15

Terminal Operations: Collectors

ORACLE

Objectives

After completing this lesson, you should be able to use Collectors to:

- Create a collection
- Group elements into a collection
- Perform summarizing on a collection
- Create a custom collector in code
- Create a custom collector by combining collectors

Agenda

- Introduction to collectors
- Three argument `collect` method of `Stream`
- Single argument `collect` method of `Stream`
- Grouping by collectors
- Nested values
- Complex custom collectors

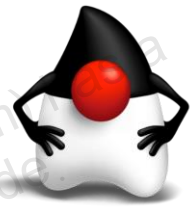
Streams and Collectors Versus Imperative Code

Functional style code has advantages over imperative coding:

- Concise
  - Defines aggregate operations
- Readable
- Flexible and extensible
  - Composition possible
- Parallel ready
- Dealing with data in the aggregate
  - Collectors are important here

Collection

- Reduction is the most important kind of terminal operation.
  - Combines a sequence or collection of an arbitrary value into one value.
  - `forEach()` is useful but should generally not be used to create results.
- Reduction uses either:
  - The reduce method of `Stream`: used with immutable types
  - The collect method of `Stream`: used with mutable types
- There are many predefined collectors that can be used with the `collect` method to create complex queries that generate a collection.
- Custom collectors can be created by:
  - Providing functions (often using lambda):
    - To the three argument `Stream.collect()` method
    - To the `Collectors.of()` method
  - By implementing the `Collector` interface directly



Reduction combines a sequence or collection of an arbitrary value into one value. A collector facilitates a particular kind of reduction, which is one that reduces stream elements into a container (such as an array or Collection) using mutation. Collectors provide mutable reduction.

Immutable reduction using the reduce method of Stream is covered in the lesson “Parallel Streams.”

Predefined Collectors

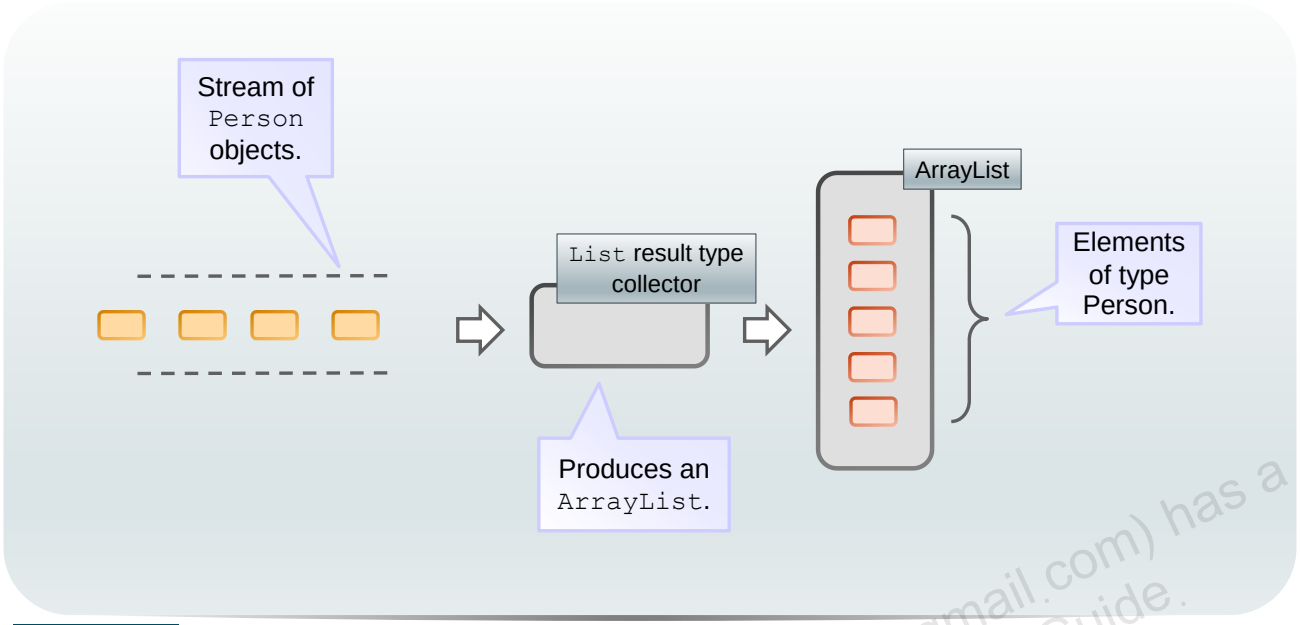
- Most examples in this lesson use a very simple Person class that contains

```
public class Person {
    public enum City {
        Belfast,
        Tulsa,
        Athens,
        London; }
    private City city;
    private String firstName;
    private String lastName;
    private int age;

    // Not shown: A constructor that populates all fields,
    // and getter and setter methods.
```

The data used in most of the examples can be created like this:

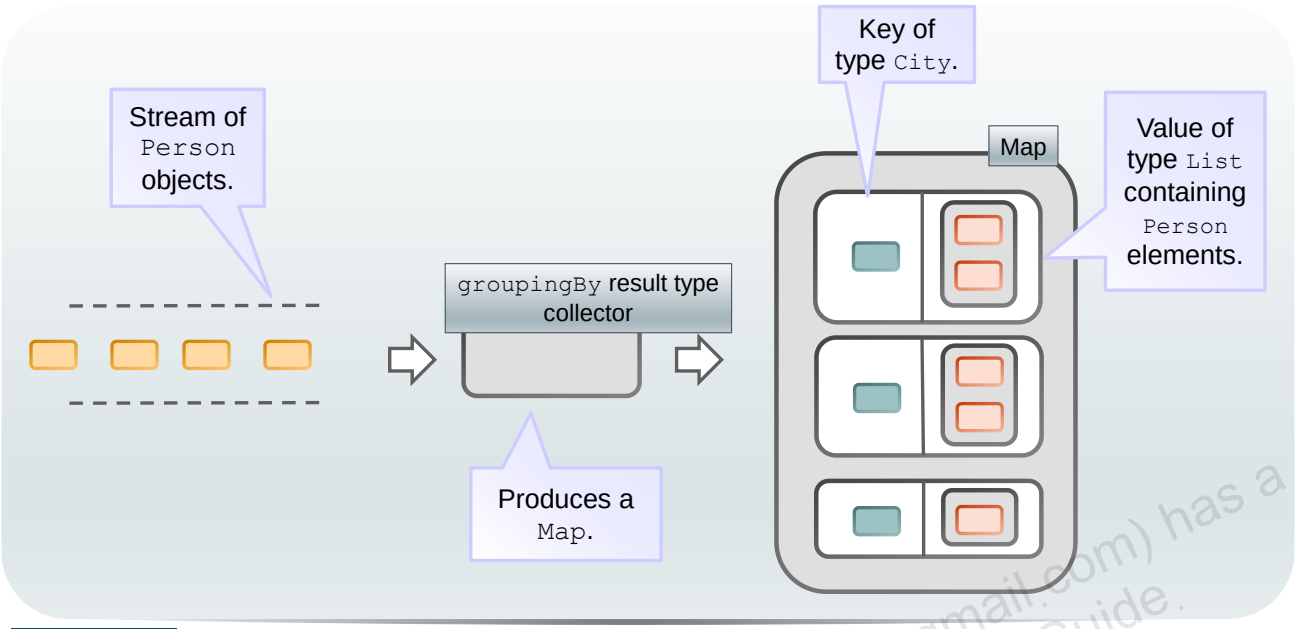
```
static List<Person> people =
    List.of(new Person(City.Tulsa, "Joe", "Bloggs", 42),
            new Person(City.Athens, "Amy", "Laverda", 21),
            new Person(City.London, "Bill", "Gordon", 33),
            new Person(City.Athens, "Eric", "Vincent", 33),
            new Person(City.Tulsa, "Eric", "Dunmore", 29));
```

A Simple Collector



The diagram shows the basic operation of a simple collector.

A More Complex Collector



The diagram shows a more complex manipulation of the stream of data. The data elements are classified by a particular data item, in this case by the city the person lives in. The groupingBy collector is the easiest way to do this, but in later slides, you see that you can use the toMap collector or even write your own custom collector to achieve similar results.

There are a number of ways to achieve this type of processing and a number of approaches are covered later in the lesson.

Agenda

- Introduction to collectors
- Three argument `collect` method of `Stream`
- Single argument `collect` method of `Stream`
- Grouping by collectors
- Nested values
- Complex custom collectors

The Three Argument `collect` Method of `Stream`

```
<R> R  collect(Supplier<R> supplier,
BiConsumer<R,? super T> accumulator,
BiConsumer<R,R> combiner)
```

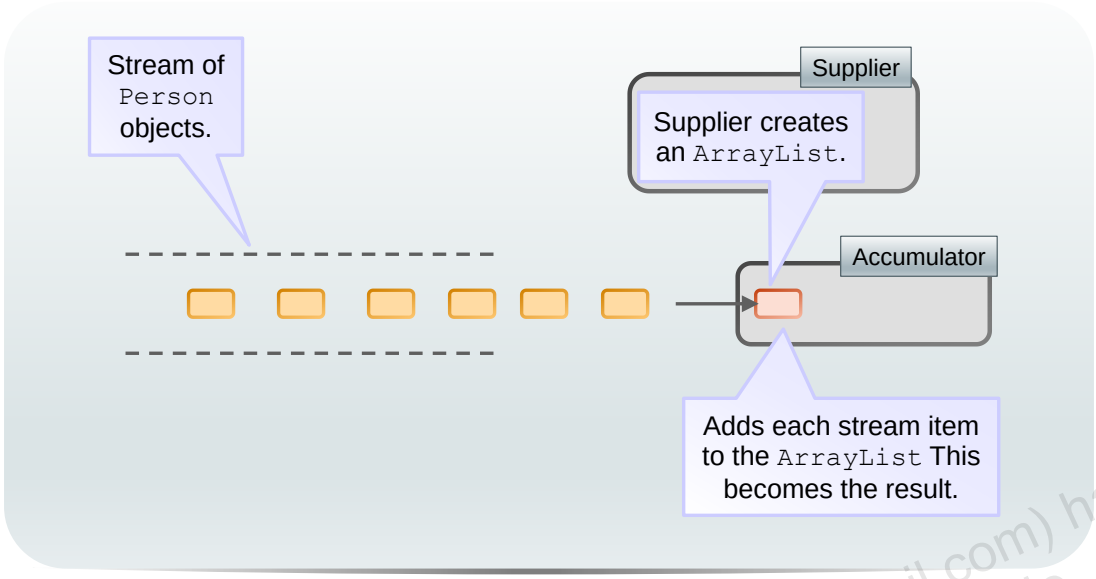
- `supplier` - a Supplier that creates a new mutable result container (or containers)
- `accumulator` - a BiConsumer that must fold an element into a result container
- `combiner` - a BiConsumer that merges two partial result containers

Types:

- R – The result type
- T – The type of the stream elements

Result container in this context is whatever type has been chosen to be created to store the accumulating data. Typically it will be a collection, but it could be a simple primitive if, for example, all that is required is a simple count.

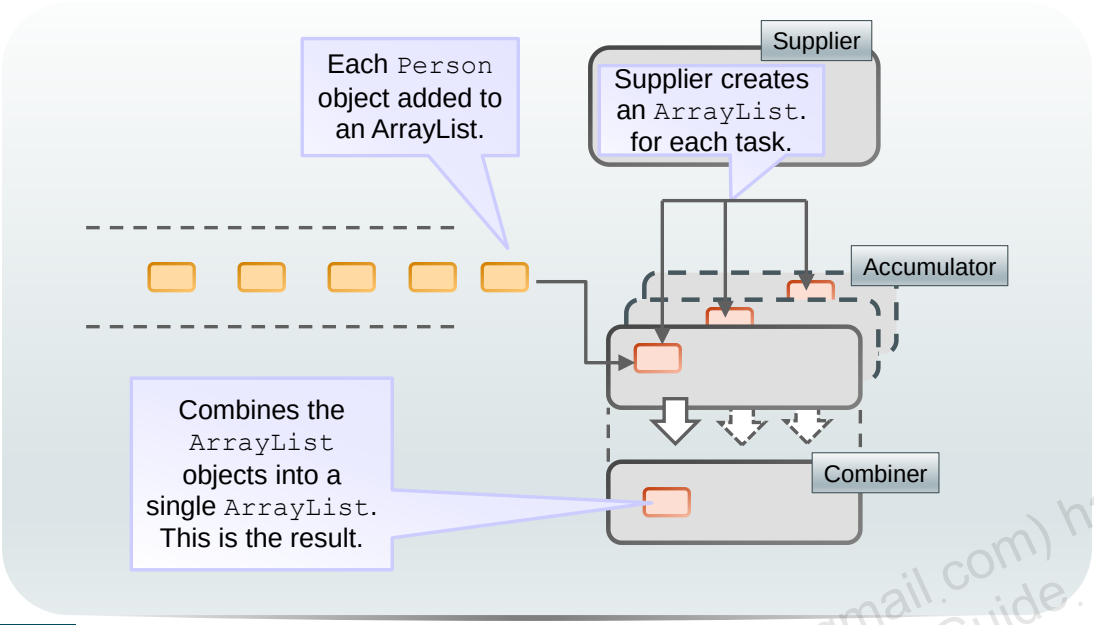
The `collect` Method Used with a Sequential Stream



In the operation of the collector, the Supplier creates the result container, here an ArrayList. As the Stream is processed, the Accumulator modifies this result container.

If the Stream is sequential, that's all that happens. BUT you should not assume that your collector will only be used with sequential Streams—you must ensure that parallel Streams are addressed also, and this is covered next.

The `collect` Method Used with a Parallel Stream



Note the addition of the combiner when the collect method operates on a parallel stream. A collector is a reusable object, and it may end up being used in different ways or under different circumstances - particularly in parallel - than the author of the collector had intended.

The `collect` Method: Collect to an `ArrayList` Example

Assuming a Stream of Person elements:

```
List<Person> myPpl = people.stream()
    .collect( ArrayList::new ,

    ArrayList::add,

    ArrayList::addAll );

    System.out.println(myPpl);

[Joe, Amy, Bill, Eric, Eric]
```

Supplier: creates an `ArrayList` as the new result type.

Accumulator: adds `Person` elements to the result `ArrayList`.

Combiner: if parallel stream, takes two `ArrayList` objects and combines them.

The slide shows an example of the three argument collect method. It is extremely simple and essentially does exactly what one of the predefined collectors does (the toList one). Nevertheless, it illustrates how you specify the supplier, accumulator, and combiner yourself, by using lambda (here method references) inline in the code.

Agenda

- Introduction to collectors
- Three argument `collect` method of `Stream`
- Single argument `collect` method of `Stream`
- Grouping by collectors
- Nested values
- Complex custom collectors

The Single Argument `collect` Method of `Stream`

```
collect(Collector<? super T,A,R> collector)
```

- `Collector` - a Collector object that uses the types shown and transforms the stream into a single object (often but not necessarily a collection)
  - Many predefined Collectors available in the Collectors class
    - For example, `Collectors.toList()` is a static factory method that creates a collector that reduces the stream to an `ArrayList`.
  - The types are:
    - `? super T` is the type of the input elements to the Collector.
    - `A` is the mutable accumulation type of the reduction operation (often hidden as an implementation detail).
    - `R` is the result type of the reduction operation.

? super T means that the code within the Collector will operate on elements of the stream or any superclass of elements of the stream. For example, with a steam of Person elements, the Collector can work with Person or a superclass of Person. For example, a Collector that returns an ArrayList<Object> elements would work with Stream<Person>.

Note that when using a particular Collector class in this way, the accumulation type is not necessarily the same as the result type. The reason for this will be covered in more details later in this lesson.

Using Predefined Collectors From the `Collectors` Class

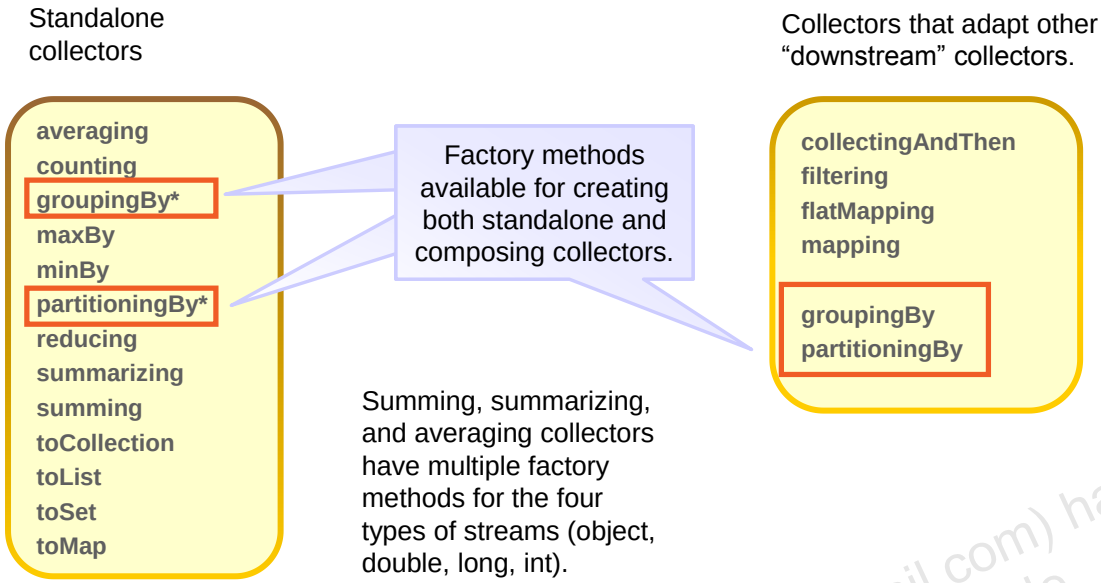
Implementations of `Collector` that implement various useful reduction operations.

- Available from static factory methods of the `Collectors` class.

Can be used in two ways:

- Standalone
  - Accumulate to collections
    - Some factory methods allow the collection type to be chosen, e.g. `HashSet` or `TreeSet`
  - Accumulate to a Map using a classifier function to determine the keys
- Composing
  - Collectors can be used to adapt the functionality of another Collector:
    - For example, the filtering Collector can be used to adapt another “downstream” Collector
  - Composing useful for collectors that partition or group elements to do further processing “downstream”

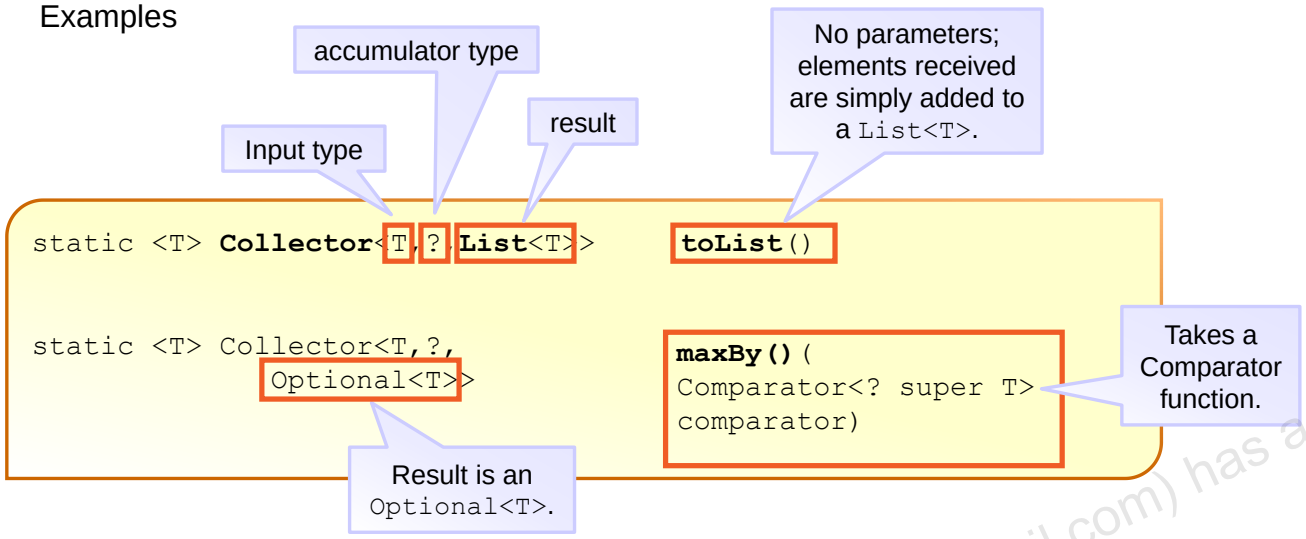
List of Predefined `Collectors`



Note that some of these collectors are especially useful in combination with grouping collectors. Otherwise, one could as easily use Stream.count(), Stream.map(), and Stream.filter as counting, mapping, and filtering.

Stand-Alone Collectors

Examples



When you look at a factory method that returns a Collector, remember it's the third parameter that is the result type.

In the toList() example, the List is of the same type, T, as the input elements. In the maxBy() example, the result is an Optional of type T.

Stand-Alone `Collector`: List all Elements

```
static List<Person> people =
   List.of(new Person(City.Tulsa, "Joe", "Bloggs", 42),
           new Person(City.Athens, "Amy", "Laverda", 21),
           new Person(City.London, "Bill", "Gordon", 33),
           new Person(City.Athens, "Eric", "Vincent", 33),
           new Person(City.Tulsa, "Eric", "Dunmore", 29));
```

Creates `List` of `Person` objects

```
List<Person> myPpl = people.stream()
   .collect(toList());

   System.out.println(myPpl);
```

Using static import of `Collectors.toList()` for readability.

```
[Joe, Amy, Bill, Eric, Eric]
```

Uses `toString` method of Person for output.


```
public class Person {
    public String toString() {
        return firstName;
    }
    public Person() {}
    public Person(City city, String name, int age) {
        this.city = city;
        private String firstName;
        private String lastName;
        this.age = age;
    }
    public enum City {
        Belfast,
        Tulsa,
        Athens,
        London;
    }
    private City city;
    private String name;
    private int age;

//… (lines omitted - setters and getters available for all fields)…
}
```

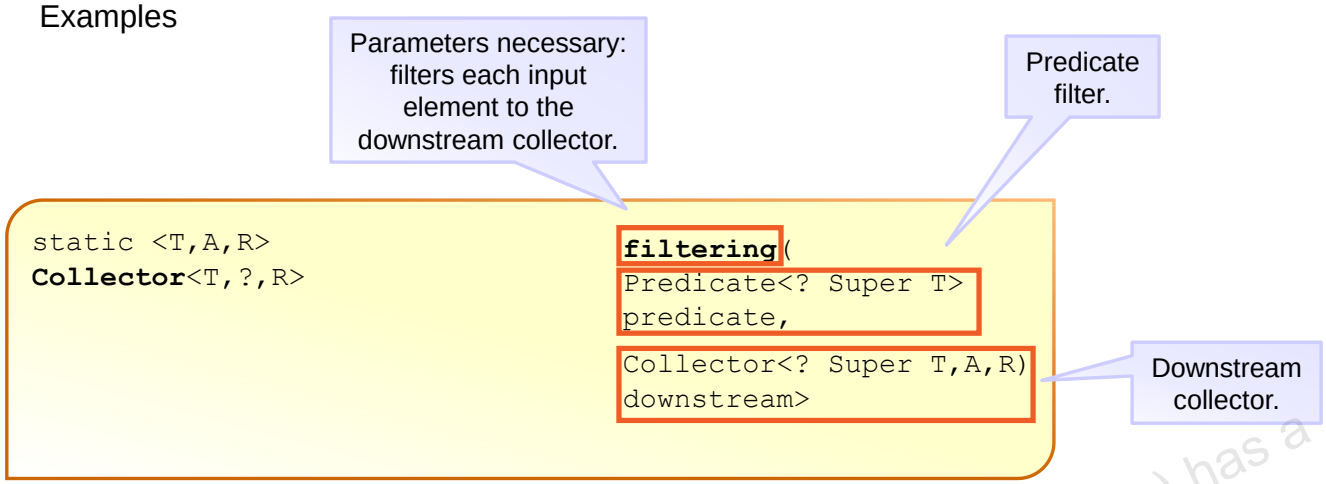
maxBy() Example

```
people.stream()
  .collect(maxBy(Comparator.comparing(Person::getAge)))
  .ifPresentOrElse(m -> System.out.println(m +
      ", " + m.getAge() + ", is oldest"),
      () -> System.out.println("No Person of max age found"));
```

```
Joe, 42, is oldest
```

Adapting Collector: `filtering()`

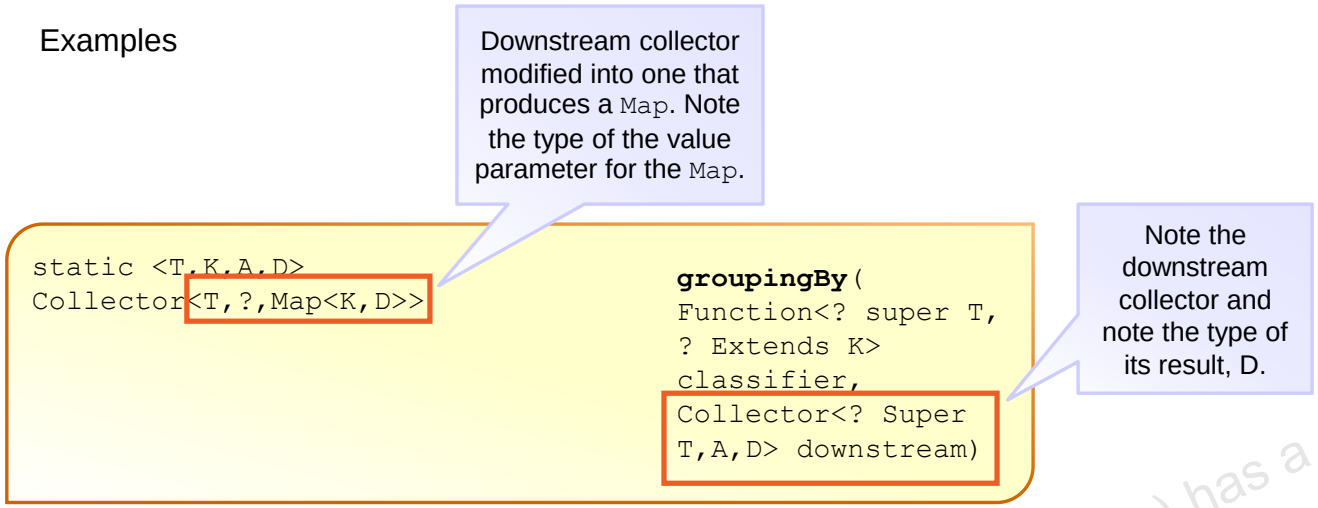
Examples



The collector being adapted is referred to as the “downstream” collector.

Composing Collectors : `groupingBy()`

Examples



When you look at a factory method that returns a Collector, remember it's the third parameter, the result, that indicates what the Collector will produce.

Composing Collectors: Using Mapping

`mapping()` adapts the downstream collector.

```
List<String> myPpl = people.stream()
    .collect(mapping(Person::getName , toList()));

System.out.println(myPpl);
[Joe Bloggs, Amy Laverda, Bill Gordon, Eric Vincent, Eric
    Dunmore]
```

Mapping `Person` to a `String`.

Adding each element to a `List`.

Result is a `List` of type `String`.

```
List<String> PplCities = people.stream()
   .collect(mapping(p -> p.getCity() + ":" + p.getName(),
    toList()));

System.out.println(PplCities);
[Tulsa:Joe Bloggs, Athens:Amy Laverda, London:Bill Gordon,
    Athens:Eric Vincent, Tulsa:Eric Dunmore]
```

Note how some cities are listed multiple times.

An example of using mapping with another downstream collector. The joining() collector adds each element to a String and separates each with a delimiter.

```
String stringOfPpl = people.stream()
    .collect(mapping(Person::getName ,joining("/")));

System.out.println(stringOfPpl);
Joe Bloggs/Amy Laverda/Bill Gordon/Eric Vincent/Eric Dunmore
```

`toMap()` and Duplicate Keys

```
Map<City, String> pplCitiesMap = people.stream()
    .collect(toMap(c -> c.getCity(), p -> p.getName(),
    (a, b) -> a + ":" + b));

System.out.println(pplCitiesMap);
{Tulsa=Joe Bloggs:Eric Dunmore, Athens=Amy Laverda:Eric Vincent,
    London=Bill Gordon}
```

toMap takes two functions for accumulating keys and values.

toMap can also take a third parameter (BinaryOperator) to handle duplicate keys.

Result is a Map of String objects, where each String lists all the people in a particular city.

The result is useful, but a `List` value would be better.

How can this be achieved?

- By writing a custom collector as shown earlier
- By using the `groupingBy()` collector

If you do not create a merging BinaryOperator, and there is a duplicate key, an IllegalStateException will be thrown.

This is a very common need, and while it is good to know about how to use the toMap collector and resolve the duplicates, there's another approach that is superior in most cases, using groupingBy().

Agenda

- Introduction to collectors
- Three argument `collect` method of `Stream`
- Single argument `collect` method of `Stream`
- Grouping by collectors
- Nested values
- Complex custom collectors

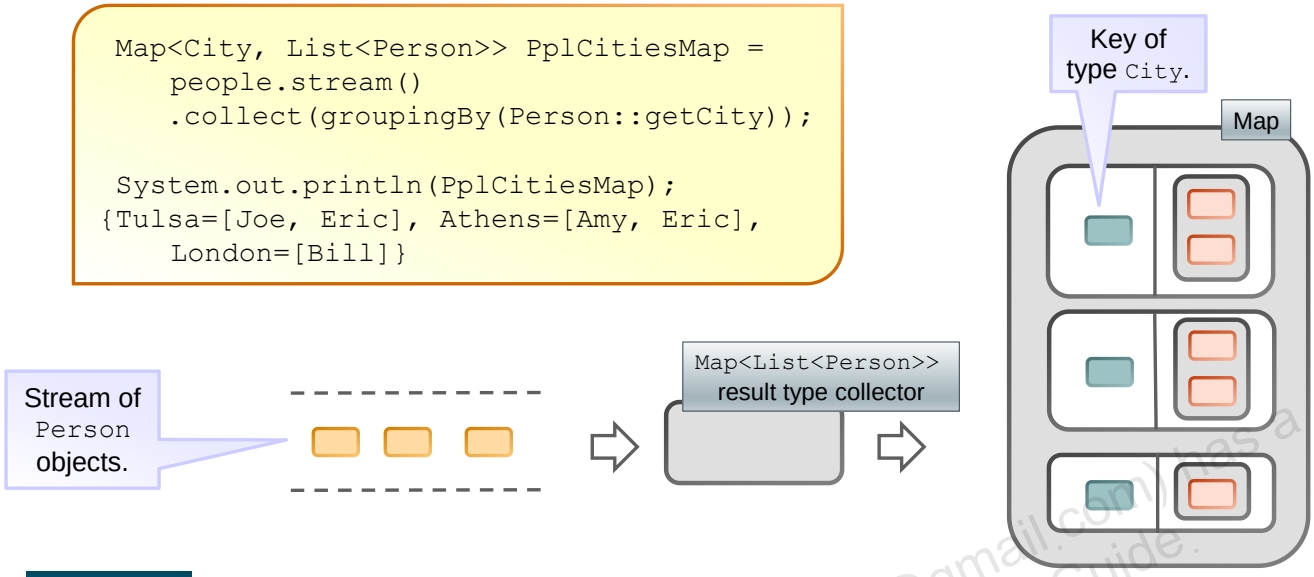
`groupingBy` and `partitioningBy` Collectors

Both `groupingBy` and `partitioningBy` create a `Map`

- `partitioningBy`
  - Creates a map with two keys, true and false
  - The standalone version creates a value (type List) for each key
  - Uses a predicate to determine whether the element should go in the true or false List
- `groupingBy`
  - Uses a Function to create a set of keys for the Map
  - The standalone version creates a value (type List) for each key
  - Based on the function, elements with a particular key are added to the value for that key
- `partitioningBy` and `groupingBy` can be standalone or can use a downstream collector.
  - The standalone collector collects each group of values into a `List`.

Stand-Alone `groupingBy`: `Person` Elements By `City`

```
Map<City, List<Person>> PplCitiesMap =
    people.stream()
    .collect(groupingBy(Person::getCity));

System.out.println(PplCitiesMap);
{Tulsa=[Joe, Eric], Athens=[Amy, Eric],
    London=[Bill]}
```



This is a stand-alone `groupingBy` collector, so by default its result type is Map<City, List<Person>>.

Stream Operations Or Equivalent Collectors?

Many stream methods are available as collectors. For example:

- `Stream.map()` and `Collectors.mapping()`
- `Stream.filter()` and `Collectors.filtering()`

```
people.stream()
   .map(p -> p.getName() + ":" + p.getAge())
   .collect(toList())
   .forEach((v) -> System.out.println(v));
```

```
people.stream()
   .collect(mapping(p -> p.getName() + ":" +
        p.getAge(), toList()))
   .forEach((v) -> System.out.println(v));
```

The code fragments produce the same output.

Stream Operations Or Equivalent Collectors with `groupingBy`

Using Stream.map() or Collectors.mapping() with groupingBy.

```
Map<City,List<String>> peopleByCity =
   people.stream()
  .collect(groupingBy(Person::getCity, mapping(p -> p.getName() +
   ":" + p.getAge(), toList())));
System.out.println(peopleByCity);

{Tulsa=[Joe Harley:42, Eric Greeves:29], London=[Bill Honda:34],
    Athens=[Amy Beemer:21, Eric Vincent:33]}
```

This works because the mapping takes place downstream of the groupingBy.

```
people.stream()
    .map(p -> p.getName() + ":" + p.getAge())
    .collect(groupingBy(Person::getCity));
```

This cannot work because the classifier cannot access City to perform grouping.

Stream.count(), Collectors.counting and groupingBy

groupingBy affects what is summarized.

```
Long numPpl = people.stream()
                .count();
System.out.println(numPpl);

5
```

Number of elements in the entire stream.

```
Map<City, Long> numPplByCity =
people.stream()
  .collect(groupingBy(Person::getCity, counting()));
    System.out.println(numPplByCity);

{Tulsa=2, London=1, Athens=2}
```

Number of elements in each group.

Composing Collectors: Tallest in Each City

Collectors have the equivalent of many `Stream` methods but operating on groups.

```
Map<City, Optional<Person>> oldestByCity = people.stream()
   .collect(groupingBy(Person::getCity,
    maxBy(comparing(Person::getAge))));
```

The `forEach` method is useful for printing.

```
 oldestByCity.forEach((k, v) ->
   { System.out.print(k + ":");
     System.out.println(v.isPresent()?
     v.get().getFirstName() + " age " + v.get().getAge()
     :"No oldest person!");});
```

Print the key.

Print the value, but as `v` is an `Optional`, you can use the `isPresent` method in a ternary expression to determine what to print. See notes.

A more elegant way to deal with the `Optional` is to use the `map` and `orElse` methods of `Stream`.

```
oldestByCity.forEach((k, v) ->
         { System.out.print(k + ":");
         System.out.println(
           v.map(person -> person.getFirstName() + " age " + person.getAge())
           .orElse("No oldest person!"));});
```

groupingBy: Additional Processing with entrySet()

The code below produces a Map of cities showing the population of each.
But what if a list of cities with population > than 1 is what is required?

```
Map<City,Long> populousCities = people.stream()
   .collect(groupingBy(Person::getCity, counting()));
```

Use entrySet method to create a new Stream.

```
Set<City> populousCities = people.stream()
   .collect(groupingBy(Person::getCity, counting()))
   .entrySet().stream()   //Set<Entry<City, Long>>
   .filter(e -> e.getValue() > 1)
   .map(Map.Entry::getKey)
   .collect(toSet());
```

This line creates a new Stream of type shown in the comment. Note that this is also a new pipeline.

It may seem that it is possible to use a Collector that adapts the first example above instead.

In looking in the documentation, the only candidate is collectingAndThen.

To use collectingAndThen, you could do the following, but the example above is more succinct and readable. Note that either way, there are two separate stream pipelines being used, so the compiler will not be able to combine them for optimization purposes.

```
Map<City,Long> populousCities = people.stream().parallel()
   .collect(collectingAndThen(groupingBy(Person::getCity, counting()), f -> {
   // Need to iterate through map to search for cities with > 1 pop
   // Will need to use entrySet to do this just as in the other version!
   f.entrySet().removeIf(d -> d.getValue() < 2);
   return f;
}));
```

Agenda

- Introduction to collectors
- Three argument `collect` method of `Stream`
- Single argument `collect` method of `Stream`
- Grouping by collectors
- Nested values
- Complex custom collectors

Nested Values

The remaining examples use a more complex type, `ComplexSalesTxn`.

- More fields than `Person`.
- One of the fields is a `List` type.

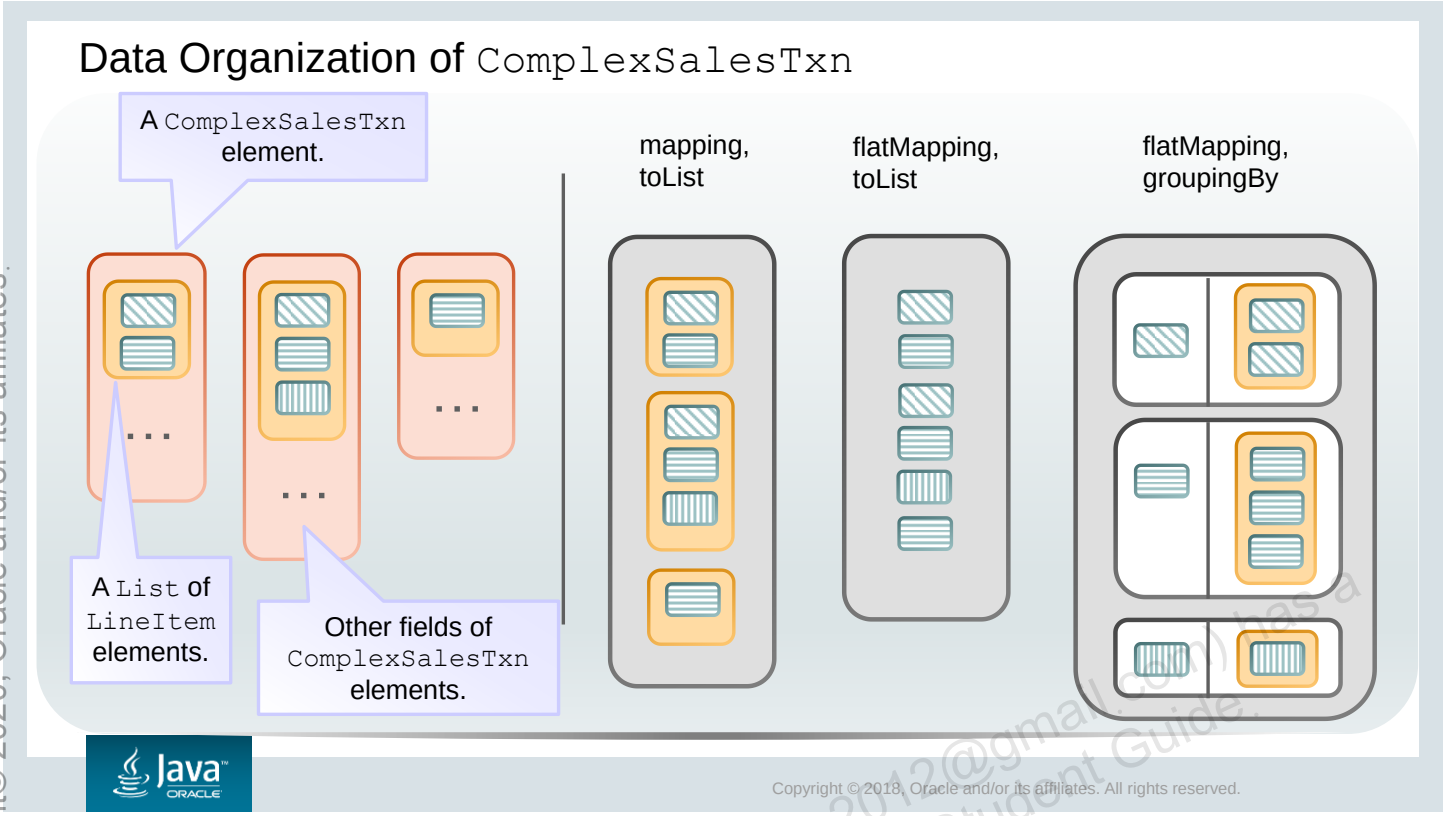
Fields of `ComplexSalesTxn` class

```
private long txnId;
private String salesPerson;
private Buyer buyer;
private List<LineItem> lineItems;
private LocalDate txnDate;
private String city;
private State state;
private String code;
```

Fields of `LineItem` class

```
private String name;
private int quantity;
private int unitPrice;;
```

A `ComplexSalesTxn` may have one or many `LineItem` elements.



Note how flatMapping ensures the individual Lists that group the LineItem elements in each transaction do not exist in the output in the flatMapping/toList collector. In the flatMapping/groupingBy collector, the individual LineItem elements are once more grouped in a List, but this time they're grouped by whatever is the groupingBy classifer, and not by transaction.

Displaying Nested Values: Listing Line Items in Each Transaction

Use `mapping` with `toList` to list `LineItem` elements for each transaction.

```
List<List<LineItem>> lineItemsInTransaction = tList.stream()
   .collect(mapping(ComplexSalesTxn::getLineItems, toList()));

System.out.println(lineItemsInTransaction);
```

`LineItem` element.

Result is a `List` of `LineItem` elements within a `List`.

```
[[Widget, Widget Pro II], [Widget Pro], [Widget Pro II, Widget], ... ]
```

`List` of `LineItem` elements.

`List` of `List` of `LineItem` elements.

This output may be useful, but what if you want to organize the data by product type. You cannot do a groupingBy as each transaction's LineItem elements are inside a List. If you want to organize by product type and not by transaction, you will need to get rid of the List of LineItems for each transaction, and instead have all the LineItems for all transactions in the same List.

Displaying Nested Values: Listing Line items

Use `flatMapping` with `toList` to list all the `LineItem` elements.

```
List<LineItem> lineItems = tList.stream()
   .collect(flatMapping(t -> t.getLineItems().stream(), toList()));

System.out.println(lineItems);
```

`LineItem` element.

Result is a `List` of `LineItem` elements. This can be grouped by product type.

```
[Widget, Widget Pro II, Widget Pro, Widget Pro II, Widget, ... ]
```

`List` of `LineItem` elements.

Now the List just contains each LineItem element without any reference to the transaction it came from.

Notice that in the code above or the code on the previous slide, instead of using the mapping or flatMapping adaptor collectors, you could use map or flatMap operations on the stream. For example, the following code would generate the same List as the code in the slide.

```
List<LineItem> lineItems = tList.stream()
   .flatMap(t -> t.getLineItems().stream())
   .collect(toList());
System.out.println(lineItems);
```

Displaying Nested Values: Grouping `LineItem` elements

Use `mapping` with `toList` to list `LineItem` elements.

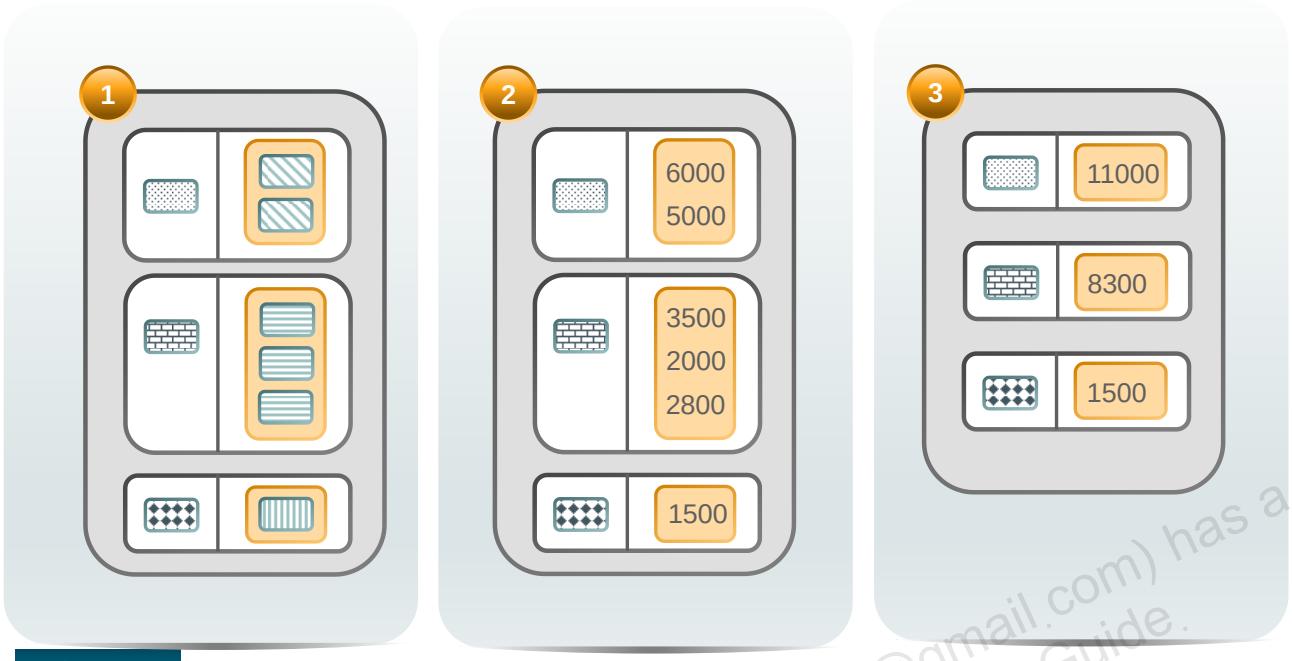
```
Map<String,List<Double>> valueOfEachLineItem = tList.stream()
   .collect(flatMapping(t -> t.getLineItems().stream(),
    groupingBy(LineItem::getName,
    mapping(o -> o.getQuantity() * o.getUnitPrice(),
    toList()))));

System.out.println(valueOfEachLineItem);
```

Result is a `Map` organized by product types. The amount for product type is shown.

```
{Widget=[6000.0, 6000.0, 36000.0, 10200.0, 16500.0, 6000.0, 11100.0],
    Widget Pro=[20000.0, ...}
```

LineItem elements now grouped by of product type of the LineItem element. This allow you to find information like how many of each product have been ordered and which product type generates the most revenue.

groupingBy Examples for ComplexSalesTxn



1. The first example groups the LineItem elements by the type of product.

```
Map<String,List<LineItem>> groupLineItemsByItemName = tList.stream()
    .collect(flatMapping(t -> t.getLineItems().stream(),
     groupingBy(LineItem::getName, toList())));
System.out.println(groupLineItemsByItemName);
{Widget=[Widget, Widget, ...], Widget Pro=[Widget Pro, ...], Widget Pro II=[Widget Pro
II, Widget Pro II, ...]}
```

2. The second uses mapping to map from the LineItem type to a Double that displays the value for each product type.

```
Map<String,List<Double>> valueOfEachLineItem = tList.stream()
   .collect(flatMapping(t -> t.getLineItems().stream(),
    groupingBy(LineItem::getName,
    mapping(o -> o.getQuantity() * o.getUnitPrice(), toList()))));
System.out.println(valueOfEachLineItem);
{Widget=[6000.0, 6000.0, 36000.0, ...], Widget Pro=[20000.0, ...], Widget Pro
II=[45000.0, 52500.0, ...]}
```

3. The third sums the individual LineItem elements for each product.

```
Map<String,Double> valueOfSalesByProduct2 =  tList.stream()
   .collect(flatMapping(t -> t.getLineItems().stream(),
    groupingBy(LineItem::getName,
    summingDouble(o -> o.getUnitPrice() * o.getQuantity()))));
System.out.println(valueOfSalesByProduct2);
{Widget=91800.0, Widget Pro=69500.0, Widget Pro II=435000.0}
```

Group Items by Salesperson

Use `flatMapping` and `summingDouble` to total sales for each salesperson.

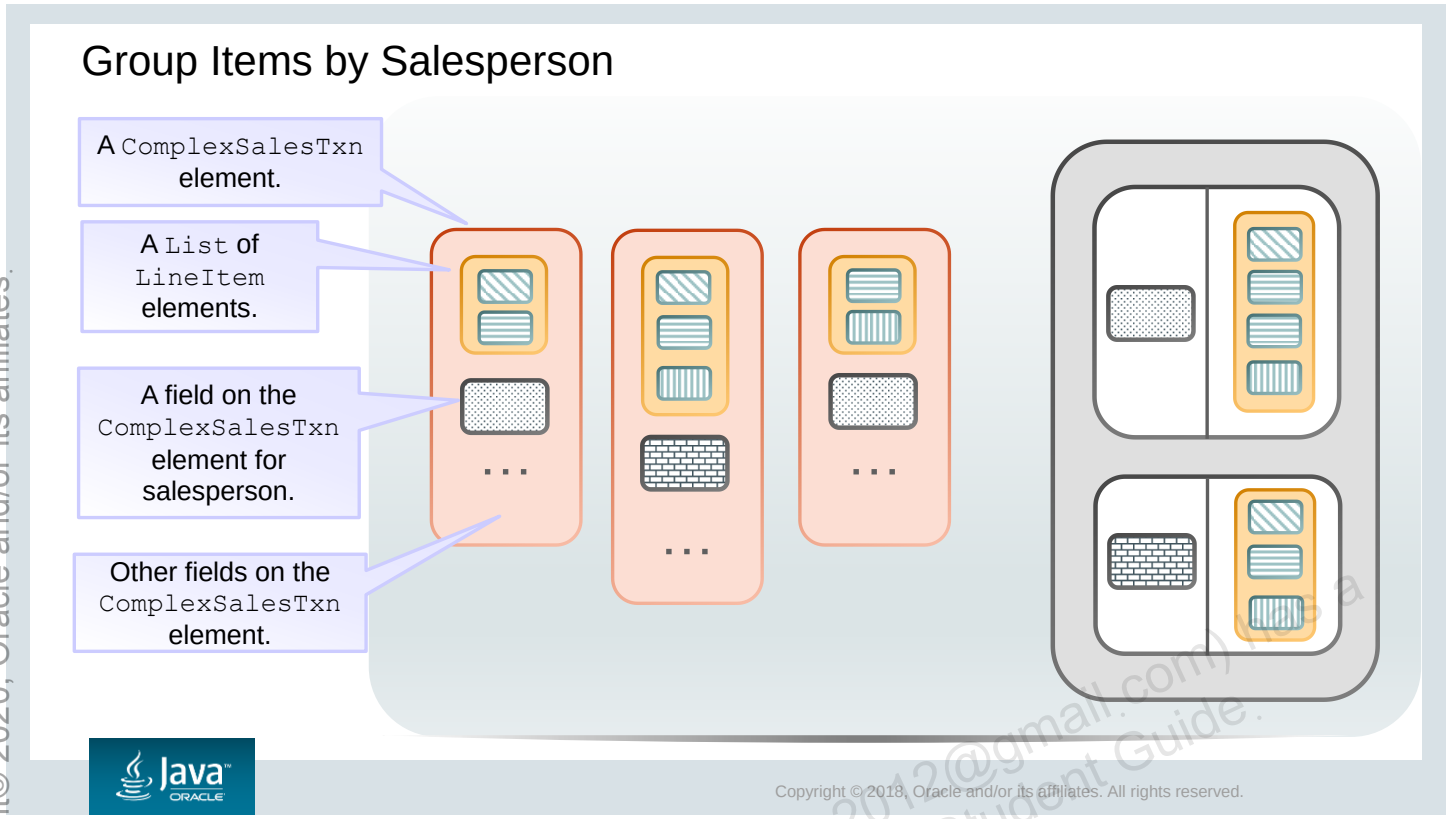
```
Map<String, Double> salesPerSalesPerson = tList.stream()
     .collect(groupingBy(ComplexSalesTxn::getSalesPerson,
     flatMapping(t -> t.getLineItems().stream(),
     summingDouble(o -> o.getQuantity() * o.getUnitPrice()))));
     System.out.println(salesPerSalesPerson);
{Samuel Adams=87600, John Smith=116000, Rob Doe=58500 ...
```

It's a `Map` so no ordering.

```
salesPerSalesPerson.entrySet().stream()
   .sorted(comparing(Entry::getValue))
   .forEach(System.out::println);
```

For a reversed sort, use reversed. Note that you now cannot use method reference and the type is required also.

```
salesPerSalesPerson.entrySet().stream()
   .sorted(Comparator.comparing((Entry<String, Double> a) ->
       a.getValue()).reversed())
   .forEach(System.out::println);
```



In the result, the Map value entry for a salesperson is a list of LineItem elements for an individual product. You could now do a further groupingBy to further group by a field on the LineItem elements. Or if you wanted to nest groupingBy collectors to group by, say, city, salesperson, buyer, you'd set up the collector to do the flatMapping after the groupingBy operations (so that the fields of ComplexSalesTxn would be available to group by). You can see an example of multiple groupingBy collectors being used in the examples for this lesson.

Agenda

- Introduction to collectors
- Three argument `collect` method of `Stream`
- Single argument `collect` method of `Stream`
- Grouping by collectors
- Nested values
- Complex custom collectors

Complex Custom Collectors

To create more complex custom collectors:

- Create a new custom collector with the functionality required.
  - Sometimes necessary, but can be complex to code.
- Combine the predefined collectors in order to achieve the functionality required.
  - Usually the best approach as the predefined collectors are designed to be combined.

The `collect` Method: Using a Custom Collector

```
collect(Collector<? super T,A,R> collector)
```

- The other `collect` method takes a `Collector` as a parameter.
  - Many predefined collectors available in the `Collectors` class

```
List<Person> myPpl = people.stream()
    .collect(new MyCustomCollector());

    System.out.println(myPpl);

[Joe, Amy, Bill, Eric, Eric]
```

Uses a custom collector to create an `ArrayList` Of `Person` elements. Also possible to use a predefined collector.

The code example in the slide shows the use of a custom Collector to reduce the stream of Person elements to an ArrayList. There is a predefined collector available for this, but before looking in detail at the predefined collectors, let's look at the process of creating a custom Collector.

Creating a Custom `Collector`: Methods to Implement

```
Supplier<A> supplier()
BiConsumer<A,T> accumulator()
BinaryOperator<A> combiner()
Function<A,R> finisher()
```

Types:

- T – the type of the stream elements
- A – the mutable accumulation type
- R – the result type

Unlike the method
`collect``(`**`Supplier`**`<R> s, `**`BiConsumer`**`<R,? super T> a, `**`BiConsumer`**`<R,R> c)`
the result type of a `Collector` is not necessarily the same as the accumulation type.

Compare the construction of the important Collector methods with the parameters passed in the method:

collect(Supplier<R> supplier,
BiConsumer<R,? super T> accumulator,
BiConsumer<R,R> c)

In the Collector interface, the result type of a Collector is not necessarily the same as the accumulation type. This is because the Collector interface has an additional method, finisher(), that can modify the result of the combiner operation so that the collector returns a different result type.

Custom Example `MyCustomCollector`

```
public class MyCustomCollector implements
Collector<Person, List, List> {
    public Supplier supplier() { return ArrayList::new; }
    public BiConsumer<List, Person> accumulator() {
        return List::add;
    }
    public BinaryOperator<List> combiner() {
        return (l1, l2) -> { l1.addAll(l2);
            return l1;
    }
    public Function<List, List> finisher() {
        return Function.identity();
    }
    public Set characteristics() {  return Set.of(); }
}
```

Unlike `collect` method that specifies combiner as an argument, here the combiner is a `BinaryOperator`.

The finisher operation allows result type different than combiner type. Here the finisher specifies using the combiner type as result type.

The characteristics method here is returning no characteristics for the collector.

Collector characteristics can be:

- `CONCURRENT` – Indicates that this collector is concurrent, meaning that the result container can support the accumulator function being called concurrently with the same result container from multiple threads.
- `IDENTITY_FINISH` – Indicates that the finisher function is the identity function and can be elided.
- `UNORDERED` – Indicates that the collection operation does not commit to preserving the encounter order of input elements.

Finisher Example `MyCustomCollector`

```
public class MyCustomCollector
   implements Collector<Person, List, Person[]> {

... < supplier, accumulator, and combiner as previous slide > ...

   public Function<List, Person[]> finisher() {
     return l -> (Person[]) l.toArray(new Person[0]);
   }
   public Set characteristics() {return
      Collections.singleton(Characteristics.UNORDERED);}
}
```

Result type different than accumulator type.

Converting the `ArrayList` to a `Person[]`.

Indicates that the collection operation does not guarantee preserving the order of the elements.

In the example, the collector has a different return type than the type used in the accumulator. An ArrayList is much more convenient to work with than an array, so it may be a better choice for the accumulator, but if a Person array is required, this could be provided by the finisher.

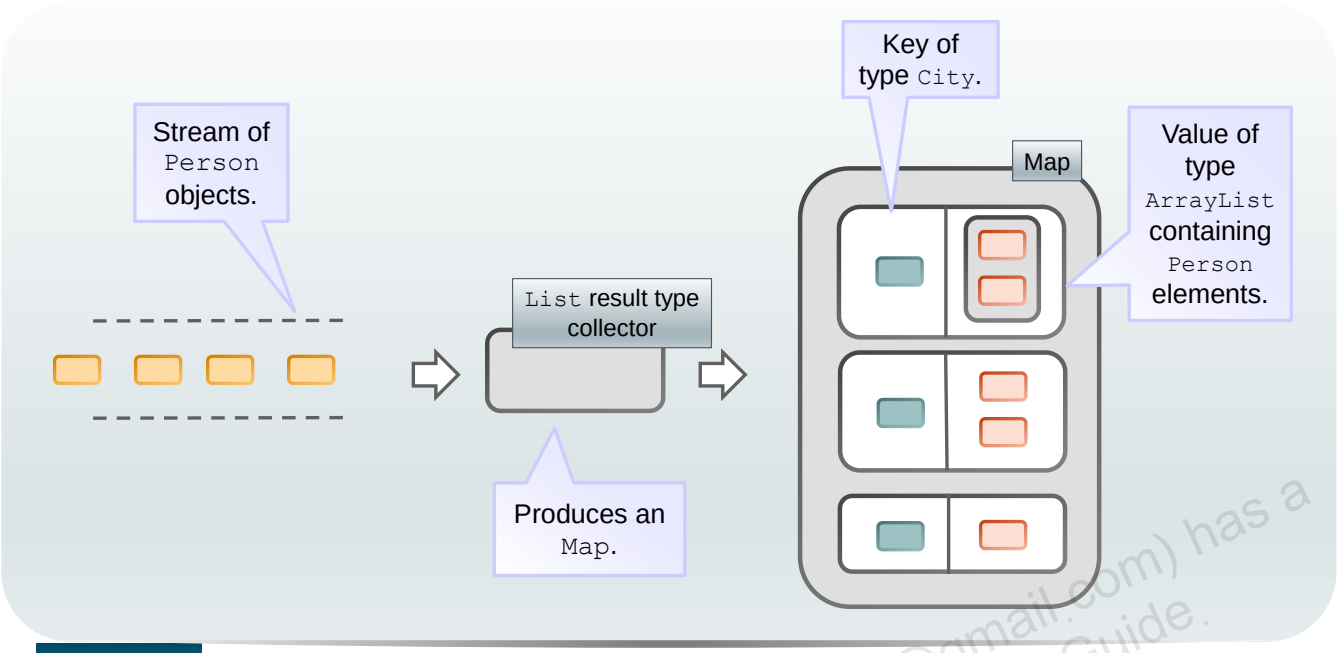
This is a very simple example and is intended to show the functionality of the collector's methods rather than a realistic typing decision.

Note that the Collector.of method can also be used to create a Collector. Making the Collector a standalone class makes sense if it's likely to be needed in more than one application, but Collector.of may be convenient if the Collector is not so general purpose. Note that it's still more reusable than collect `(Supplier<R> supplier, BiConsumer<R,? super T> accumulator, BiConsumer<R,R> combiner)` as you can create a reference to reuse, and it also gives you the opportunity to create a finisher and to set Characteristics.

For example:

```
Collector<Person, List, Person[]> myCustomCollector
            = Collector.of(
                    ArrayList::new,      // Supplier
                    List::add,               // Accumulator
                    (l1, l2) -> {            // Combiner - not called if
sequential stream
                        l1.addAll(l2);
                        return l1;
                    },
                    l -> (Person[]) l.toArray(new Person[0])    //
Collector.of() is overloaded so the finisher does not have to be included.
            );
```

A More Complex Collector



For coding a collector that groups Person elements within a Map requires coding the supplier, accumulator, combiner. The code is shown on the next page. It is quite a bit more involved than the simple collector for producing a List of elements.

In the next topic of this lesson, you look at another way to do this by combining predefined collectors. It is interesting to compare the complexity of coding a collector for this functionality versus combining collectors to achieve the same result.

A More Complex Collector `CustomGroupingBy`

```
Map<City,List<Person>> myPpl = people.stream()
    .collect(toCustomGroupingBy());

System.out.println("\nResult: " +
   myPpl.getClass().getCanonicalName() + " : " + myPpl + "\n");

Result Ppl4: java.util.HashMap : {Athens=[Amy, Eric], Tulsa=[Joe,
Eric], London=[Bill]}
```

- The functionality of the custom grouping collector is similar but less flexible than the supplied grouping by collectors.
- It illustrates why it is almost always better to build the collector you need by combining a number of the collectors from the `Collectors` class.

```
public class CustomGroupingBy implements Collector<Person, Map<City,
   List<Person>>, Map<City, List<Person>>> {
   public static  CustomGroupingBy toCustomGroupingBy() {
      return new CustomGroupingBy();
   }
   @Override
   public Supplier supplier() {
      return HashMap::new;
   }
   @Override
   public BiConsumer<Map<City, List<Person>>, Person> accumulator() {
      return (Map<City, List<Person>> h, Person p) -> {
         h.merge(p.getCity(),
            new ArrayList<>(List.of(p)),
           (List<Person> a, List<Person> b) -> {a.add(p); return a;});
      };
   }
   @Override
   public BinaryOperator<Map<City, List<Person>>> combiner() {
      return (Map<City, List<Person>> h1, Map<City, List<Person>> h2) -> {
         h2.forEach((City c, List<Person> l) -> {
            h1.merge(c, l,
                  (List<Person> a, List<Person> b) -> {
                     b.forEach((Person y) -> a.add(y));
                           return a;
                  });
         });
         return h1;
      };
   }
   @Override
   public Function<Map<City, List<Person>>,Map<City, List<Person>>>
      finisher() {
      return Function.identity();
   }
   @Override
   public Set characteristics() {
      return Collections.singleton(Characteristics.IDENTITY_FINISH);
   }
}
```

Summary

In this lesson, you should have learned how to:

- Create a collection
- Group elements into a collection
- Perform summarizing on a collection
- Compose collectors into a collection
- Create a custom collector

Practice 15: Overview

This practice covers the following topics:

- Practice 15-1: Review: A Comparison of Iterative Approach, Streams, and Collectors
- Practice 15-2: Using Collectors for Grouping