Lesson 3-4:
Using Collectors

Collector Basics

- A `Collector` performs a mutable reduction on a stream
  - Accumulates input elements into a mutable result container
  - Results container can be a `List`, `Map`, `String`, etc
- Use the `collect()` method to terminate the stream
- `Collectors` utility class has many methods that can create a `Collector`

Composing Collectors

- Several `Collectors` methods have versions with a downstream collector
- Allows a second collector to be used
  - `collectingAndThen()`
  - `groupingBy()/groupingByConcurrent()`
  - `mapping()`
  - `partitioningBy()`

Collecting Into A Collection

- `toCollection(Supplier factory)`
  - Adds the elements of the stream to a `Collection` (created using factory)
  - Uses encounter order
- `toList()`
  - Adds the elements of the stream to a `List`
- `toSet()`
  - Adds the elements of the stream to a `Set`
  - Eliminates duplicates

Collecting To A Map

- `toMap(Function keyMapper, Function valueMapper)`
  - Creates a `Map` from the elements of the stream
  - key and value produced using provided functions
  - Use `Functions.identity()` to get the stream element

```
Map<Student, Double> studentToScore = students.stream()
  .collect(toMap(Functions.identity(),
                 student -> getScore(student)));
```

Collecting To Map

Handling Duplicate Keys

```
toMap(Function keyMapper, Function valueMapper,
  BinaryOperator merge)
```

- The same process as first `toMap()` method
  - But uses the `BinaryOperator` to merge values for duplicate keys

```
Map<String, String> occupants = people.stream()
  .collect(toMap(Person::getAddress,
                 Person::getName,
                 (x, y) -> x + "," + y));
```

People at the same address are merged into a CSV string

Grouping Results

- groupingBy(Function)
  - Groups stream elements using the Function into a Map
  - Result is Map<K, List<V>>

```
Map m = words.stream()
  .collect(Collectors.groupingBy(String::length));
```

- groupingBy(Function, Collector)
  - Groups stream elements using the Function
  - A reduction is performed on each group using the downstream Collector

```
Map m = words.stream()
  .collect(Collectors.groupingBy(String::length, counting()));
```

|

Joining String Results

- `joining()`
  - Collector concatenates input strings
- `joining(delimiter)`
  - Collector concatenates stream strings using CharSequence delimiter

```
collect(Collectors.joining(",")); // Create CSV
```

- `joining(delimiter, prefix, suffix)`
  - Collector concatenates the prefix, stream strings separated by delimiter and suffix

Numeric Collectors

Also Available In Double And Long Forms

- `averagingInt(ToIntFunction)`
  - Averages the results generated by the supplied function
- `summarizingInt(ToIntFunction)`
  - Summarises (count, sum, min, max, average) results generated by supplied function
- `summingInt(ToIntFunction)`
  - equivalent to a `map()` then `sum()`
- `maxBy(Comparator), minBy(Comparator)`
  - Maximum or minimum value based on `Comparator`

Other Collectors

- reducing(BinaryOperator)
  - Equivalent Collector to reduce() terminal operation
  - Only use for multi-level reductions, or downstream collectors
- partitioningBy(Predicate)
  - Creates a Map<Boolean, List> containing two groups based on Predicate
- mapping(Function, Collector)
  - Adapts a Collector to accept different type elements mapped by the Function

```
Map<City, Set<String>> lastNamesByCity = people.stream()
  .collect(groupingBy(Person::getCity,
                    mapping(Person::getLastName, toSet())));
```

Section 4

Summary

- Collectors provide powerful ways to gather elements of an input stream
  - Into collections
  - In numerical ways like totals and averages
- Collectors can be composed to build more complex ones
- You can also create your own `Collector`

ORACLE®