Programming withSpark

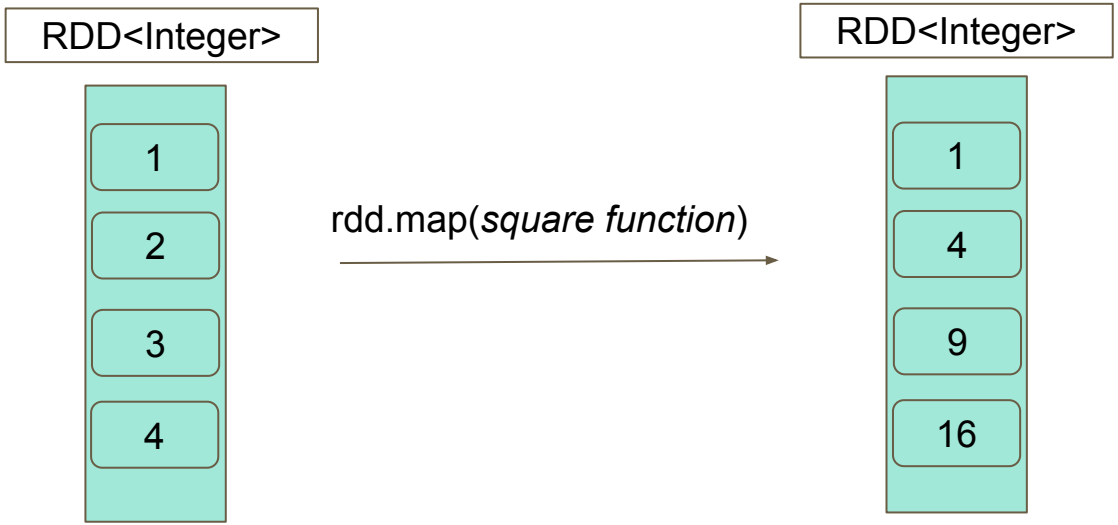
Using RDDs

Basic RDD transformations

- **Single-RDD Transformations**
  - `flatMap` – maps one element in the base RDD to multiple elements
  - `distinct` – filter out duplicates
  - `sortBy` – use provided function to sort
- **Multi-RDD Transformations**
  - `intersection` – create a new RDD with all elements in both original RDDs
  - `union` – add all elements of two RDDs into a single new RDD
  - `zip` – pair each element of the first RDD with the corresponding element of the second

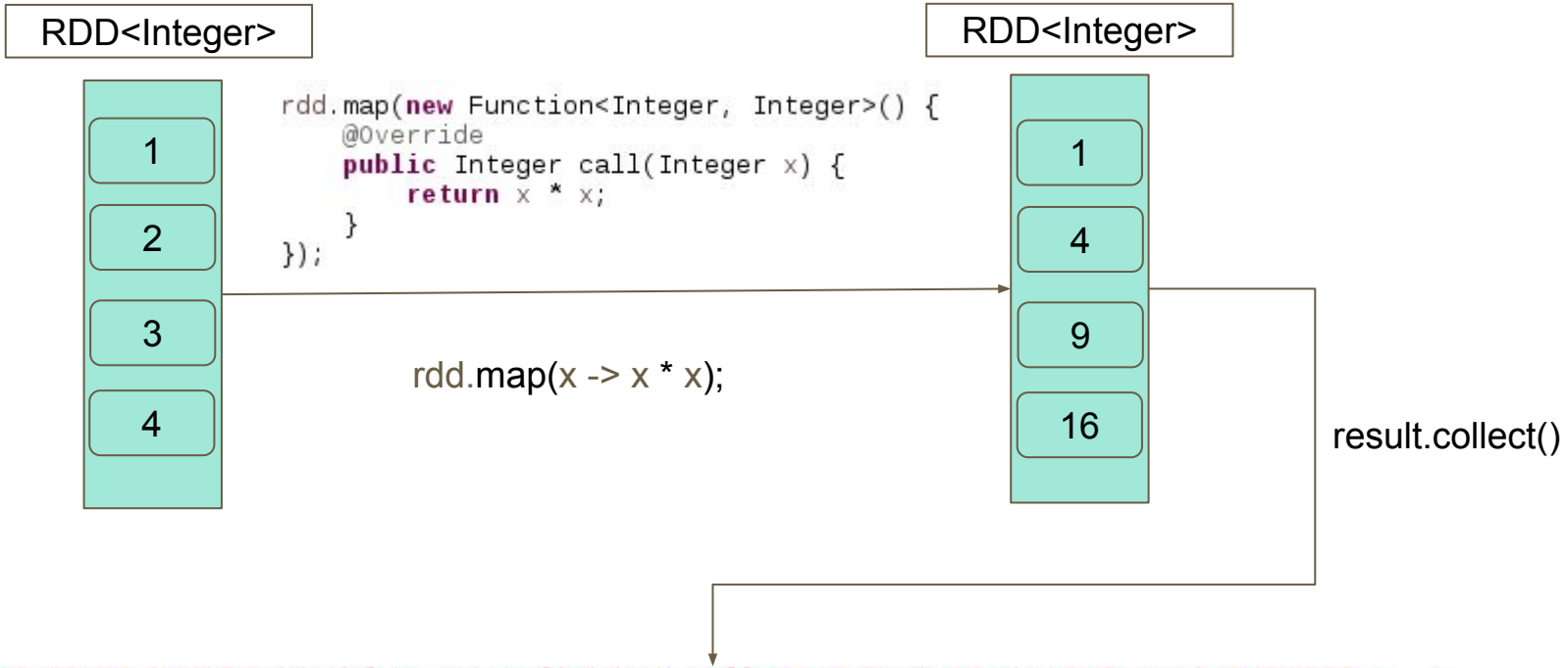
JavaPairRDD methods

map example



```java
public class BasicMap {

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

        JavaSparkContext sc = new JavaSparkContext("local", "basicmap");

        // call parallelize method on an existing Collection
        JavaRDD<Integer> rdd = sc.parallelize(Arrays.asList(1, 2, 3, 4));

        // using map with an anonymous class
        JavaRDD<Integer> result = rdd.map(new Function<Integer, Integer>() {
            @Override
            public Integer call(Integer x) {
                return x * x;
            }
        });

        // using map with lambdas
        JavaRDD<Integer> equivalentResult = rdd.map(x -> x * x);

        System.out.println("anonymous function results:  " + StringUtils.join(result.collect(), ",")
                + ", lambda results:  " + StringUtils.join(equivalentResult.collect(), ","));
    }
}
```

map results with collect




```
16/11/10 12:36:54 INFO DAGScheduler: Job 1 finished: collect at BasicMap.java:45, took 0.011933 s
anonymous function results:  1,4,9,16, lambda results:  1,4,9,16
16/11/10 12:36:54 INFO SparkContext: Invoking stop() from shutdown hook
```

snippet of info printing out to console

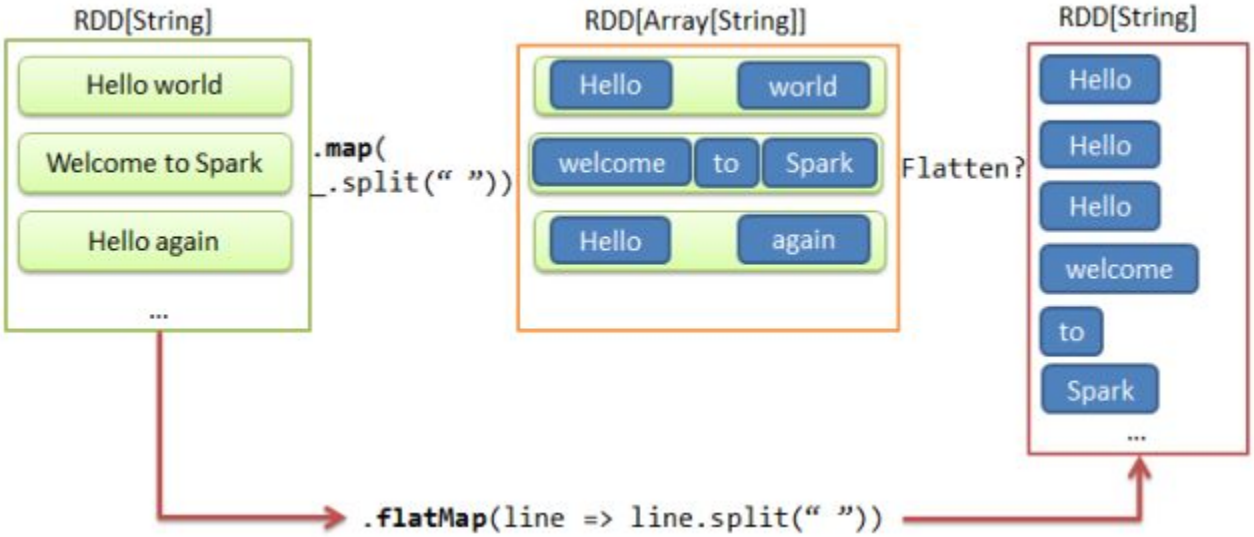
To run example:

click on `BasicMap.java`

select “Run As” -> select “Java application”

Note: you don’t need to specify input because the input is created at the beginning of the job.

flatMap


RDD[String]
Hello world
Welcome to Spark
Hello again
...
.map(
_.split(" "))
RDD[Array[String]]
Hello
world
welcome
to
Spark
Hello
again
Flatten?
RDD[String]
Hello
Hello
Hello
welcome
to
Spark
...
.flatMap(line => line.split(" "))

```java
public class BasicFlatMap {

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

        if (args.length != 1) {
            System.out.println("For running in Eclipse - the argument is:  inputFile");
            System.exit(-1);
        }

        JavaSparkContext sc = new JavaSparkContext("local", "basicflatmap");

        JavaRDD<String> rdd = sc.textFile(args[0]);

        JavaRDD<String> words = rdd.flatMap(new FlatMapFunction<String, String>() {
            @Override
            public Iterable<String> call(String x) {
                return Arrays.asList(x.split(" "));
            }
        });

        Map<String, Long> result = words.countByValue();

        for (Entry<String, Long> entry : result.entrySet()) {
            if (entry.getValue() > 50)
                System.out.println(entry.getKey() + ":" + entry.getValue());
        }
    }
}
```

FlatMapFunction

FlatMapFunction is class

Used to encapsulate a function for Spark's flatMap method

```
JavaRDD<String> words = rdd.flatMap(new FlatMapFunction<String, String>() {
    @Override
    public Iterable<String> call(String x) {
        return Arrays.asList(x.split(" "));
    }
});
```

Takes in a single value, writes out an array of values.

```java
public class BasicFlatMap_Java8 {

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

        if (args.length != 1) {
            System.out.println("For running in Eclipse - the argument is:  inputFile");
            System.exit(-1);
        }

        JavaSparkContext sc = new JavaSparkContext("local", "basicflatmap");

        JavaRDD<String> rdd = sc.textFile(args[0]);

        JavaRDD<String> words = rdd.flatMap(x -> Arrays.asList(x.split(" ")));

        Map<String, Long> result = words.countByValue();

        for (Entry<String, Long> entry : result.entrySet()) {
            if (entry.getValue() > 50)
                System.out.println(entry.getKey() + ":" + entry.getValue());
        }
    }
}
```

Using anonymous classes vs lambdas

Using flatMap with FlatMapFunction as an anonymous class

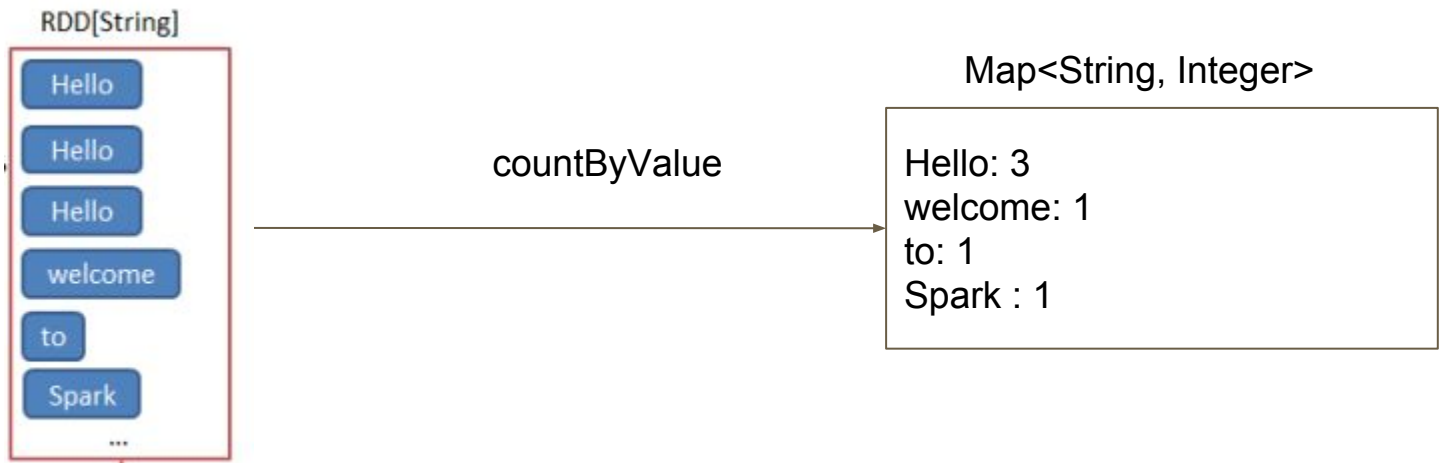
```
JavaRDD<String> words = rdd.flatMap(new FlatMapFunction<String, String>() {
    @Override
    public Iterable<String> call(String x) {
        return Arrays.asList(x.split(" "));
    }
});
```

Using flatMap with lambdas

```
JavaRDD<String> words = rdd.flatMap(x -> Arrays.asList(x.split(" ")));
```

Action: CountByValue

Returns a Map containing a count for each distinct input value



When this action runs, will it kick off a shuffle-sort?

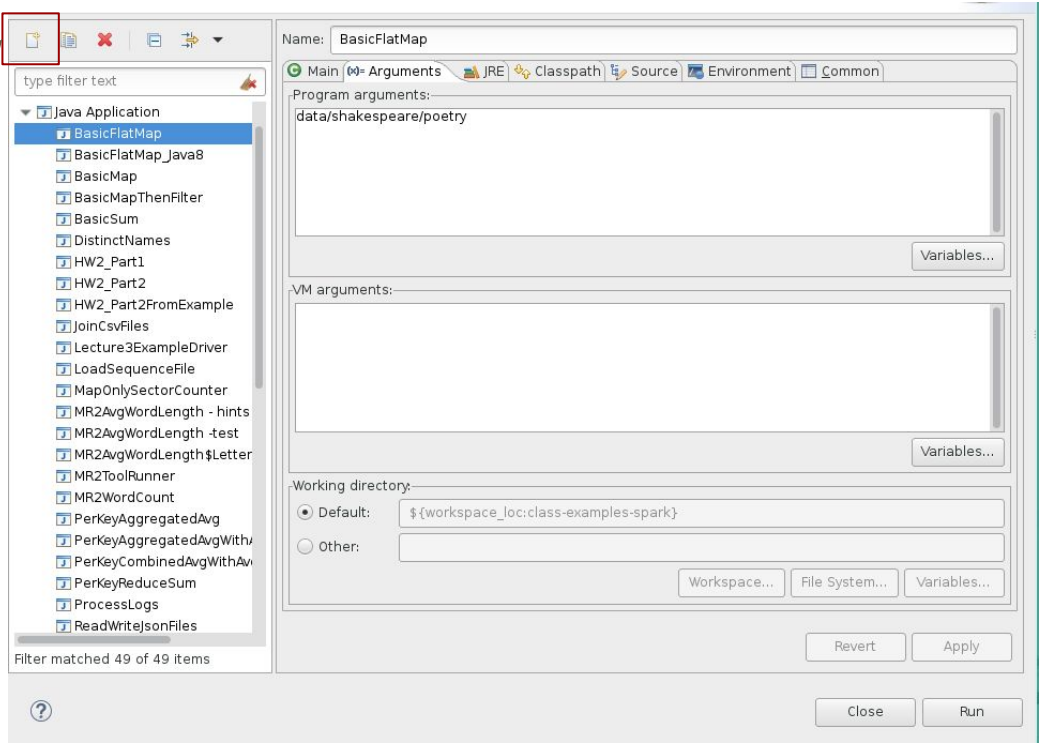
To run example:

In Package Explorer:

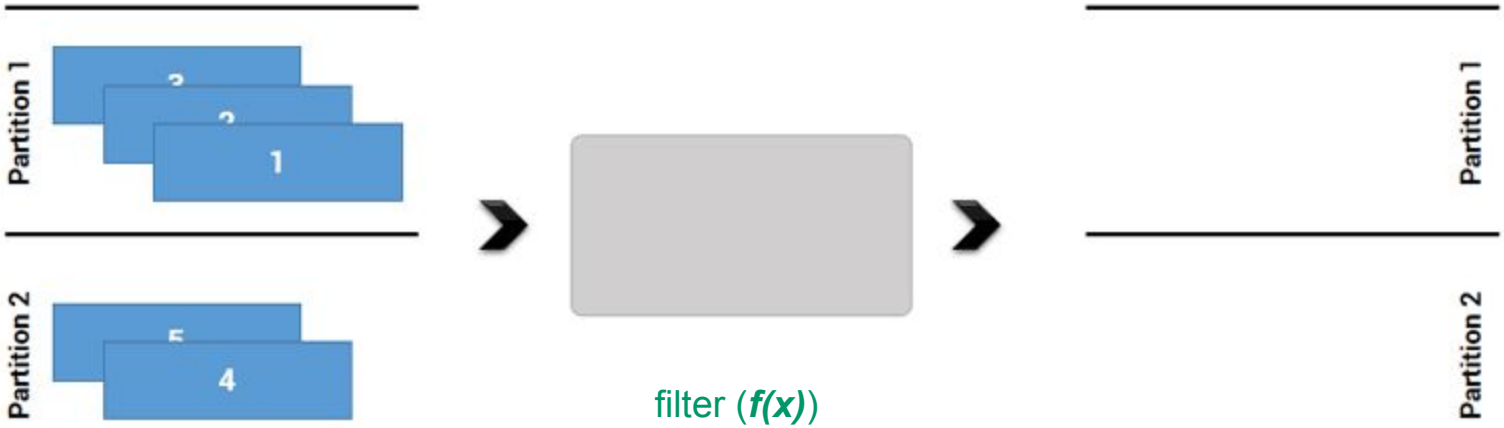
Click on **`BasicFlatMap.java`** or **`BasicFlatMap_Java8.java`**

Select “Run As” -> “Run Configurations”

- Click on new config
- Under Arguments tab use
  - data/shakespeare/poetry



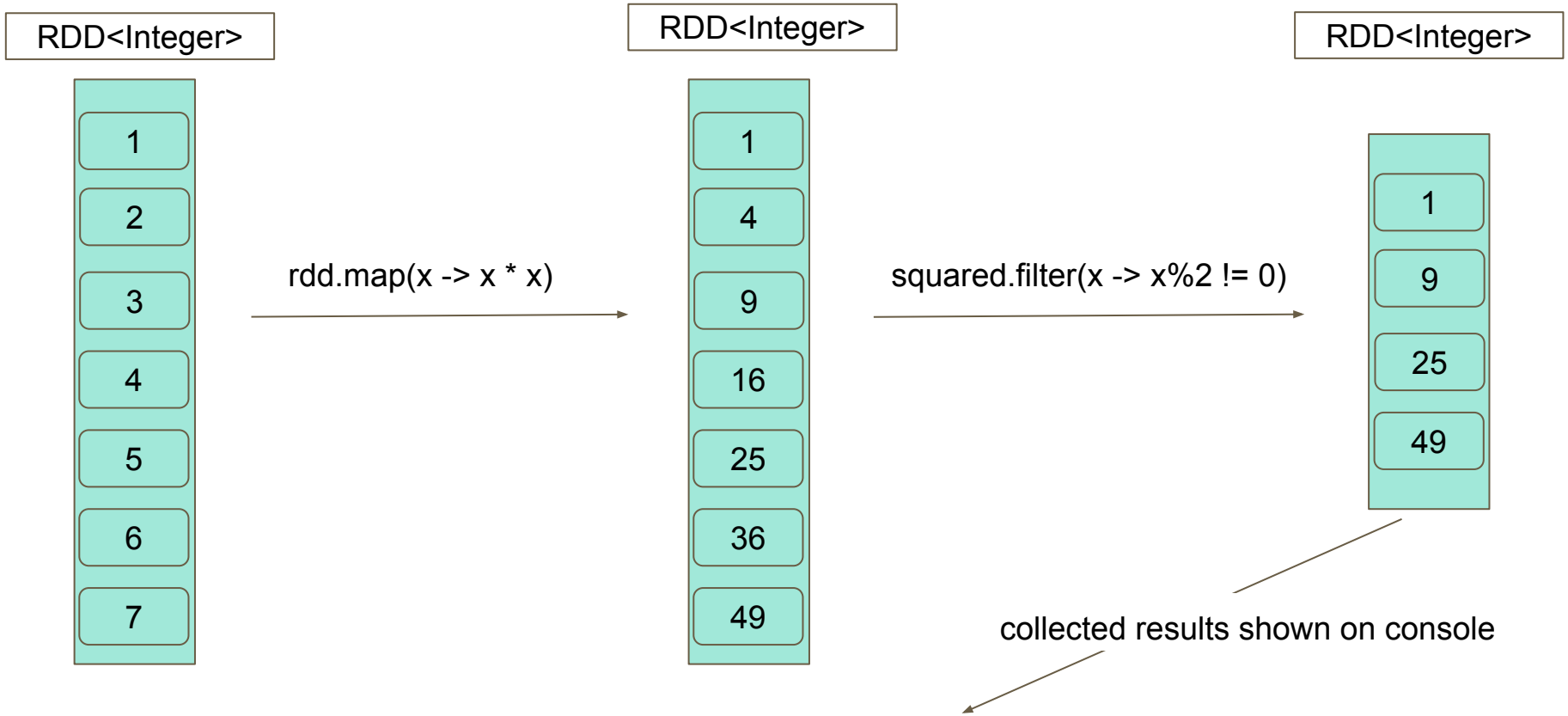
filter



filter (***f(x)***)

```
filter(x-> x%2 != 0)
```

```java
public class BasicMapThenFilter {

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

        JavaSparkContext sc = new JavaSparkContext("local", "basicmapfilter");

        JavaRDD<Integer> rdd = sc.parallelize(Arrays.asList(1, 2, 3, 4, 5, 6, 7));

        JavaRDD<Integer> squared = rdd.map(new Function<Integer, Integer>() {
            @Override
            public Integer call(Integer x) {
                return x * x;
            }
        });
        JavaRDD<Integer> result = squared.filter(new Function<Integer, Boolean>() {
            @Override
            public Boolean call(Integer x) {
                return x % 2 != 0;
            }
        });
        JavaRDD<Integer> equivalentResult = squared.filter(x -> x % 2 != 0);

        System.out.println("anonymous function results:  " + result.collect() + "\nlambda results:  "
                + equivalentResult.collect());
    }
}
```




```
16/11/10 13:14:33 INFO DAGScheduler: Job 1 finished: collect at BasicMapThenFilter.java:32, took 0.014121 s
anonymous function results:  [1, 9, 25, 49]
lambda results:  [1, 9, 25, 49]
16/11/10 13:14:33 INFO SparkContext: Invoking stop() from shutdown hook
```

To run example:

click on `BasicMapThenFilter.java`

select “Run As” -> select “Java application”

Note: you don’t need to specify input because the input is created at the beginning of the job.

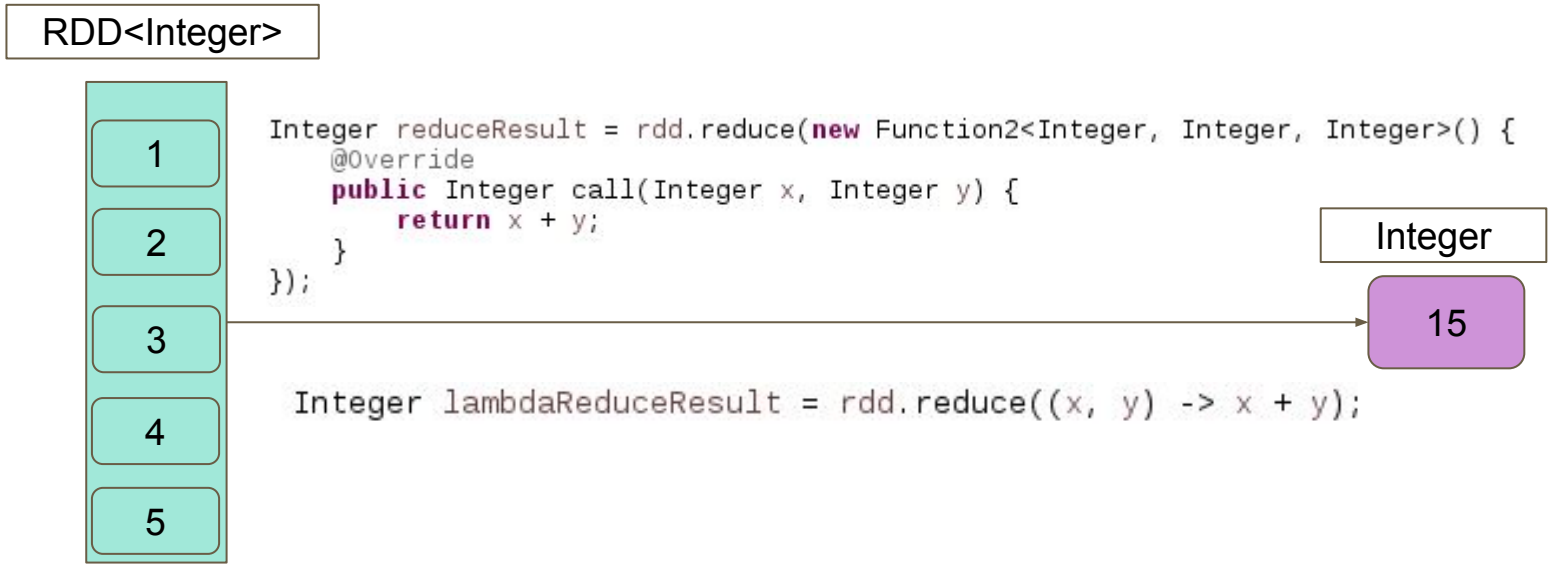
reduce (an Action)

reduce (***f(x,y)***)

```
reduce((a,n) -> a + n);
```

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

        JavaSparkContext sc = new JavaSparkContext("local", "basicSum");

        JavaRDD<Integer> rdd = sc.parallelize(Arrays.asList(1, 2, 3, 4));

        Integer reduceResult = rdd.reduce(new Function2<Integer, Integer, Integer>() {
            @Override
            public Integer call(Integer x, Integer y) {
                return x + y;
            }
        });

        // reduce with lambda
        Integer lambdaReduceResult = rdd.reduce((x, y) -> x + y);

        System.out.println(
                "Comparing results from the two representations:  " + reduceResult + " vs " + lambdaReduceResult);
    }
}
```

reduce results




```
Integer reduceResult = rdd.reduce(new Function2<Integer, Integer, Integer>() {
    @Override
    public Integer call(Integer x, Integer y) {
        return x + y;
    }
});
```

```
Integer lambdaReduceResult = rdd.reduce((x, y) -> x + y);
```

To run example:

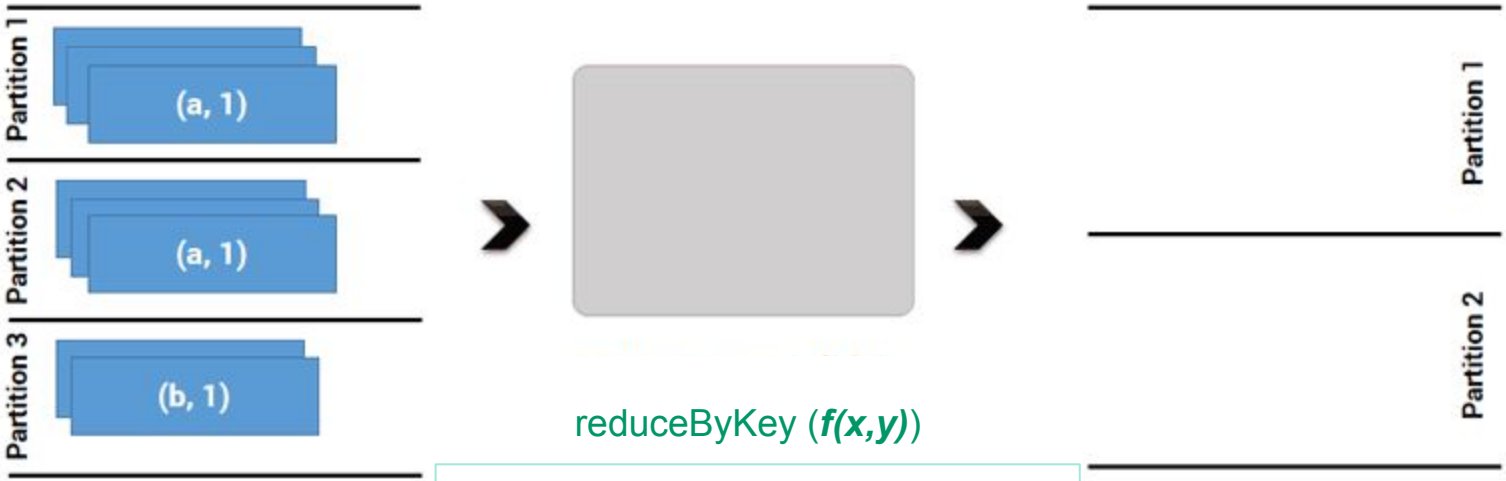
click on `BasicSum.java`

select “Run As” -> select “Java application”

Note: you don’t need to specify input because the input is created at the beginning of the job.

JavaPairRDD methods

reduceByKey



reduceByKey (***f(x,y)***)

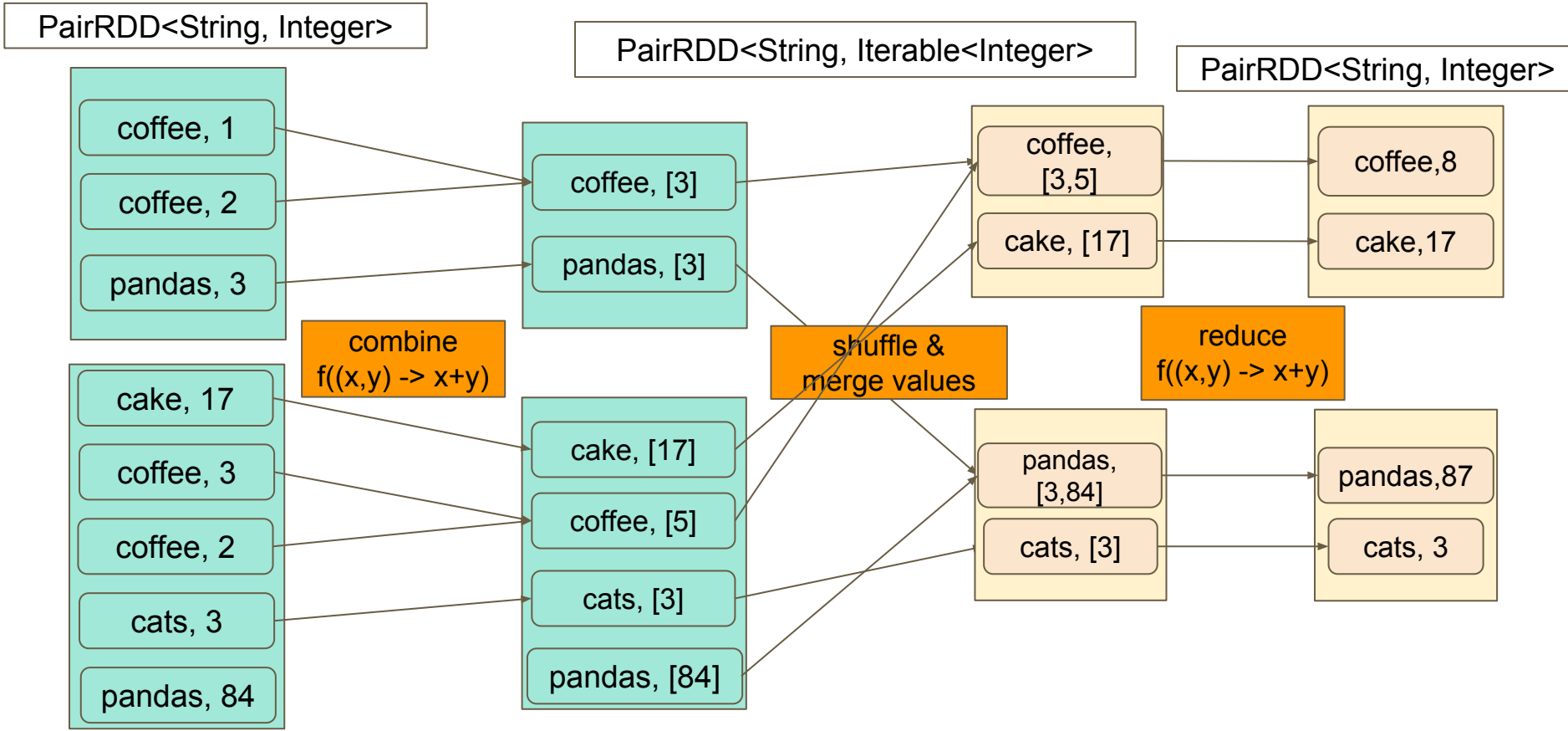
```
reduceByKey((x,y) -> x + y);
```

Note on processing

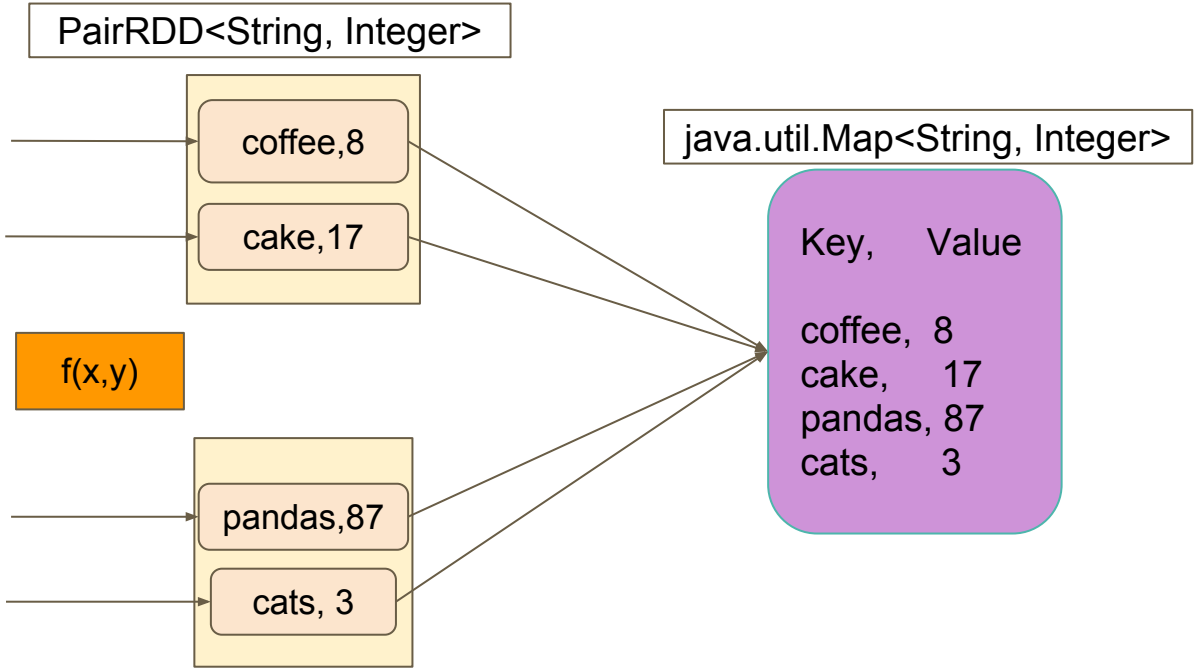
- step 1: groups values by key
- step 2: apply f(x,y) to group

```java
public final class PerKeyReduceSum {

    public static void main(String args[]) {

        JavaSparkContext sc = new JavaSparkContext("local", "PerKeyAggregatedAvg");

        List<Tuple2<String, Float>> inputList = new ArrayList();
        inputList.add(new Tuple2("coffee", 1f));
        inputList.add(new Tuple2("coffee", 2f));
        inputList.add(new Tuple2("pandas", 3f));

        JavaPairRDD<String, Float> input = sc.parallelizePairs(inputList);

        Map<String, Float> resultMap = computeSum(input);

        for (Entry<String, Float> entry : resultMap.entrySet()) {
            System.out.println(entry.getKey() + ":" + entry.getValue());
        }

        sc.stop();
    }

    static Map<String, Float> computeSum(JavaPairRDD<String, Float> input) {

        JavaPairRDD sum = input.reduceByKey((x, y) -> (x + y));
        return sum.collectAsMap();

    }

}
```

reduceByKey with shuffle



collectAsMap on the reduceByKey RDD



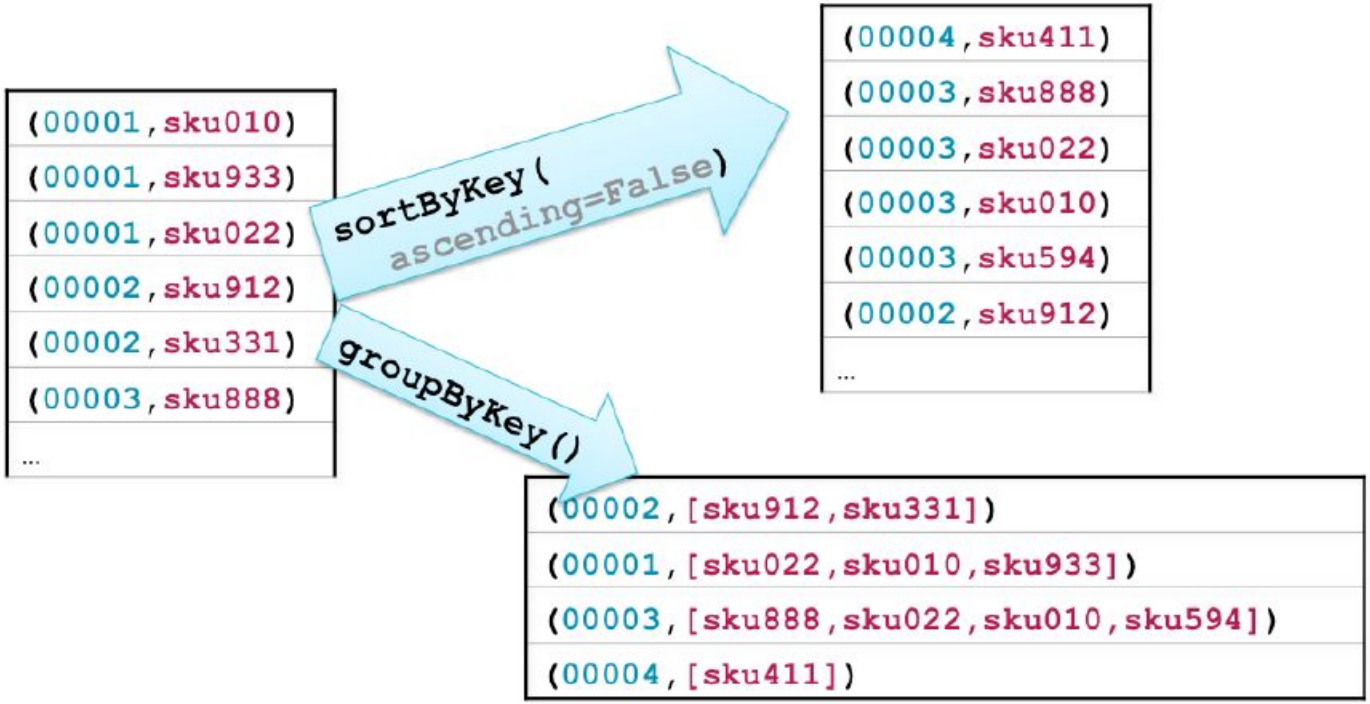
To run example:

click on `PerKeyReduceSum.java`

select “Run As” -> select “Java application”

Note: you don’t need to specify input because the input is created at the beginning of the job.

sortByKey and groupByKey



mapValues and reduceByKey => aggregateByKey

key	value
panda	0
pink	3
pirate	3
panda	1
pink	4

mapValues

f1(x)

f(x ->(x, 1))

key	value
panda	(0, 1)
pink	(3, 1)
pirate	(3, 1)
panda	(1, 1)
pink	(4, 1)

reduceByKey

f2(a,b)

(x, y) -> (x._1 + y._1, x._2 + y._2)

key	value
panda	(1, 2)
pink	(7, 2)
pirate	(3, 1)

takes in a single value,
returns a pair

takes in a pair,
returns a pair

```java
public final class PerKeyMapValueReduceByKeyAvg {

    public static void main(String args[]) {

        JavaSparkContext sc = new JavaSparkContext("local", "PerKeyAggregatedAvg");

        List<Tuple2<String, Float>> inputList = new ArrayList();
        inputList.add(new Tuple2("panda", 0f));
        inputList.add(new Tuple2("pink", 3f));
        inputList.add(new Tuple2("pirate", 3f));
        inputList.add(new Tuple2("panda", 1f));
        inputList.add(new Tuple2("pink", 4f));

        JavaPairRDD<String, Float> input = sc.parallelizePairs(inputList);

        Map<String, Tuple2<Float, Float>> resultMap = computeAvg(input);

        for (Entry<String, Tuple2<Float, Float>> entry : resultMap.entrySet()) {
            Tuple2<Float, Float> result = entry.getValue();
            Float avg = result._1 / result._2;
            System.out.println(entry.getKey() + ":" + avg);
        }

        sc.stop();
    }

    static Map<String, Tuple2<Float, Float>> computeAvg(JavaPairRDD<String, Float> input) {

        Tuple2<Float, Float> zeroValue = new Tuple2(0f, 0f);
        JavaPairRDD<String, Tuple2<Float, Float>> mapValues = input.mapValues(x -> new Tuple2<Float, Float>(x, 1f));
        JavaPairRDD<String, Tuple2<Float, Float>> avgCounts = mapValues
                .reduceByKey((x, y) -> new Tuple2<Float, Float>(x._1 + y._1, x._2 + y._2));
        return avgCounts.collectAsMap();

    }

}
```

mapValues and reduceByKey

```
static Map<String, Tuple2<Float, Float>> computeAvg(JavaPairRDD<String, Float> input) {

    Tuple2<Float, Float> zeroValue = new Tuple2(0f, 0f);
    JavaPairRDD<String, Tuple2<Float, Float>> mapValues = input.mapValues(x -> new Tuple2<Float, Float>(x, 1f));
    JavaPairRDD<String, Tuple2<Float, Float>> avgCounts = mapValues
            .reduceByKey((x, y) -> new Tuple2<Float, Float>(x._1 + y._1, x._2 + y._2));
    return avgCounts.collectAsMap();

}
```

f(x,y) for reduceByKey
==
f(x,y) for combineOp

aggregateByKey

```
static Map<String, Tuple2> computeAvg(JavaPairRDD<String, Float> input) {

    Tuple2<Float, Float> zeroValue = new Tuple2(0f, 0f);
    JavaPairRDD avgCounts = input.aggregateByKey(zeroValue, (a, x) -> new Tuple2(a._1 + x, a._2 + 1f),
            (a, b) -> new Tuple2(a._1 + b._1, a._2 + b._2));
    return avgCounts.collectAsMap();

}
```

Making the combineOp function understandable

(x, y) -> (x._1 + y._1, x._2 + y._2)

uses closures

```
Function2<AvgSer, AvgSer, AvgSer> combine = new Function2<AvgSer, AvgSer, AvgSer>() {
    @Override
    public AvgSer call(AvgSer a, AvgSer b) {
        a.total_ += b.total_;
        a.num_ += b.num_;
        return a;
    }
};
```

uses AvgSer class

mapValues and reduceByKey

```
static Map<String, Tuple2<Float, Float>> computeAvg(JavaPairRDD<String, Float> input) {

    Tuple2<Float, Float> zeroValue = new Tuple2(0f, 0f);
    JavaPairRDD<String, Tuple2<Float, Float>> mapValues = input.mapValues(x -> new Tuple2<Float, Float>(x, 1f));
    JavaPairRDD<String, Tuple2<Float, Float>> avgCounts = mapValues
            .reduceByKey((x, y) -> new Tuple2<Float, Float>(x._1 + y._1, x._2 + y._2));
    return avgCounts.collectAsMap();

}
```

f(x) for mapValue

seqOp

aggregateByKey

```
static Map<String, Tuple2> computeAvg(JavaPairRDD<String, Float> input) {

    Tuple2<Float, Float> zeroValue = new Tuple2(0f, 0f);
    JavaPairRDD avgCounts = input.aggregateByKey(zeroValue, (a, x) -> new Tuple2(a._1 + x, a._2 + 1f),
            (a, b) -> new Tuple2(a._1 + b._1, a._2 + b._2));
    return avgCounts.collectAsMap();

}
```

```java
public final class PerKeyAggregatedAvg {

    public static void main(String args[]) {

        JavaSparkContext sc = new JavaSparkContext("local", "PerKeyAggregatedAvg");

        List<Tuple2<String, Float>> inputList = new ArrayList();
        inputList.add(new Tuple2("panda", 0f));
        inputList.add(new Tuple2("pink", 3f));
        inputList.add(new Tuple2("pirate", 3f));
        inputList.add(new Tuple2("panda", 1f));
        inputList.add(new Tuple2("pink", 4f));

        JavaPairRDD<String, Float> input = sc.parallelizePairs(inputList);

        Map<String, Tuple2> resultMap = computeAvg(input);

        for (Entry<String, Tuple2> entry : resultMap.entrySet()) {
            Tuple2<Float, Float> result = entry.getValue();
            Float avg = result._1 / result._2;
            System.out.println(entry.getKey() + ":" + avg);
        }

        sc.stop();
    }

    static Map<String, Tuple2> computeAvg(JavaPairRDD<String, Float> input) {

        Tuple2<Float, Float> zeroValue = new Tuple2(0f, 0f);
        JavaPairRDD avgCounts = input.aggregateByKey(zeroValue, (a, x) -> new Tuple2(a._1 + x, a._2 + 1f),
                (a, b) -> new Tuple2(a._1 + b._1, a._2 + b._2));
        return avgCounts.collectAsMap();

    }

}
```

Running the examples

mapValues and reduceByKey aggregation:

```
PerKeyMapValueReduceByKeyAvg.java
```

aggregateByKey using lambdas:

```
PerKeyAggregatedAvg
```

aggregateByKey using AvgSer class and explicit functions:

```
PerKeyAggregatedAvgWithAvgSer
```

No need to define a Run Configuration, these all create their own data.

Discuss Homework 2