataFrame = loadData();
    // Displays the content of the DataFrame to stdout
    dataFrame.show();
  }

  /**
   * Print the schema
   */
  public void printSchema() {
    DataFrame dataFrame = loadData();

    dataFrame.printSchema();
  }
```

Exercises

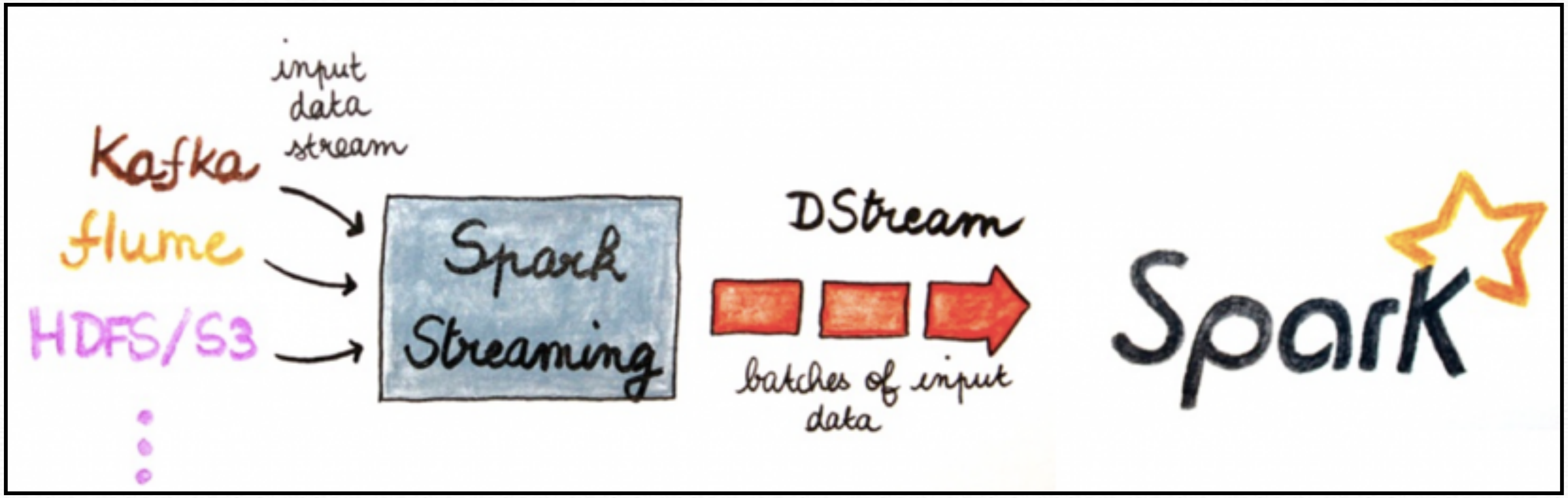
SPARK DataFrame: DataFrameOnTweets



```
/**
   *  Find people who are located in Paris
   */
  public DataFrame filterByLocation() {
    DataFrame dataFrame = loadData();

    DataFrame filtered = dataFrame.filter(dataFrame.col("place").equalTo("Paris")).toDF();

    return filtered;
  }

  /**
   *  Find the user who tweets the more
   */
  public Row mostPopularTwitterer() {
    DataFrame dataFrame = loadData();

    // group the tweets by user first
    DataFrame countByUser = dataFrame.groupBy(dataFrame.col("user")).count();
    // sort by descending order and take the first one
    JavaRDD<Row> result = countByUser.javaRDD().sortBy(x -> x.get(1), false, 1);

    return result.first();
  }
```

Exercises

- **SPARK Streaming**



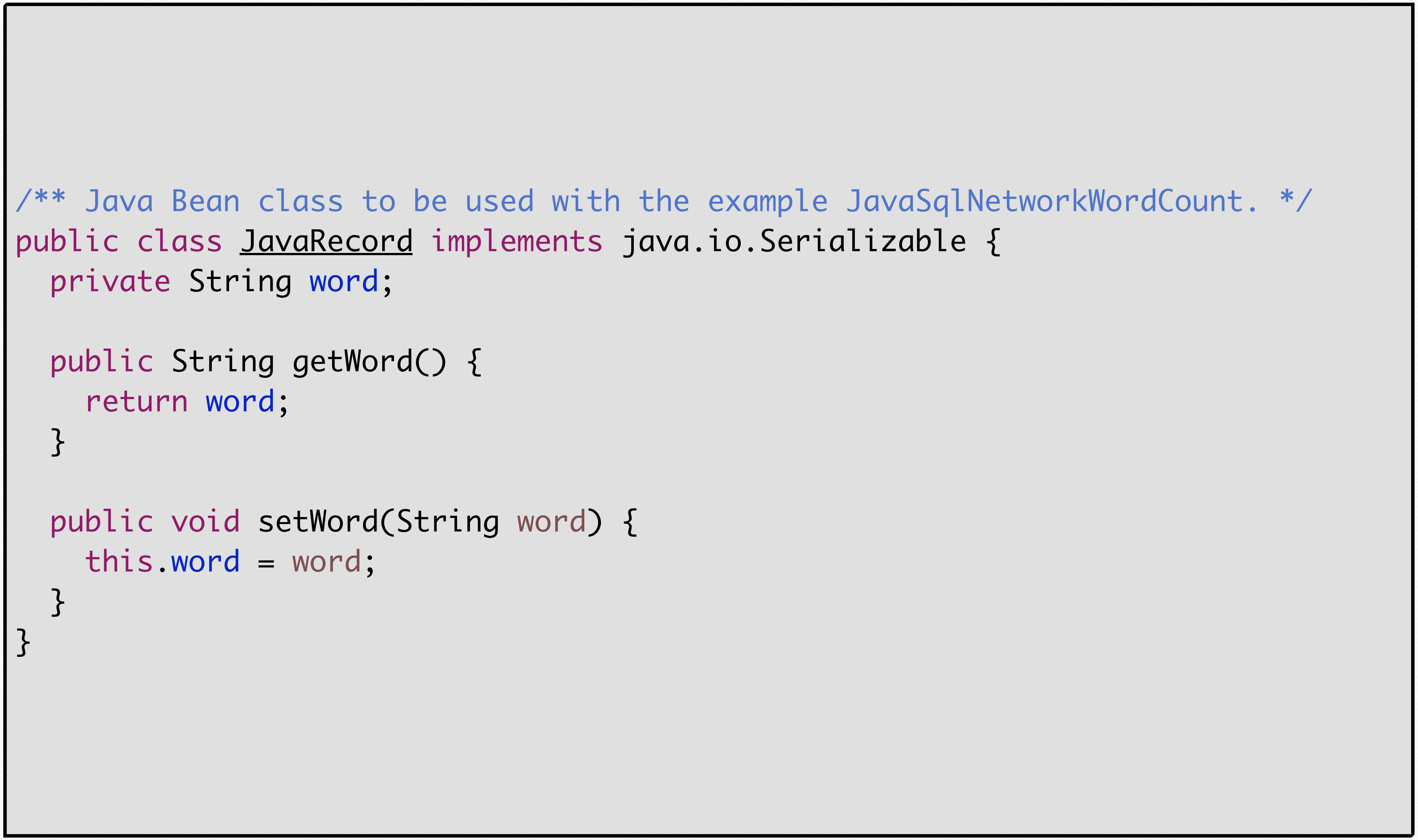
Exercises

SPARK Streaming: SparkSQLStreaming

```
/**
 * Use DataFrames and SQL to count words in UTF8 encoded, '\n'
delimited text received from the
 * network every second.
 *
 * Usage: JavaSqlNetworkWordCount <hostname> <port>
 * <hostname> and <port> describe the TCP server that Spark
Streaming would connect to receive data.
 *
 * To run this on your local machine, you need to first run a
Netcat server
 *    `$ nc -lk 9999`
 * and then run the example
 *    `$ SparkSQLStreaming localhost 9999`
 */
```

Exercises

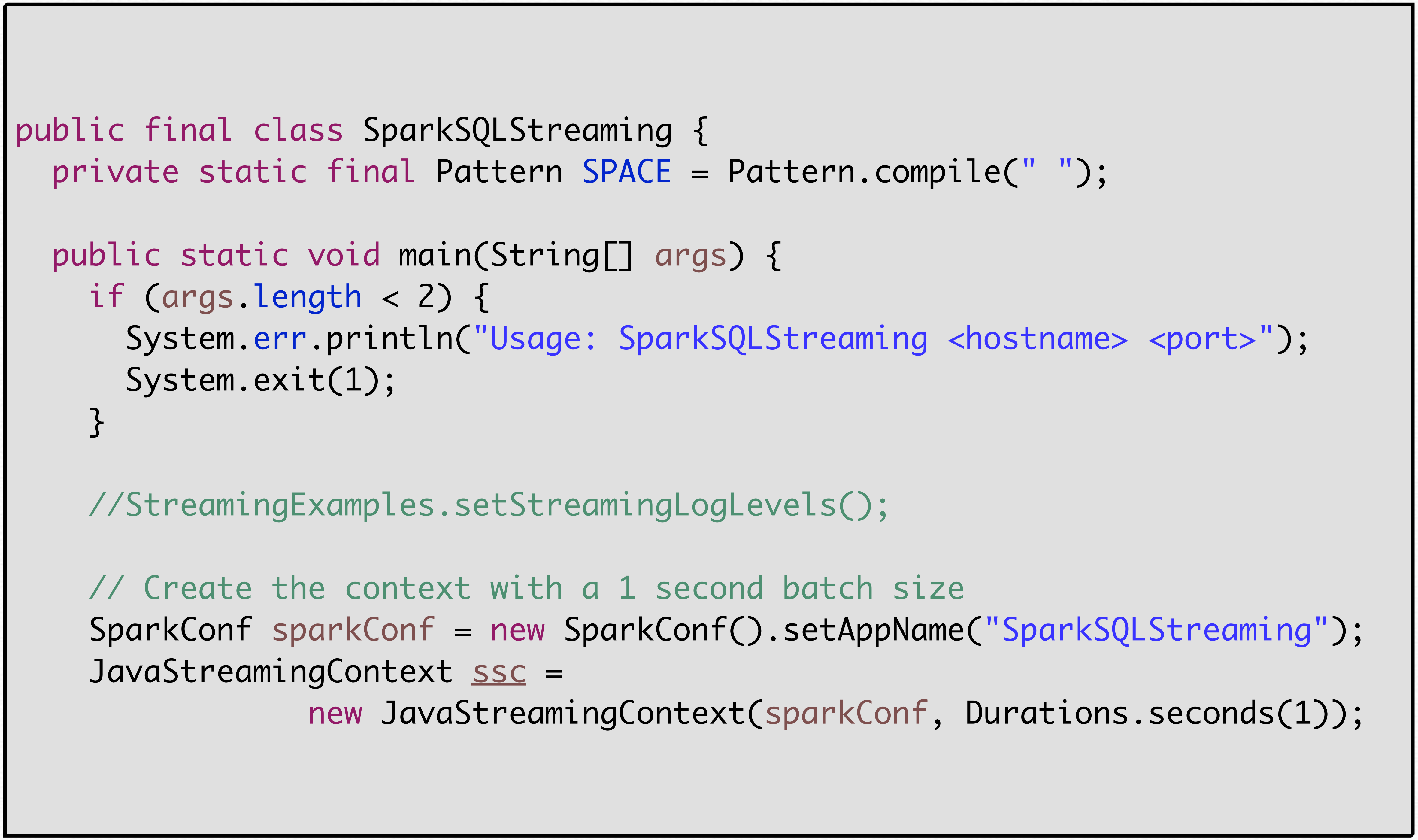
SPARK Streaming: SparkSQLStreaming



```java
/** Java Bean class to be used with the example JavaSqlNetworkWordCount. */
public class JavaRecord implements java.io.Serializable {
  private String word;

  public String getWord() {
    return word;
  }

  public void setWord(String word) {
    this.word = word;
  }
}
```

Exercises

SPARK Streaming: SparkSQLStreaming



```java
public final class SparkSQLStreaming {
  private static final Pattern SPACE = Pattern.compile(" ");

  public static void main(String[] args) {
    if (args.length < 2) {
      System.err.println("Usage: SparkSQLStreaming <hostname> <port>");
      System.exit(1);
    }

    //StreamingExamples.setStreamingLogLevels();

    // Create the context with a 1 second batch size
    SparkConf sparkConf = new SparkConf().setAppName("SparkSQLStreaming");
    JavaStreamingContext ssc =
                new JavaStreamingContext(sparkConf, Durations.seconds(1));
```

Exercises

SPARK Streaming: SparkSQLStreaming

```
// Create a JavaReceiverInputDStream on target ip:port and count the
    // words in input stream of \n delimited text (eg. generated by 'nc')
    // Note that no duplication in storage level only for running locally.
    // Replication necessary in distributed scenario for fault tolerance.
    JavaReceiverInputDStream<String> lines =
                 ssc.socketTextStream(args[0],
                                      Integer.parseInt(args[1]),
                                      StorageLevels.MEMORY_AND_DISK_SER);
    JavaDStream<String> words =
      lines.flatMap(new FlatMapFunction<String, String>() {
      @Override
      public Iterable<String> call(String x) {
        return Arrays.asList(SPACE.split(x));
      }
    });
```

Exercises

SPARK Streaming: SparkSQLStreaming

```
// Convert RDDs of the words DStream to DataFrame and run SQL query
    words.foreachRDD(new VoidFunction2<JavaRDD<String>, Time>() {
      @Override
      public void call(JavaRDD<String> rdd, Time time) {
        SQLContext sqlContext = JavaSQLContextSingleton.getInstance(rdd.context());

        // Convert JavaRDD[String] to JavaRDD[bean class] to DataFrame
        JavaRDD<JavaRecord> rowRDD = rdd.map(new Function<String, JavaRecord>() {
          @Override
          public JavaRecord call(String word) {
            JavaRecord record = new JavaRecord();
            record.setWord(word);
            return record;
          }
        });
```

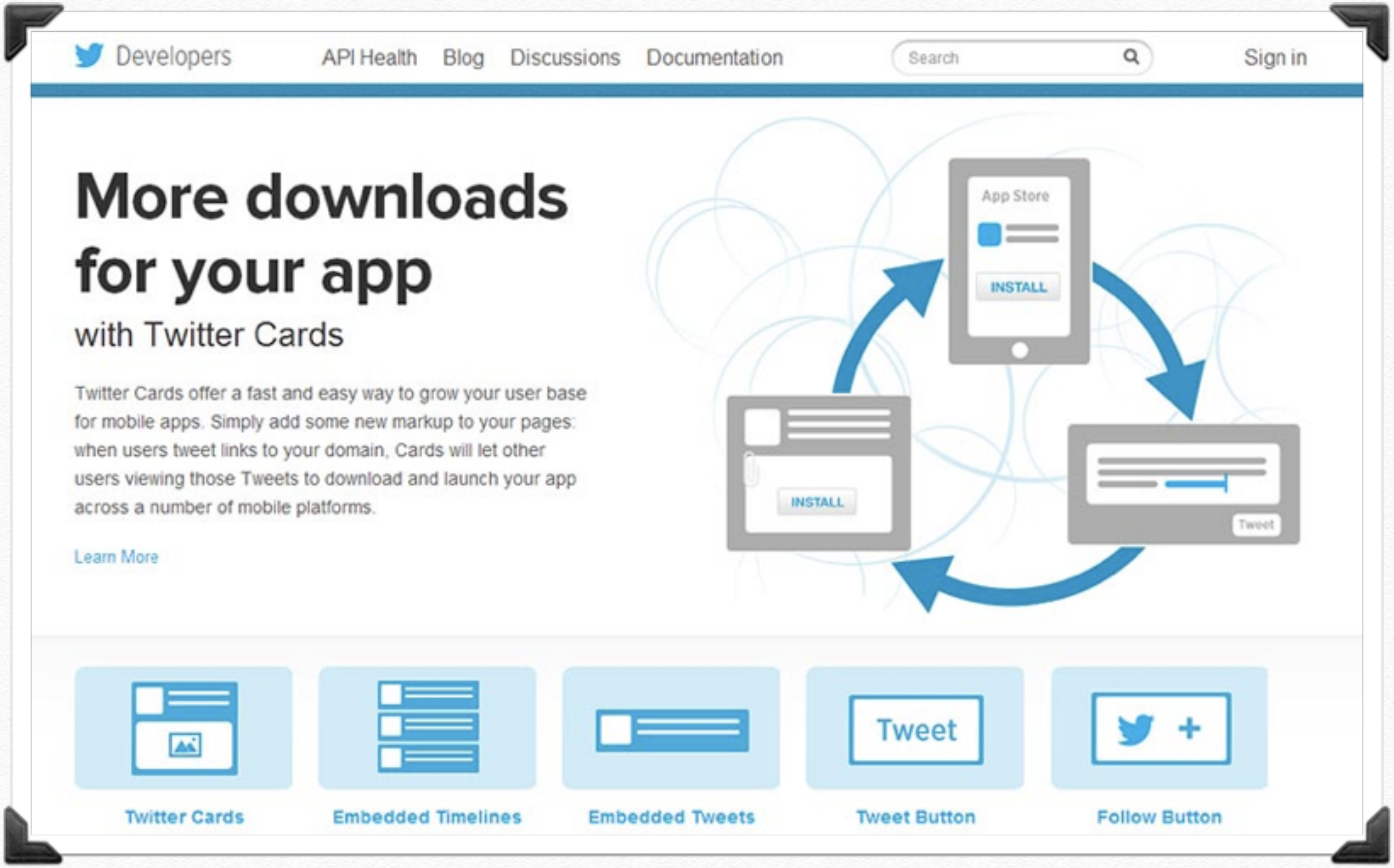
Exercises

SPARK Streaming: SparkSQLStreaming

```
DataFrame wordsDataFrame = sqlContext.createDataFrame(rowRDD, JavaRecord.class);

        // Register as table
        wordsDataFrame.registerTempTable("words");

        // Do word count on table using SQL and print it
        DataFrame wordCountsDataFrame =
            sqlContext.sql("select word, count(*) as total from words group by word");
        System.out.println("========= " + time + "=========");
        wordCountsDataFrame.show();
      }
    });

    ssc.start();
    ssc.awaitTermination();
  }
}
```

How to Register a Twitter App in 8 Easy Steps

#1 Visit the Twitter Developers' Site



More downloads for your app

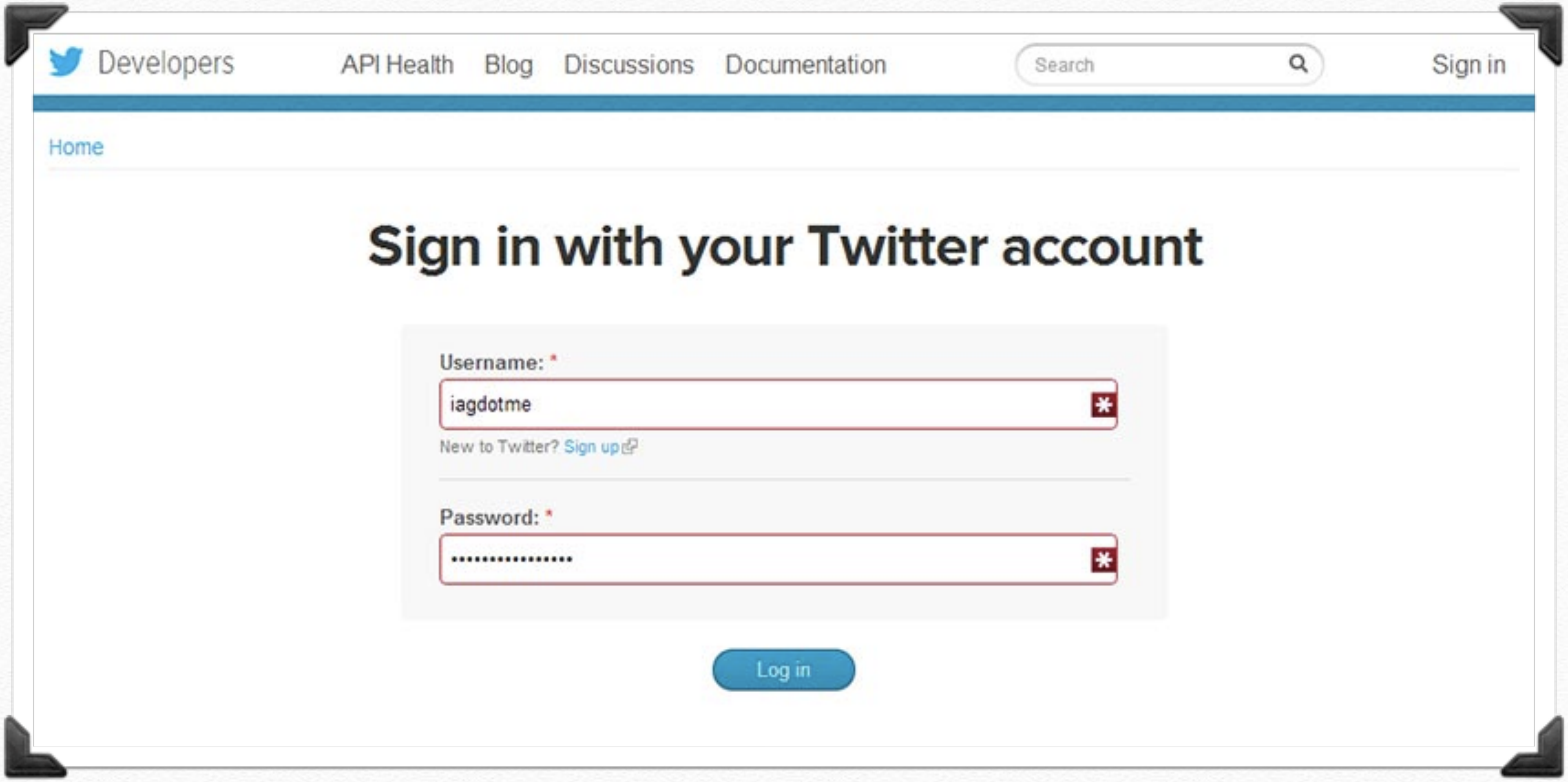
with Twitter Cards

Twitter Cards offer a fast and easy way to grow your user base for mobile apps. Simply add some new markup to your pages: when users tweet links to your domain, Cards will let other users viewing those Tweets to download and launch your app across a number of mobile platforms.

Learn More

Spark How to Register a Twitter App in 8 Easy Steps

#2 Sign in with your Twitter Account



Sign in with your Twitter account

Username: *

iagdotme

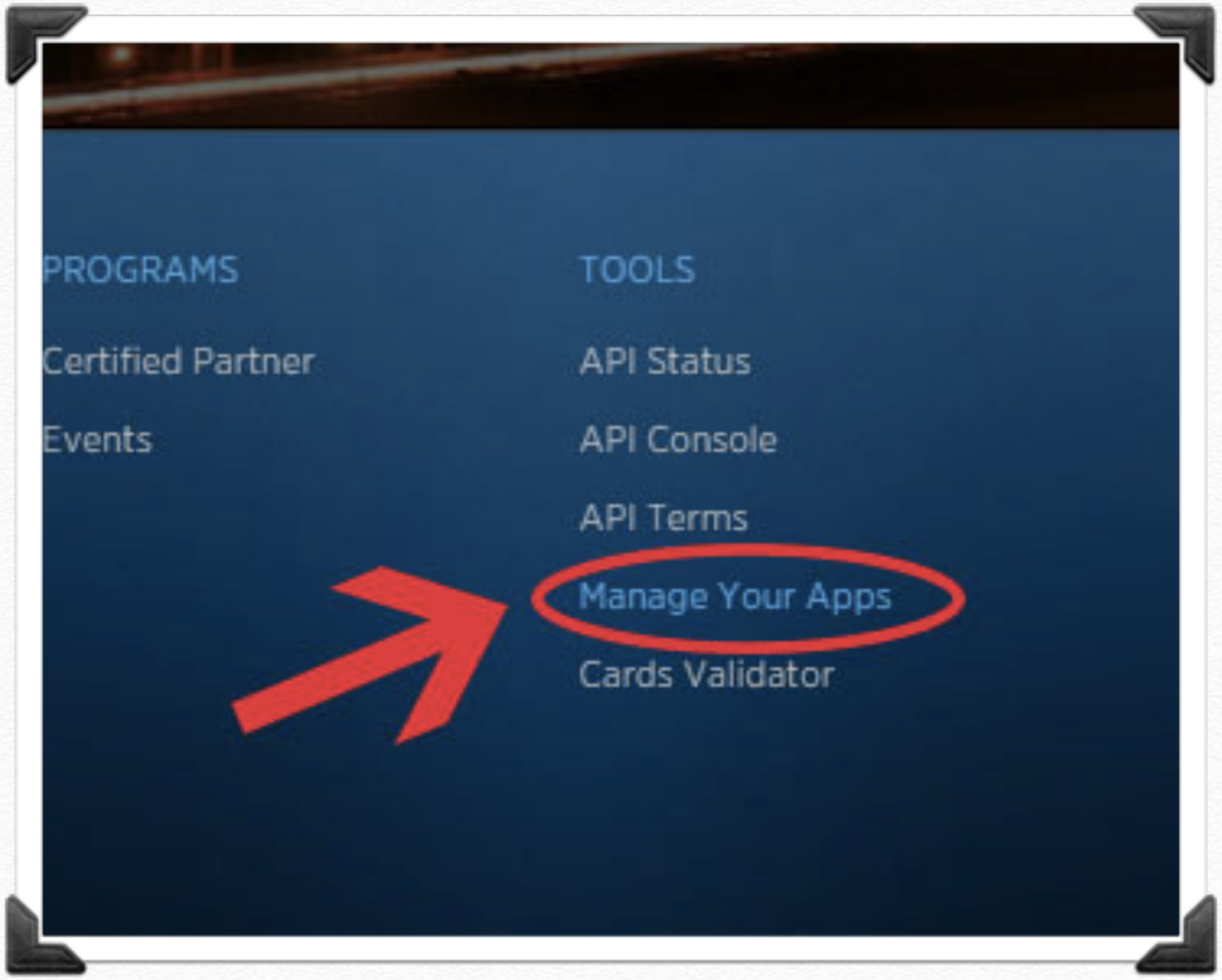
New to Twitter? Sign up

Password: *

•••••••••••••••••

How to Register a Twitter App in 8 Easy Steps

- **#3 Go to apps.twitter.com**



PROGRAMS

Certified Partner

Events

TOOLS

API Status

API Console

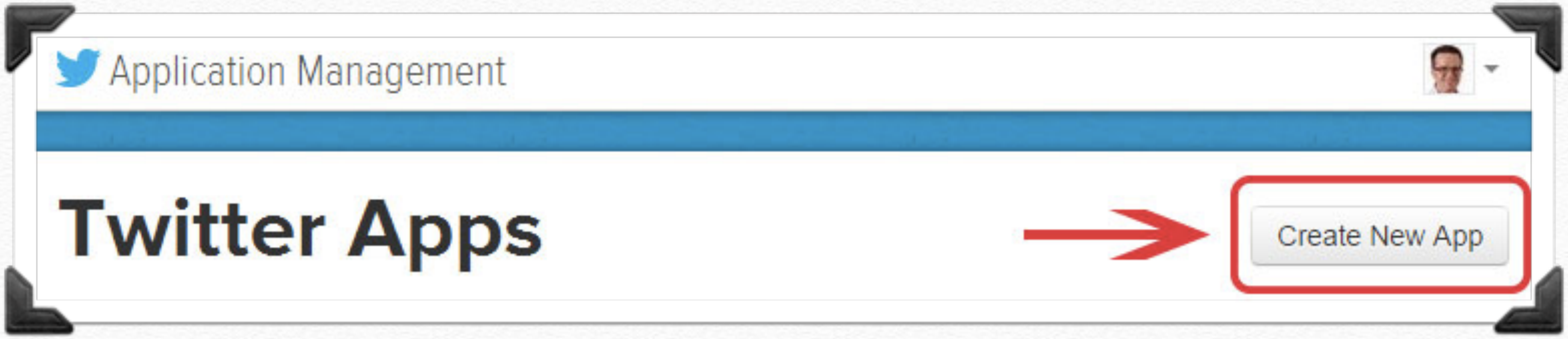
API Terms

Manage Your Apps

Cards Validator

Spark How to Register a Twitter App in 8 Easy Steps

#4 Create a New Application

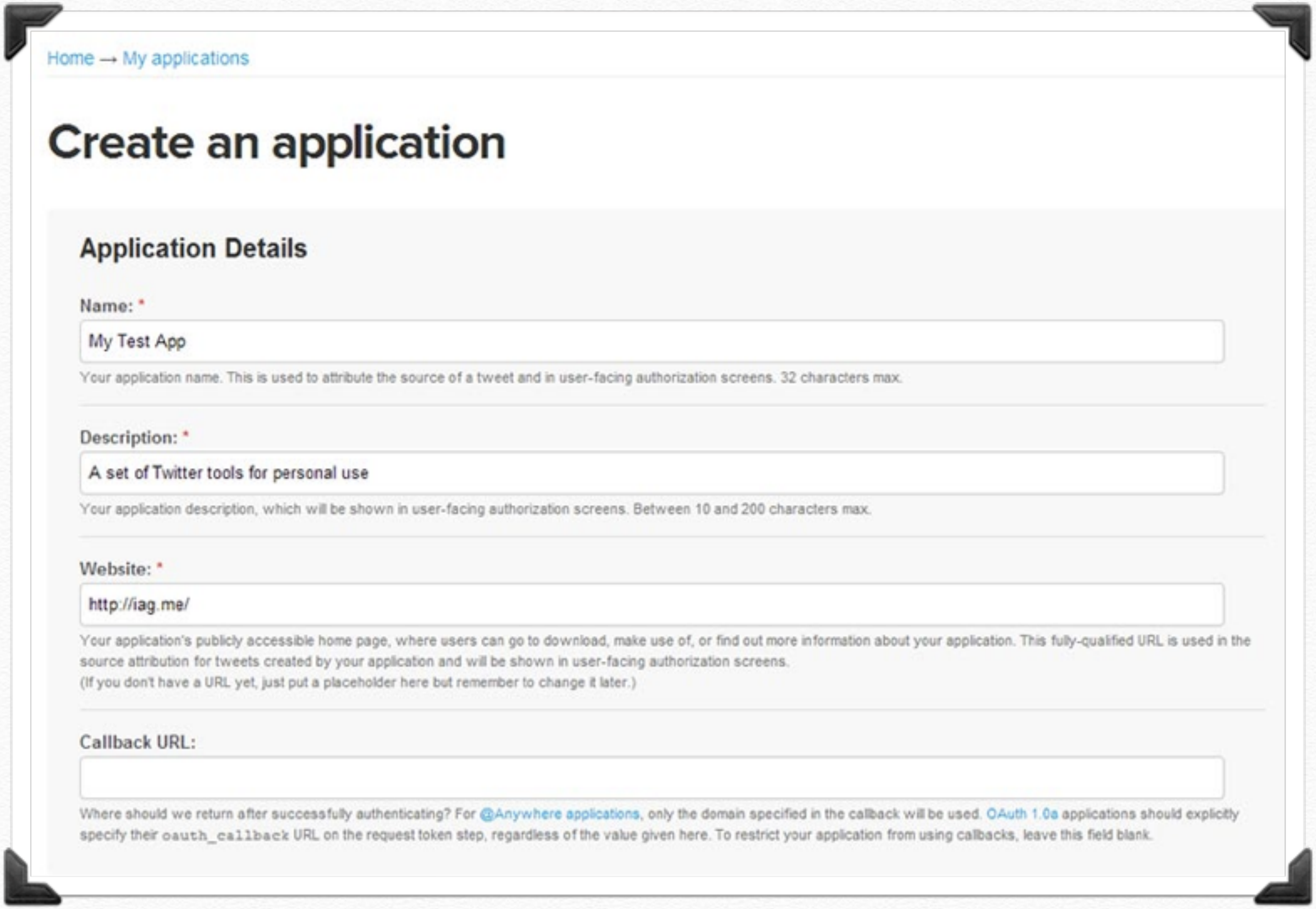


Application Management

Twitter Apps

How to Register a Twitter App in 8 Easy Steps

#5 Fill in your Application Details



Home → My applications

Create an application

Application Details

Name: *

My Test App

Your application name. This is used to attribute the source of a tweet and in user-facing authorization screens. 32 characters max.

Description: *

A set of Twitter tools for personal use

Your application description, which will be shown in user-facing authorization screens. Between 10 and 200 characters max.

Website: *

http://iag.me/

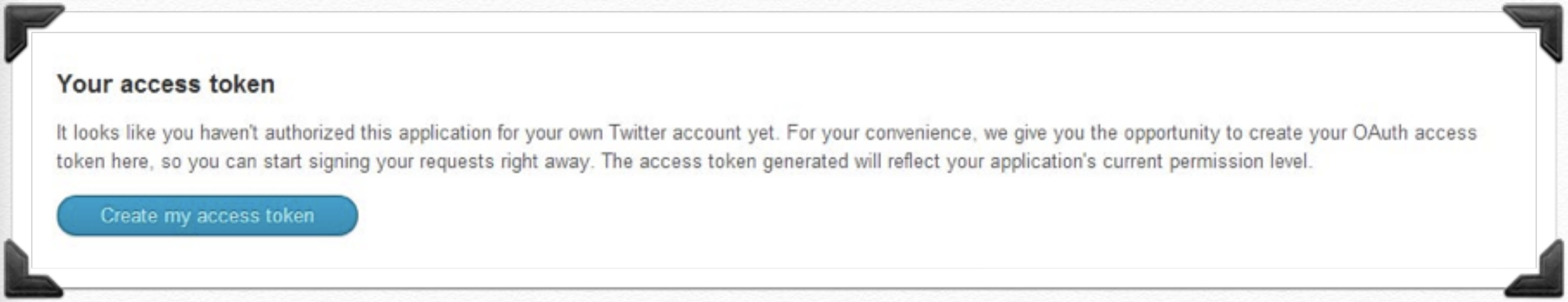
Your application's publicly accessible home page, where users can go to download, make use of, or find out more information about your application. This fully-qualified URL is used in the source attribution for tweets created by your application and will be shown in user-facing authorization screens.
(If you don't have a URL yet, just put a placeholder here but remember to change it later.)

Callback URL:

Where should we return after successfully authenticating? For @Anywhere applications, only the domain specified in the callback will be used. OAuth 1.0a applications should explicitly specify their `oauth_callback` URL on the request token step, regardless of the value given here. To restrict your application from using callbacks, leave this field blank.

How to Register a Twitter App in 8 Easy Steps

#6 Create Your Access Token

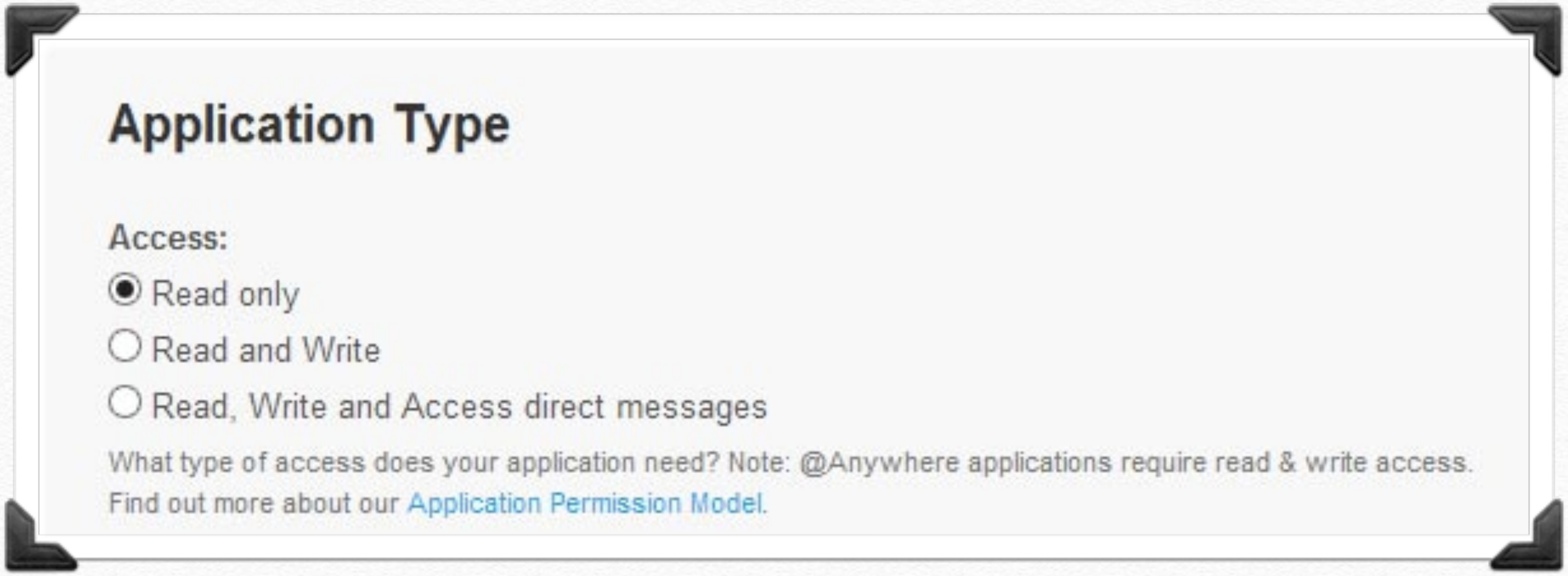


Your access token

It looks like you haven't authorized this application for your own Twitter account yet. For your convenience, we give you the opportunity to create your OAuth access token here, so you can start signing your requests right away. The access token generated will reflect your application's current permission level.

How to Register a Twitter App in 8 Easy Steps

#7 Choose what Access Type You Need



Application Type

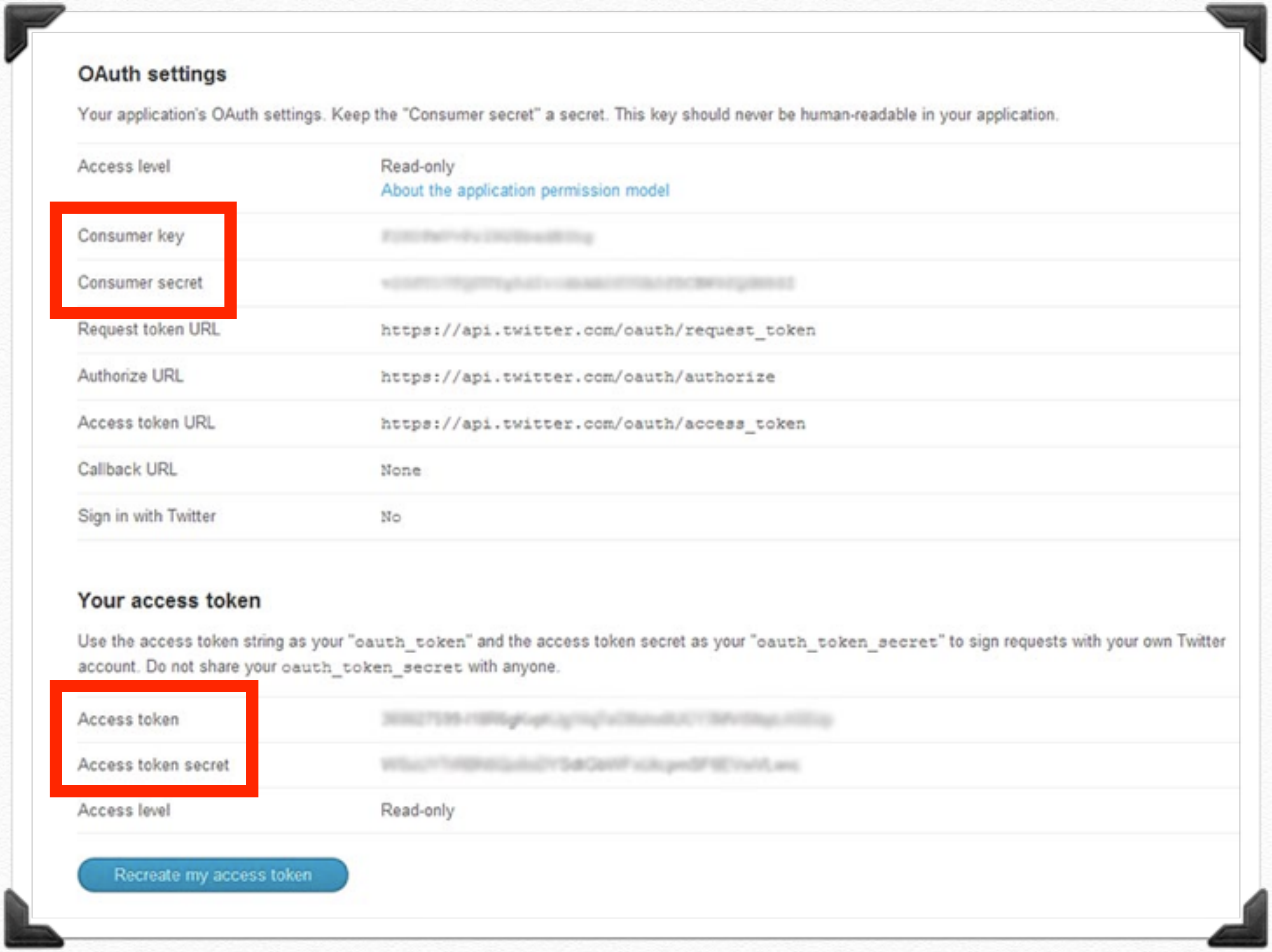
Access:

● Read only
○ Read and Write
○ Read, Write and Access direct messages

What type of access does your application need? Note: @Anywhere applications require read & write access.
Find out more about our Application Permission Model.

How to Register a Twitter App in 8 Easy Steps

#8 Make a note of your OAuth Settings



OAuth settings

Your application's OAuth settings. Keep the "Consumer secret" a secret. This key should never be human-readable in your application.

Access level	Read-only About the application permission model
Consumer key	[redacted]
Consumer secret	[redacted]
Request token URL	https://api.twitter.com/oauth/request_token
Authorize URL	https://api.twitter.com/oauth/authorize
Access token URL	https://api.twitter.com/oauth/access_token
Callback URL	None
Sign in with Twitter	No

Your access token

Use the access token string as your "oauth_token" and the access token secret as your "oauth_token_secret" to sign requests with your own Twitter account. Do not share your oauth_token_secret with anyone.

Access token	[redacted]
Access token secret	[redacted]
Access level	Read-only

Exercises

SPARK Streaming: StreamUtils

```
public class StreamUtils {

  private static String CONSUMER_KEY = "AFiNCb8vxYZfhPls2DXyDpF";
  private static String CONSUMER_SECRET = "JRg7SyVFkXEESWbzFzC1xaIGRC3xNdTvrekMvMFk6tjKooOR";
  private static String ACCESS_TOKEN = "493498548-HCt6LCposCb3Ij7Ygt7ssTxTBPwGoPrnkkDQoaN";
  private static String ACCESS_TOKEN_SECRET = "3px3rnBzWa9bmOmOQPWNMpYc4qdOrOdxGFgp6XiCkEKH";

  public static OAuthAuthorization getAuth() {

    return new OAuthAuthorization(
        new ConfigurationBuilder().setOAuthConsumerKey(CONSUMER_KEY)
            .setOAuthConsumerSecret(CONSUMER_SECRET)
            .setOAuthAccessToken(ACCESS_TOKEN)
            .setOAuthAccessTokenSecret(ACCESS_TOKEN_SECRET)
            .build());
  }
}
```

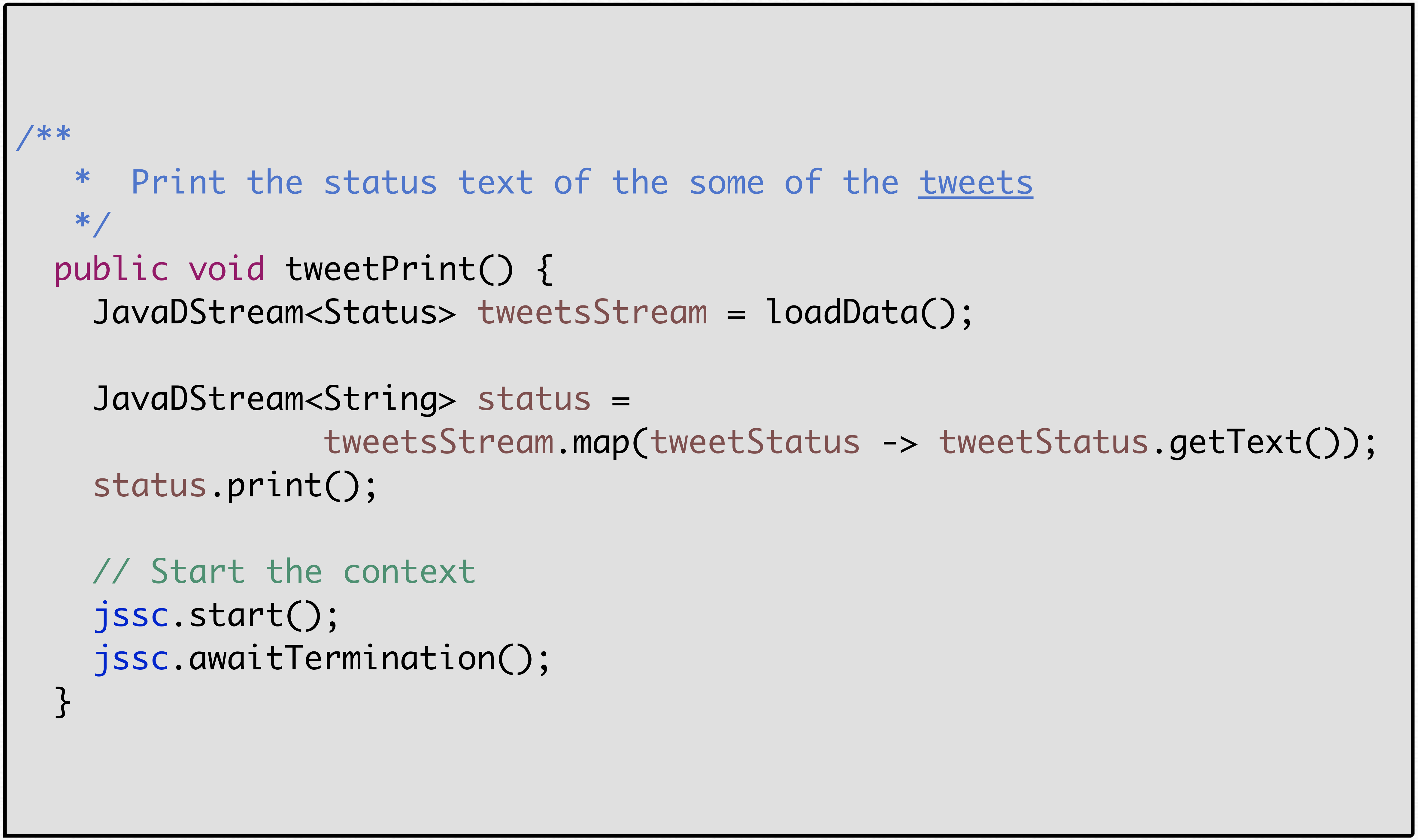
Exercises

SPARK Streaming: StreamingOnTweets

```
public class StreamingOnTweets {

  JavaStreamingContext jssc;

  public JavaDStream<Status> loadData() {
    SparkConf conf = new SparkConf()
        .setAppName("Play with Spark Streaming”);

    // create a java streaming context and define the window (2 seconds batch)
    jssc = new JavaStreamingContext(conf, Durations.seconds(2));

    System.out.println("Initializing Twitter stream...");

    // create a DStream (sequence of RDD). The object tweetsStream is a
    // DStream of tweet statuses:
    // - the Status class contains all information of a tweet
    // See http://twitter4j.org/javadoc/twitter4j/Status.html
    JavaDStream<Status> tweetsStream =
                    TwitterUtils.createStream(jssc, StreamUtils.getAuth());

    return tweetsStream;

  }
```

Exercises

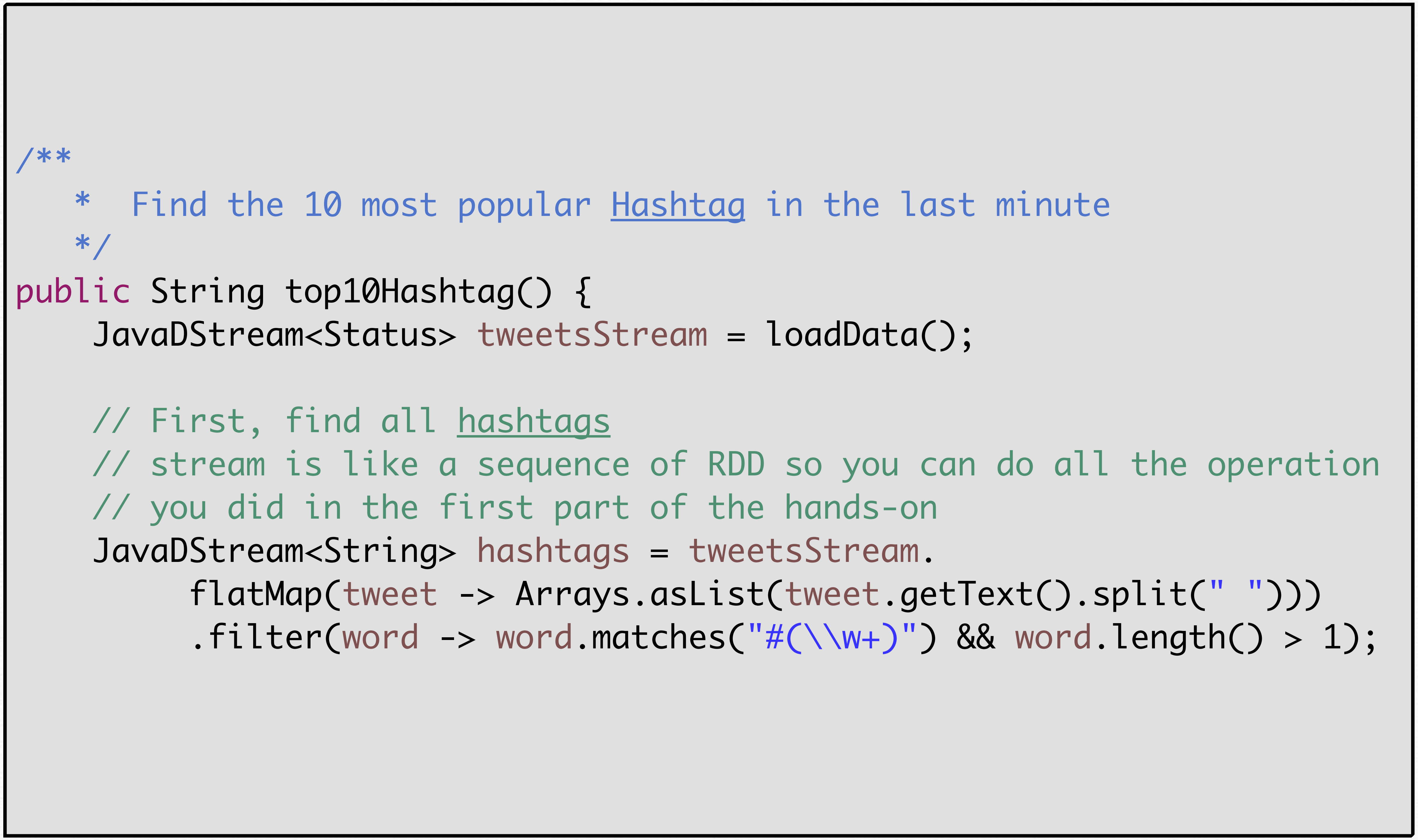
SPARK Streaming: StreamingOnTweets



```
/**
   *  Print the status text of the some of the tweets
   */
  public void tweetPrint() {
    JavaDStream<Status> tweetsStream = loadData();

    JavaDStream<String> status =
                tweetsStream.map(tweetStatus -> tweetStatus.getText());
    status.print();

    // Start the context
    jssc.start();
    jssc.awaitTermination();
  }
```

Exercises

SPARK Streaming: StreamingOnTweets



```
/**
   *  Find the 10 most popular Hashtag in the last minute
   */
public String top10Hashtag() {
    JavaDStream<Status> tweetsStream = loadData();

    // First, find all hashtags
    // stream is like a sequence of RDD so you can do all the operation
    // you did in the first part of the hands-on
    JavaDStream<String> hashtags = tweetsStream.
         flatMap(tweet -> Arrays.asList(tweet.getText().split(" ")))
         .filter(word -> word.matches("#(\\w+)") && word.length() > 1);
```

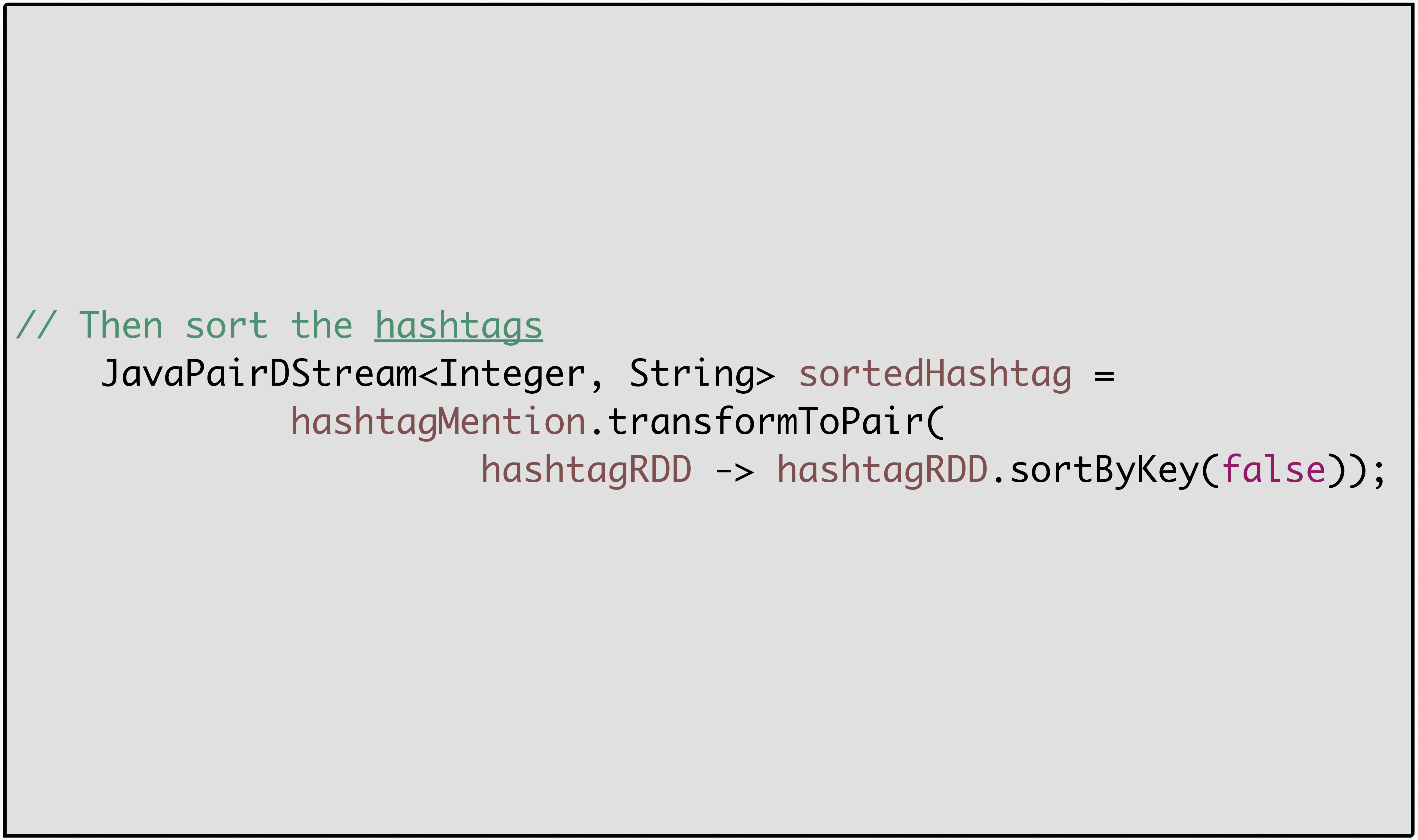
Exercises

SPARK Streaming: StreamingOnTweets

```
// Make a "wordcount" on hashtag
// Reduce last 60 seconds of data
    JavaPairDStream<Integer, String> hashtagMention =
            hashtags.mapToPair(mention -> new Tuple2<>(mention, 1))
            .reduceByKeyAndWindow((x, y) -> x + y, new Duration(60000))
            .mapToPair(pair -> new Tuple2<>(pair._2(), pair._1()));
```

Exercises

SPARK Streaming: StreamingOnTweets



```
// Then sort the hashtags
    JavaPairDStream<Integer, String> sortedHashtag =
             hashtagMention.transformToPair(
                      hashtagRDD -> hashtagRDD.sortByKey(false));
```

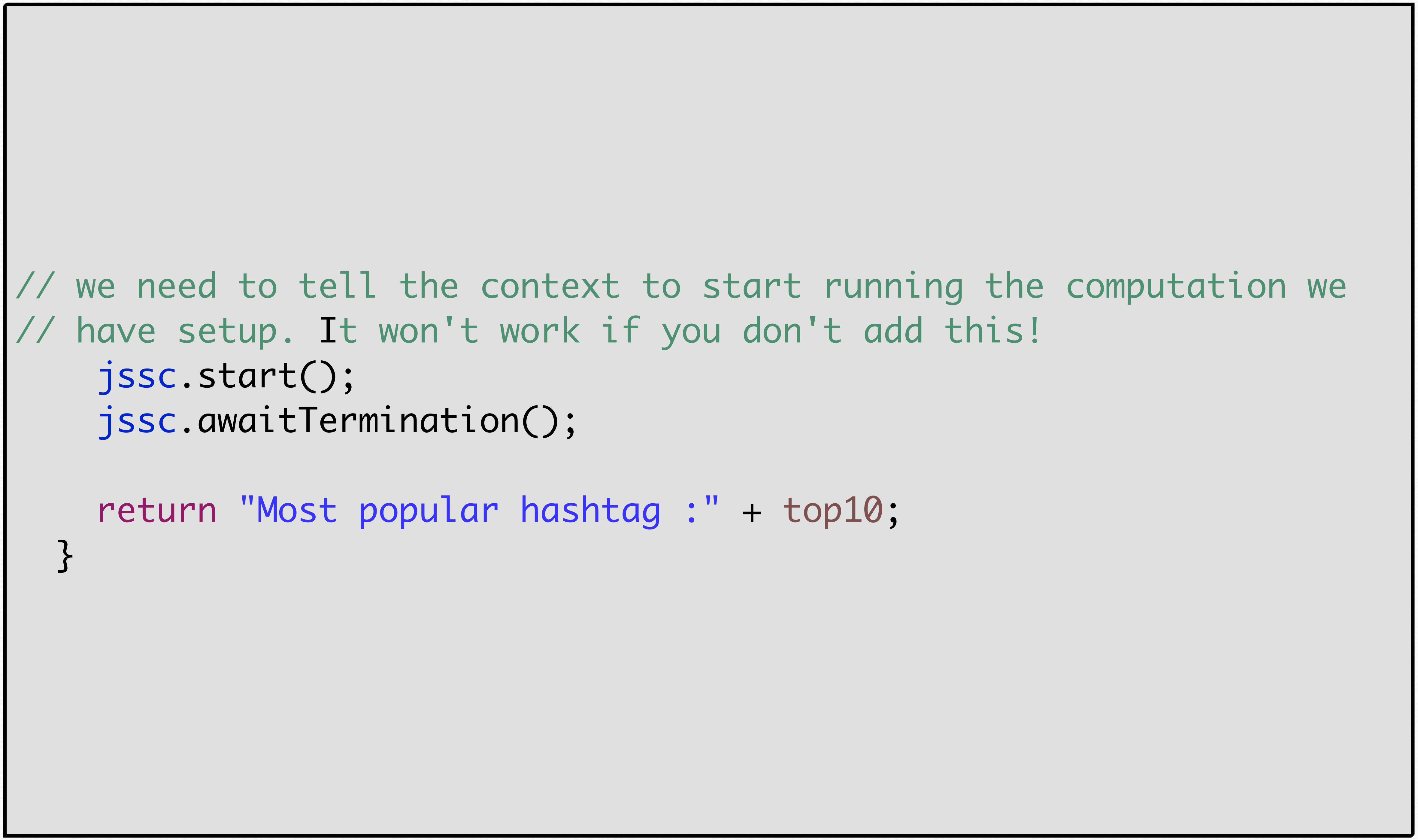
Exercises

- **SPARK Streaming: StreamingOnTweets**

```
// and return the 10 most populars
    List<Tuple2<Integer, String>> top10 = new ArrayList<>();

    sortedHashtag.foreachRDD(rdd -> {
      List<Tuple2<Integer, String>> mostPopular = rdd.take(10);
      top10.addAll(mostPopular);

      return null;
    });
```

Exercises

SPARK Streaming: StreamingOnTweets



```
// we need to tell the context to start running the computation we
// have setup. It won't work if you don't add this!
    jssc.start();
    jssc.awaitTermination();

    return "Most popular hashtag :" + top10;
  }
```

Exercises

SPARK Streaming: execution

```
$HOME/spark-1.6.1-bin-hadoop2.4/bin/spark-submit
                   --class "streaming.StreamingOnTweets"
                   --master local[4]
                   --packages "org.apache.spark:spark-streaming-twitter_2.10:1.5.1"
                   --jars $HOME/spark-in-practice-1.0.jar
                          $HOME/twitter4j-core-3.0.3.jar
```

DataFrame and Streaming

13/05/2019 - Big Data 2019