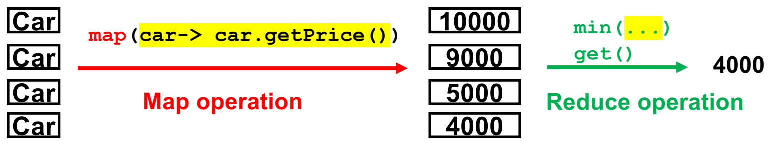
Map-Reduce Data Processing Pattern

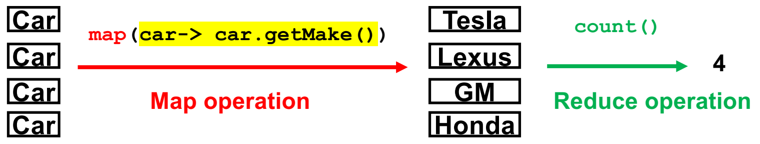
Map-Reduce Data Processing Pattern

- Intent
  - Obtain/generate a single value from a dataset through the *map* and *reduce* operations.
  - *Map* operation (intermediate operation)
    - Transforms an input dataset to another dataset
    - e.g., `map()`, `flatMap()`
  - *Reduce* operation (terminal operation)
    - Processes the transformed dataset to generate a *single* value
    - e.g. `count()`, `max()`, `min()`

```
Integer price = cars.stream()
                    .map( (Car car)-> car.getPrice() )
                    .min( Comparator.comparing((Integer price)-> price ) )
                    .get();
```

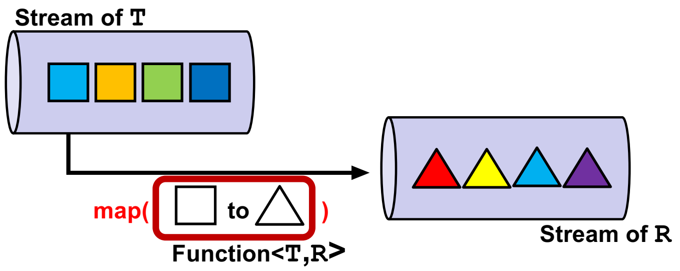


```
long carMakerNum = cars.stream()
                           .map( (Car car)-> car.getMake() )
                           .count();
```



`Stream.map()`

- `map(Function<T,R>)` : *intermediate* operation
  - Performs a stream-to-stream transformation
    - Takes a `Function` that converts a value of `T` to another of `R`.
      - `T` and `R` can be different types.
    - Applies the function on stream elements one by one.
    - Returns another stream of new values.
      - The # of elements do not change in b/w the input and output streams.



Stream.reduce()

- **Steam** provides "ready-made" reduce operations for common data processing tasks.
  - e.g. **count(), max(), min()**
- Use **reduce()** if you implement your own (i.e., custom) reduce operation

```
– Optional<T>   reduce(BinaryOperator<T> accumulator)

– T             reduce(T initVal,
                       BinaryOperator<T> accumulator)

– U             reduce(U initVal,
                       BiFunction<U,T> accumulator,
                       BinaryOperator<U> combiner)
```

1st Version of reduce()

- `Optional<T> reduce(BinaryOperator<T> accumulator)`
  - Takes a reduction function (`accumulator`) as a LE.
    - Applies it on stream elements (`T`) one by one.
  - Returns the reduced value (`T`).
  - `T result = aStream.reduce( (T result, T elem)-> {...} )`

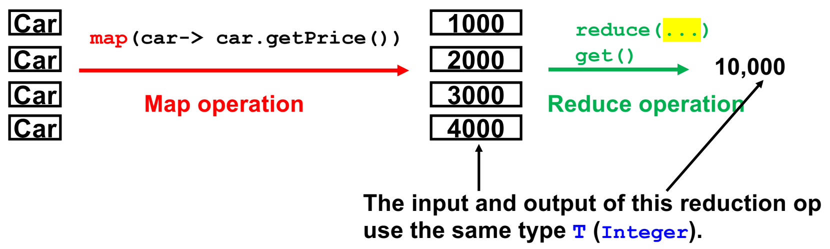
  -
    ```
    Iterator<T> it = collection.iterator();
    T result = it.next();          // first element
    while(it.hasNext()){           // for each remaining element
        T elem = it.next();
        result = accumulate(result, elem);
    }
    ```

	Params	Returns	Example use case
BinaryOperator<T>	T, T	T	Multiplying two numbers (*)

- `T result = aStream.reduce( (T result, T elem)-> {...} );`

-
  ```
  Iterator<T> it = collection.iterator();
  T result = it.next();          // first element
  while(it.hasNext()){           // for each remaining element
      T elem = it.next();
      result = accumulate(result, elem); }
  ```
- **result**
  - is *initialized* with the first element.
  - is *updated* in each iteration of the loop by
    - Getting accumulated with the next element (`elem`) with `accumulate()`
- A reduce operation (`accumulator`) is implemented as an anonymous version of `accumulate()`.
  - `accumulator`'s code block == `accumulate()`'s method body

-
  ```
  Integer totalValue
        = cars.stream().map( (Car car)-> car.getPrice() )
                       .reduce((result, price)->{result+price})
                       .get();
  ```


Car
Car
Car
Car
map(car-> car.getPrice())
Map operation
1000
2000
3000
4000
reduce(...)
get()
10,000
Reduce operation
The input and output of this reduction op use the same type T (Integer).

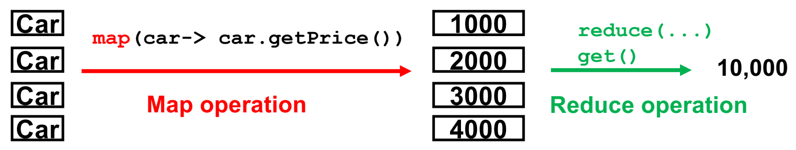
result = 1000
result = result + 2000 // 3,000
result = result + 3000 // 6,000
result = result + 4000 // 10,000

((1000 + 2000) + 3000) + 4000) = 10000

Important Note

- The order of stream elements is **NOT** guaranteed.
  - Even if a stream's source collection is ordered (`List`).
- A reduction function must be **associative**.

```
– Integer totalValue
      = cars.stream().map( (Car car)-> car.getPrice() )
                     .reduce( (result, price)->{result+price})
                     .get();
```



```
result = 1000                                  result = 3000
result = result + 2000      // 3,000          result = result + 4000      // 7000
result = result + 3000      // 6,000          result = result + 1000      // 8000
result = result + 4000      // 10,000         result = result + 2000      // 10000

((1000 + 2000) + 3000) + 4000) = 10000        ((3000 + 4000) + 1000) + 2000) = 10000
```

- Associative operator
  - (x **op** y) **op** z = x **op** (y **op** z)
  - e.g., Numerical sum, numerical product, string concatenation, max, min, union, product set, etc.
- Non-associative operators
  - e.g., Numerical subtraction, numerical division, etc.
    - (10 - 5) - 2 = 3 V.S. 10 - (5 - 2) = 7
    - 10/5/2 = 1 V.S. 10/(5/2) = 4

2nd Version of `reduce()`

- `T reduce(T initVal, BinaryOperator<T> accumulator)`
  - Takes the initial value (`T`) for the reduced value (i.e. reduction result) as the first parameter.
  - Takes a reduction function (`accumulator`) as the second parameter.
    - Applies the function on stream elements (`T`) one by one.
  - Returns the reduced value (`T`).

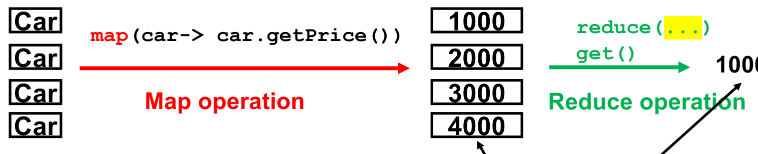
```
– T result = aStream.reduce(initValue, (T result, T elem)-> {...});

– T result = initValue;
  for(T element: collection){
      result = accumulate(result, element);
  }
```

	Params	Returns	Example use case
BinaryOperator<T>	T, T	T	Multiplying two numbers (*)

```
• T result = aStream.reduce(initValue, (T result, T elem)-> {...});

• T result = initValue;
  for(T element: collection){
      result = accumulate(result, element);
  }
```

- `result`
  - is *initialized* with `initValue`.
  - is *updated* in each iteration of the loop by
    - Getting accumulated with the next element (`elem`) with `accumulate()`
- A reduce operation (`accumulator`) is implemented as an anonymous version of `accumulate()`.
  - `accumulator`'s code block == `accumulate()`'s method body

-
  ```
  Integer minPrice
      = cars.stream().map( (Car car)-> car.getPrice() )
                     . reduce(0, (result, carPrice)->{
                             if(result==0) return carPrice;
                             else if(carPrice < result) return carPrice;
                             else return result;} );
  ```



The input and output of this reduction op use the same type `T` (`Integer`).

result = 0
result = 1000
result = 1000 (1000 < 2000)
result = 1000 (1000 < 3000)
result = 1000 (1000 < 4000)

- With `reduce()` in the Stream API

  -
    ```
    Integer price = cars.stream()
                        .map( (Car car)-> car.getPrice() )
                        .reduce(0, (result, carPrice)->{
                                if(result==0) return carPrice;
                                else if(carPrice < result) return carPrice;
                                else return result;} );
    ```
- In a traditional style

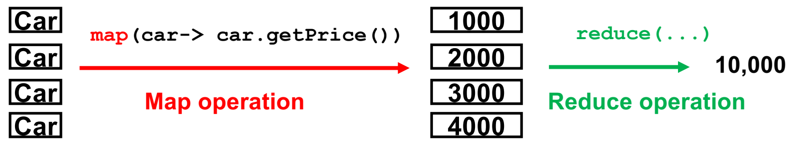
  -
    ```
    List<Integer> carPrices = ...
    int result = 0;
    for(Integer carPrice: carPrices){
            if(result==0) result = carPrice;
            else if(carPrice < result) result = carPrice;
            else result = result;
    }
    ```
- With `min()` in the Stream API

  -
    ```
    Integer price = cars.stream()
                        .map( (Car car)-> car.getPrice() )
                        .min( Comparator.comparing(price-> price) )
                        .get();
    ```

Important Note

- The order of stream elements is **NOT** guaranteed.
  - Even if a stream's source collection is ordered (`List`).
- A reduction operator must be **associative**.

  -
    ```
    Integer price = cars.stream()
                        .map( (Car car)-> car.getPrice() )
                        .reduce(0, (result, carPrice)->{
                                if(result==0) return carPrice;
                                else if(carPrice < result) return carPrice;
                                else return result;} );
    ```



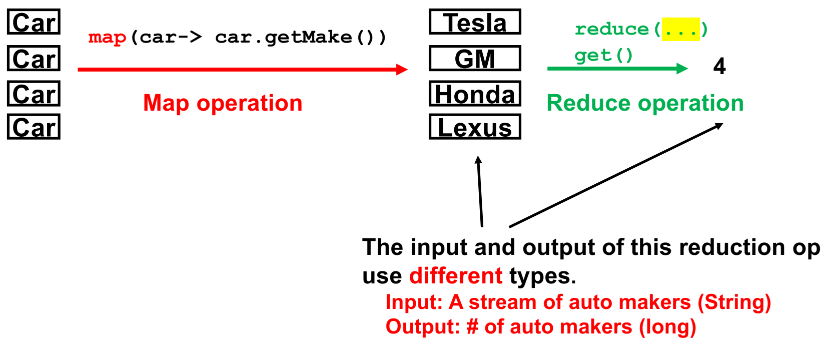
result = 0	result = 0
result = 1000	result = 3000
result = 1000 (1000 < 2000)	result = 3000 (3000 < 4000)
result = 1000 (1000 < 3000)	result = 1000 (1000 < 3000)
result = 1000 (1000 < 4000)	result = 1000 (1000 < 2000)

3rd Version of `reduce()`

-
  ```
  U reduce(U initVal,
           BiFunction<U,T> accumulator,
           BinaryOperator<U> combiner)
  ```
  - Takes the initial value (`U`) for the reduced value (i.e. reduction result) as the first parameter.
  - Takes a reduction function (`accumulator`) as the second parameter.
    - Applies the function on stream elements (`T`) one by one.
  - Takes a combination function (`combiner`) as the third parameter.
    - Applies the function on *intermediate* reduction results (`U`) one by one.
  - Returns the final (combined) result (`U`).
  - Useful when stream elements (`T`) and a reduced value (`U`) use different types.

	Params	Returns	Example use case
BinaryOperator<T>	T, T	T	Multiplying two numbers (*)
BiFunction<U,T>	U, T	U	Return TRUE (R) if two params (U and T) match.

- Think of implementing `count()` yourself with `reduce()`.
  - »
```
long carMakerNum = cars.stream()
                       .map( (Car car)-> car.getMake() )
                       .count();
```


Car
Car
Car
Car
map(car-> car.getMake())
Map operation
Tesla
GM
Honda
Lexus
reduce(...)
get()
4
Reduce operation
The input and output of this reduction op use different types.
Input: A stream of auto makers (String)
Output: # of auto makers (long)

```
• U finalResult = aStream.reduce(initValue,
                                 (U result, T element)-> {...} );
                                 (U finalResult, U intermediateResult)->...);
• U result = initValue;
  for(T element: collection){
      result = accumulate(result,element);
  }
```

- `result`
  - is *initialized* with `initValue`.
  - is *updated* in each iteration of the loop by
    - Getting accumulated with the next element (`elem`) with `accumulate()`

- A reduce operation (`accumulator`) is implemented as an anonymous version of `accumulate()`.
  - `accumulator`'s code block == `accumulate()`'s method body

- With `reduce()` in the Streams API
  -
```
long carMakerNum =
            cars.stream()
                .map( (Car car)-> car.getMake() )
                .reduce(0,
                        (result,carMaker)-> ++result,
                        (finalResult,intermediateResult)->finalResult);
```

- In traditional style
  -
```
List<String> carMakers = ...
long result = 0;
for(String carMaker: carMakers){
  if(carMaker != null){
         result++;
  }
}
long carMakerNum = result;
```

- With `count()` in the Streams API
  -
```
long carMakerNum = cars.stream()
                       .map( (Car car)-> car.getMake() )
                       .count();
```

- With `reduce()` in the Stream API
  -
```
long carMakerNum =
            cars.stream()
                .map( (Car car)-> car.getMake() )
                .reduce(0,
                  (result,carMaker)-> ++result,
                  (finalResult,intermidiateResult)->finalResult);
```

- `reduce()` executes `result = ++result;`

- Just in case, note that:
  -
```
int i = 0;
i++;          // i==1
int x = i++;  // i==1, x==0
int y = ++i;  // i==1, y==1
```

- If you use a sequential stream, just return `finalResult` in the second LE (combination function).

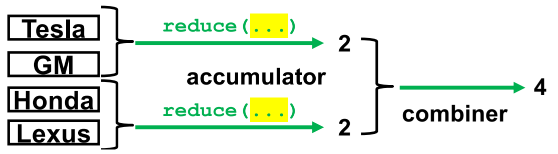
```
– long carMakerNum =
            cars.stream()
                .map( (Car car)-> car.getMake() )
                .reduce(0,
                        (result,carMaker)-> ++result,
                        (finalResult,intermidiateResult)->finalResult);
```

```
• Collection<Long> intermediateResults = ... //

  Iterator<Long> it = intermediateResults.iterator();
  Long finalResult = it.next();      // first element

  while(it.hasNext()){               // for each remaining element
      Long intermediateResult = it.next();
      finalResult = combine(finalResult, intermediateResult);
  }
```

- Using a sequential stream, there is ONLY ONE intermediate result.

- If you use a parallel stream, it
  - creates multiple threads to process the first LE (reduction function) in a parallel/concurrent manner.
    - Each thread generates one intermediate reduction result.
  - combines all intermediate reduction results.

```
– long carMakerNum =
    cars.stream()
        .parallel()
        .map( (Car car)-> car.getMake() )
        .reduce(
            0,
            (result,carMaker)-> ++result,
            (finalResult,intermidiateResult)->finalResult+intermediateResult);
```



3 Versions of reduce()

- If the input (stream elements) and output (reduced result) use the same type, use the 1st or 2nd version:

```
– Optional<T>   reduce(BinaryOperator<T> accumulator)

– T             reduce(T initVal,
                       BinaryOperator<T> accumulator)
```

  - Use the 2nd version if you need a custom initial value.

- If the input (stream elements) and output (reduced result) use the same type, use the 3rd version:

```
– U             reduce(U initVal,
                       BiFunction<U,T> accumulator,
                       BinaryOperator<U> combiner)
```

HW 2

- Implement your own min(), max() and count() with reduce() for a stream of Car instances.
  - Implementing min() with reduce()

```
– Integer price = cars.stream()
                      .map((Car car)-> car.getPrice())
                      .reduce(0, (result, carPrice)->{
                          if(result==0) return carPrice;
                          else if(carPrice < result) return carPrice;
                          else return result;} );
```

- Deadline: Feb 21 midnight

HW 3

- Recall HW 14 in CS680
  - Sorting Car instances with `Comparator` and `Collections.sort()`.
    - 4 different sorting policies: price-, year-, mileage-, and domination rank-based sorting.

- Do the same with Stream API.
  - c.f. Stream.sorted() and Stream.collect()
  - Implement 4 different sorting policies.

- Deadline: Feb 21 midnight