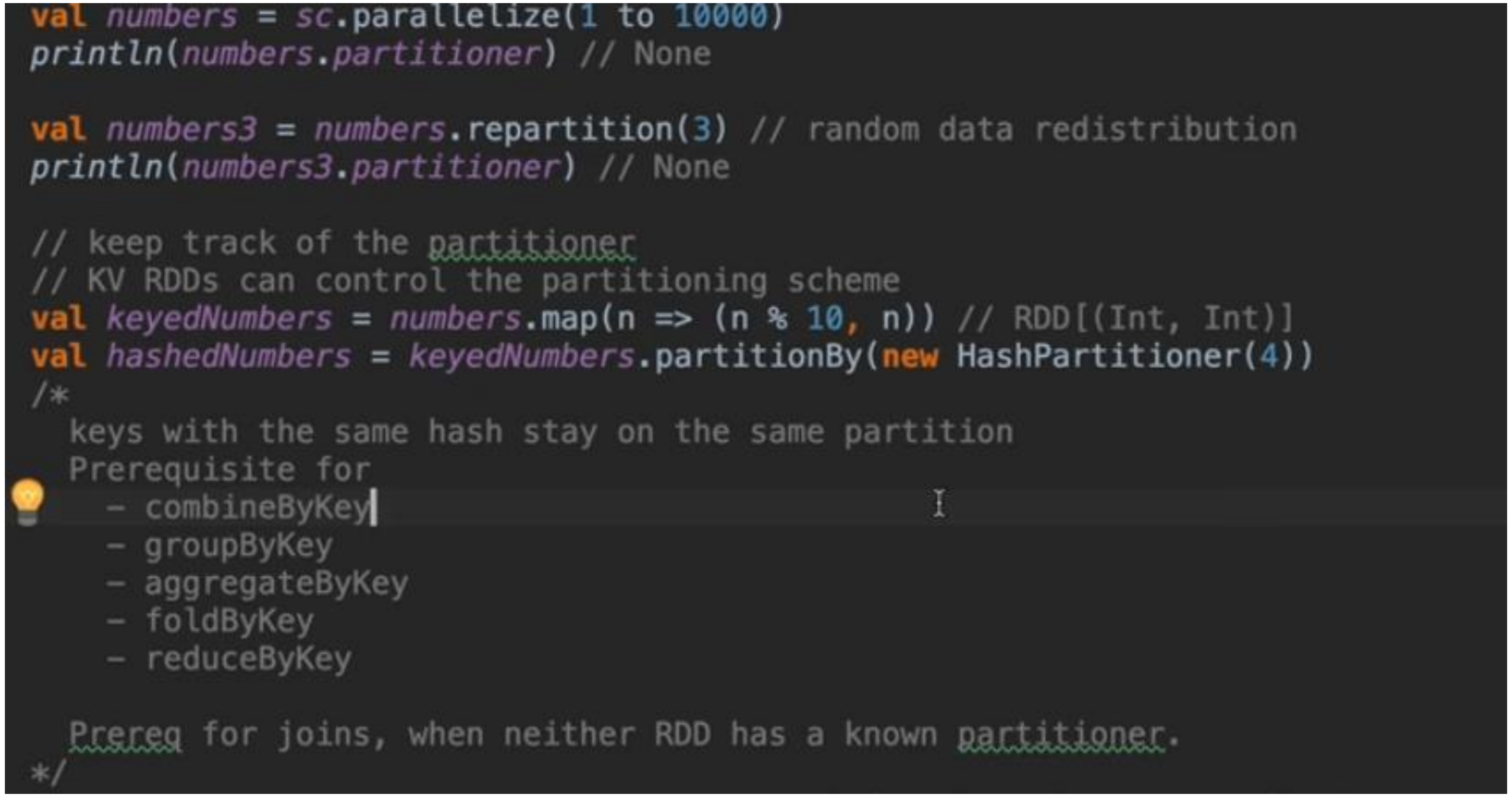
Hash Partitioner



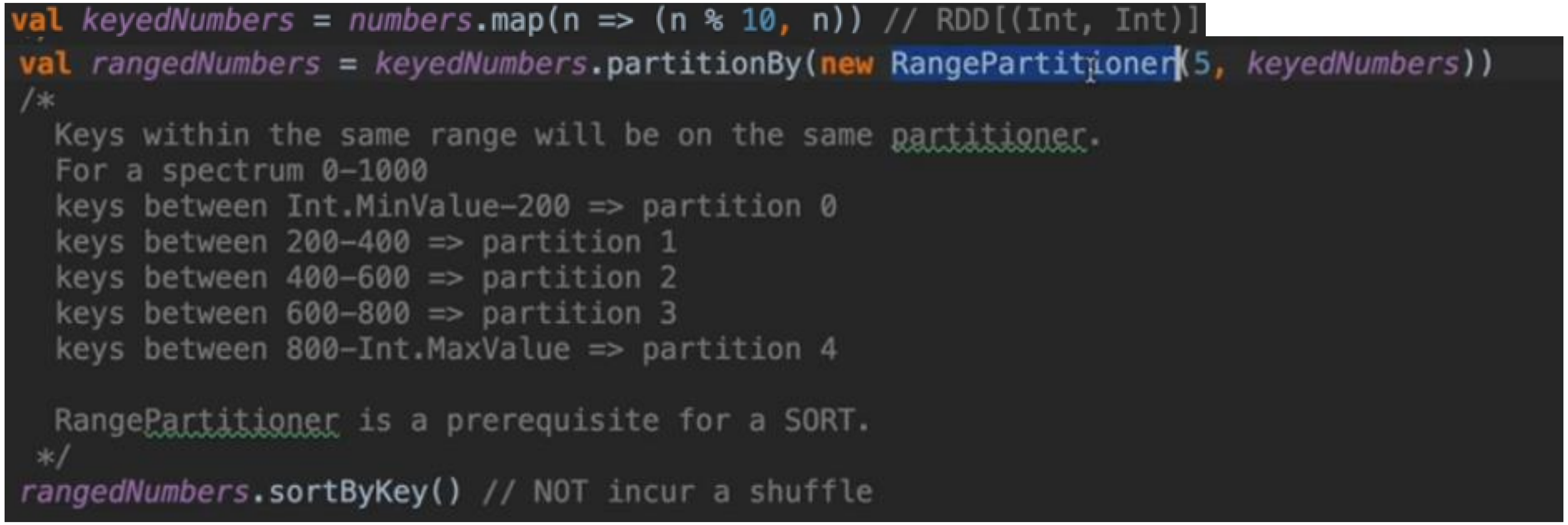
```
val numbers = sc.parallelize(1 to 10000)
println(numbers.partitioner) // None

val numbers3 = numbers.repartition(3) // random data redistribution
println(numbers3.partitioner) // None

// keep track of the partitioner
// KV RDDs can control the partitioning scheme
val keyedNumbers = numbers.map(n => (n % 10, n)) // RDD[(Int, Int)]
val hashedNumbers = keyedNumbers.partitionBy(new HashPartitioner(4))
/*
  keys with the same hash stay on the same partition
  Prerequisite for
    - combineByKey
    - groupByKey
    - aggregateByKey
    - foldByKey
    - reduceByKey

  Prereq for joins, when neither RDD has a known partitioner.
*/
```

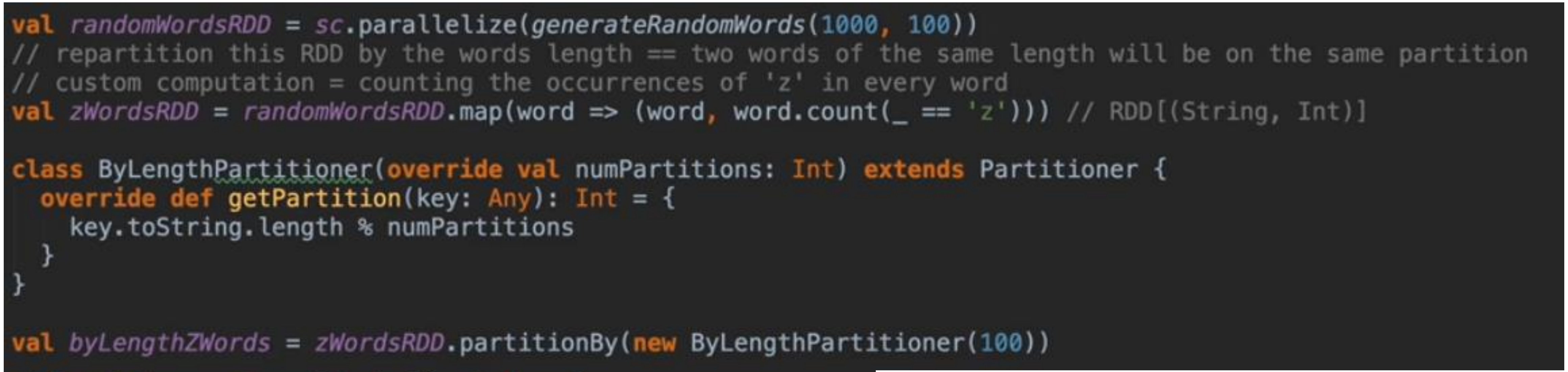
Range Partitioner



```scala
val keyedNumbers = numbers.map(n => (n % 10, n)) // RDD[(Int, Int)]
val rangedNumbers = keyedNumbers.partitionBy(new RangePartitioner(5, keyedNumbers))
/*
  Keys within the same range will be on the same partitioner.
  For a spectrum 0-1000
  keys between Int.MinValue-200 => partition 0
  keys between 200-400 => partition 1
  keys between 400-600 => partition 2
  keys between 600-800 => partition 3
  keys between 800-Int.MaxValue => partition 4

  RangePartitioner is a prerequisite for a SORT.
 */
rangedNumbers.sortByKey() // NOT incur a shuffle
```

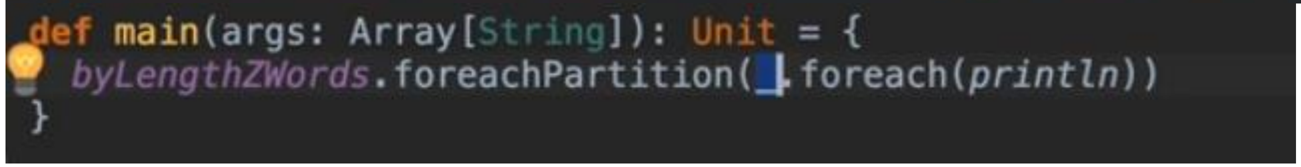
Custom Partitioner



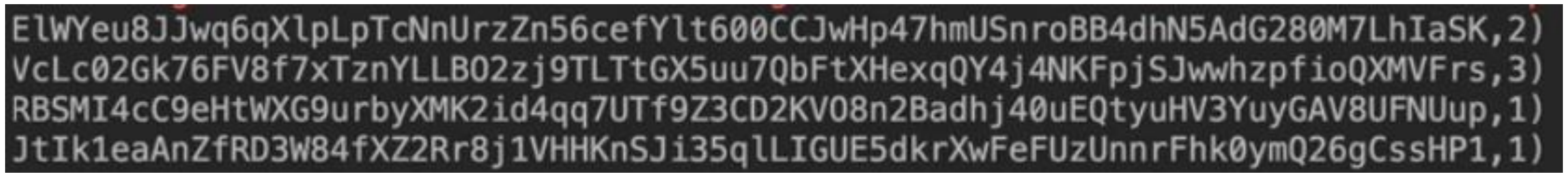
```
val randomWordsRDD = sc.parallelize(generateRandomWords(1000, 100))
// repartition this RDD by the words length == two words of the same length will be on the same partition
// custom computation = counting the occurrences of 'z' in every word
val zWordsRDD = randomWordsRDD.map(word => (word, word.count(_ == 'z'))) // RDD[(String, Int)]

class ByLengthPartitioner(override val numPartitions: Int) extends Partitioner {
  override def getPartition(key: Any): Int = {
    key.toString.length % numPartitions
  }
}

val byLengthZWords = zWordsRDD.partitionBy(new ByLengthPartitioner(100))
```



```
def main(args: Array[String]): Unit = {
  byLengthZWords.foreachPartition(_.foreach(println))
}
```



```
ElWYeu8JJwq6qXlpLpTcNnUrzZn56cefYlt600CCJwHp47hmUSnroBB4dhN5AdG280M7LhIaSK,2)
VcLc02Gk76FV8f7xTznYLLB02zj9TLTtGX5uu7QbFtXHexqQY4j4NKFpjSJwwhzpfioQXMVFrs,3)
RBSMI4cC9eHtWXG9urbyXMK2id4qq7UTf9Z3CD2KVO8n2Badhj40uEQtyuHV3YuyGAV8UFNUup,1)
JtIk1eaAnZfRD3W84fXZ2Rr8j1VHHKnSJi35qlLIGUE5dkrXwFeFUzUnnrFhk0ymQ26gCssHP1,1)
```

Partitioners

Decide which record stays on which partition (key-value RDDs only)

- hash partitioning = same hash, same partition
- range partitioning = same range, same partition
- custom partitioning = you decide where each key stays, for custom computations

Partitioning has advantages and does not incur shuffles

- hash partitioning for joins and by-key functions
- range partitioning for sorts

DFs cannot control partitioning logic, but follow rules

- sort/orderBy => RangePartitioning
- aggregation by key => HashPartitioning
- join => both DFs obey HashPartitioning
- repartition with a number => RoundRobinPartitioning
- repartition by column => HashPartitioning

Joins Speedup

Make sure the same keys are on the same partition

- RDDs must have the same partitioner
- otherwise, Spark will pick one

Co-partitioning: RDDs share the same partitioner

- no shuffle involved for joins

Colocation: RDD partitions are already loaded in memory

- fastest join possible