# Streams

## Collecting Results

# Collecting Results from the Stream

- we've already learned about `collect()` terminal operation
- now we'll introduce some other predefined collectors
    - there are `static` methods on the `Collectors` class
- let's first list the methods and then work on some examples...

# Common grouping/partitioning collectors (1/6)

| Collector | Description | Return value (w/ collect) |
|---|---|---|
| `averagingDouble( ToDoubleFunction f)` | Calculates average for doubles | `Double` |
| `averagingInt( ToIntFunction f)` | Calculates average for integers | `Double` |
| `averagingLong( ToLongFunction f)` | Calculates average for longs | `Double` |
| `counting()` | Counts numbers of elements | `Long` |
| `filtering(Predicate p, Collector c)` | Applies filter before calling downstream collector | `R` |

# Common grouping/partitioning collectors (2/6)

| Collector | Description | Return value (w/ collect) |
|---|---|---|
| `groupingBy(Function f)` | Creates map | `Map<K, List<T>>` |
| `groupingBy(Function f,`<br>`  Collector dc)` | Creates map where downstream collector is provided | `Map<K, List<T>>` |
| `groupingBy(Function f,`<br>`  Supplier s`<br>`  Collector dc)` | Creates map where both map supplier and downstream collector are provided | `Map<K, List<T>>` |
| `joining(CharSequence cs)` | Creates `String` using `cs` as delimiter | `String` |

# Common grouping/partitioning collectors (3/6)

| Collector | Description | Return value (w/ collect) |
|---|---|---|
| `maxBy(Comparator c)`<br>`minBy(Comparator c)` | Finds largest/smallest elements | `Optional<T>` |
| `mapping(Function f,`<br>`  Collector dc)` | Adds another level of collectors | `Collector` |
| `partitioningBy(`<br>`  Predicate p)` | Creates map grouping | `Map<Boolean, List<T>>` |
| `partitioningBy(`<br>`  Predicate p,`<br>`  Collector dc)` | Creates map grouping | `Map<Boolean, List<T>>` |

# Common grouping/partitioning collectors (4/6)

| Collector | Description | Return value (w/ collect) |
|---|---|---|
| `summarizingDouble(`<br>`  ToDoubleFunction f)` | Calculates summarizing statistics | `DoubleSummaryStatistics` |
| `summarizingInt(`<br>`  ToIntFunction f)` | Calculates summarizing statistics | `IntSummaryStatistics` |
| `summarizingLong(`<br>`  ToLongFunction f)` | Calculates summarizing statistics | `LongSummaryStatistics` |
| `summingDouble(`<br>`  ToDoubleFunction f)` | Calculates sum for doubles | `Double` |
| `summingInt(`<br>`  ToIntFunction f)` | Calculates sum for integers | `Integer` |
| `summingLong(`<br>`  ToLongFunction f)` | Calculates sum for longs | `Long` |

# Common grouping/partitioning collectors (5/6)

| Collector | Description | Return value (w/ collect) |
|---|---|---|
| `teeing(Collector c1,`<br>`Collector c2,`<br>`BiFunction f)` | Takes the results of to collectors to create a new type | `R` |
| `toList()` | Creates arbitrary type of list | `List` |
| `toSet()` | Creates arbitrary type of set | `Set` |
| `toCollection(`<br>`Supplier s)` | Creates `Collection` of specified type | `Collection` |

# Common grouping/partitioning collectors (6/6)

| Collector | Description | Return value (w/ collect) |
|---|---|---|
| `toMap(Function k,`<br>`  Function v)` | Creates map using functions to map keys and values | `Map` |
| `toMap(Function k,`<br>`  Function v,`<br>`  BinaryOperator m)` | Creates map using functions to map keys and values with merge rule | `Map` |
| `toMap(Function k,`<br>`  Function v,`<br>`  BinaryOperator m,`<br>`  Supplier s)` | Creates map using functions to map keys and values with merge rule and map type supplier | `Map` |

```
// joining() example

var names = Stream.of("John", "George", "Luke");

String result = names.collect(Collectors.joining("-"));

System.out.println(result);

   => John-George-Luke
```

static `Collectors` method is passed as an argument in `collect()`

```
// averaging() example

var names = Stream.of("John", "George", "Luke");

Double result = names.collect(Collectors.averagingInt(String::length));

System.out.println(result);

   => 4.666666666666667
```

argument is `ToIntFunction`

```
// toCollection() example
var names = Stream.of("John", "George", "Luke", "Joe");
TreeSet<String> result = names
  .filter(s -> s.startsWith("J"))
  .collect(Collectors.toCollection(TreeSet::new));
System.out.println(result);
  => [Joe, John]
```

argument is Supplier

```
// toMap() example #1
var names = Stream.of("John", "George", "Luke");
Map<String, Integer> result = names
  .collect(Collectors.toMap(s -> s, String::length));
System.out.println(result);
  => {George=6, Luke=4, John=4}
```

key

value

(both arguments are Function)

```
// toMap() example #2
var names = Stream.of("John", "George", "Luke");
Map<Integer, String> result = names
  .collect(Collectors.toMap(String::length, k -> k));
System.out.println(result);             key     value
  => Exception java.lang.IllegalStateException: Duplicate key 4

// to solve this we have to provide a merge rule, e.g.
Map<Integer, String> result = names.collect(Collectors.toMap(
  String::length,
  k -> k,
  (s1, s2) -> s1 + ";" + s2)); merge rule given by BinaryOperator
System.out.println(result);
  => {4=John;Luke, 6=George}
```

```
// if we don't specify the class, toMap can return any class
// which implements Map interface (usually HashMap, but not guaranteed)
// ...or we can specify the class:
Map<Integer, String> result = names.collect(Collectors.toMap(
  String::length,
  k -> k,
  (s1, s2) -> s1 + ";" + s2),
  TreeMap::new); specifying class by providing Supplier
System.out.println(result);
  => {4=John;Luke, 6=George}
System.out.println(result.getClass());
  => class java.util.TreeMap
```

```
// groupingBy() example #1

var names = Stream.of("John", "George", "Luke");

Map<Integer, List<String>> result = names

  .collect(Collectors.groupingBy(String::length));

System.out.println(result);

  => {4=[John, Luke], 6=[George]}
```

argument is Function

```
// groupingBy() example #2 (using downstream collector)

var names = Stream.of("John", "George", "Luke");

Map<Integer, Set<String>> result = names.collect(

  Collectors.groupingBy(

    String::length,

    Collectors.toSet()));

System.out.println(result);

  => {4=[Luke, John], 6=[George]}
```

*downstream collector* ensures that the value will be Set

```
// groupingBy() example #3 (using map supplier and downstream collector)

var names = Stream.of("John", "George", "Luke");

TreeMap<Integer, Set<String>> result = names.collect(

  Collectors.groupingBy(

    String::length,

    TreeMap::new,

    Collectors.toSet()));

System.out.println(result);

  => {4=[Luke, John], 6=[George]}
```

*map supplier* ensures that the `Map` implementation will be `TreeMap`

*downstream collector* ensures that the value will be `Set`

```
// partitioningBy() has only two groups: true and false

var names = Stream.of("John", "George", "Luke");

Map<Boolean, List<String>> result = names.collect(
  Collectors.partitioningBy(s -> s.length() <= 4));

System.out.println(result);

  => {false=[George], true=[John, Luke]}
```

argument is Predicate

```
// partitioningBy() with Set instead of List
var names = Stream.of("John", "George", "Luke");
Map<Boolean, Set<String>> result = names.collect(
  Collectors.partitioningBy(
    s -> s.length() <= 4,
    Collectors.toSet()));
System.out.println(result);
  => {false=[George], true=[John, Luke]}
```

we have provided *downstream collector*

```
// teeing() is used for returning multiple values, e.g. sum and average

// step 1: create a type which stores values:

record MyData(int sum, double avg) {}

// step 2: use stream to return the result of the type MyData

var numbers = Stream.of(1, 2, 3, 4, 5);

MyData result = numbers.collect(

  Collectors.teeing(

    Collectors.summingInt(i -> i),      first collector

    Collectors.averagingDouble(i -> i),     second collector

    MyData::new     merging function

));

System.out.println("Sum: " + result.sum() + ", Average: " + result.avg());

  => Sum: 15, Average: 3.0
```