Reduce Operation

- **Steam** provides "ready-made" reduce operations for common data processing tasks.
  - e.g. `count()`, `max()`, `min()`, `sum()`, `average()`
- Use `reduce()` if you implement your own (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)-> {...} )
                    .get();
```

» `result`: result holder

» `elem`: next available element

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

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)-> {...} )
                    .get();

- 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);
  }
```



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

• 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`: *result holder*
  - 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((Integer result, Integer price)->{result+price})
          .get();
```

Car → 1000
Car → 2000
Car → 3000
Car → 4000

`map(car-> car.getPrice())`
Map operation

`reduce(...)`
Reduce operation
10,000

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 Notes (1)

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

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

1000
2000
3000
4000

`reduce(...)`
`get()`
Reduce operation
10,000

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

Important Notes (2)

- The order of applying a LE on stream elements is **NOT** guaranteed.
  - Even though stream elements are ordered.
    - A stream's elements are ordered if its source collection is ordered (e.g., `List`).
- A reduction function must be **associative**.

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

1000
2000
3000
4000

`reduce(...)`
`get()`
Reduce operation
10,000

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

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

3000
4000
1000
2000

`reduce(...)`
`get()`
Reduce operation
10,000

result = 3000
result = result + 4000 // 7000
result = result + 1000 // 8000
result = result + 2000 // 10000

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

- Associative operator
  - (x **op** y) **op** z = x **op** (y **op** z)
  - Gives the same result regardless of the way the operands are grouped.
  - e.g., Numerical sum, numerical product, string concatenation, max, min, matrix product, set union, set intersection, logical AND, logical OR, logical XOR, etc.
- Non-associative operators
  - e.g., Numerical subtraction, numerical division, power, logical NOR, logical NAND, etc.
    - (10 - 5) - 2 = 3 V.S. 10 - (5 - 2) = 7
    - (10/5)/2 = 1 V.S. 10/(5/2) = 4
    - (10^5)^2 V.S. 10^(5^2)

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(T initValue, (T result, T elem)-> {...});
```

» `result`: **result holder**

» `elem`: **next available element**

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

- `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(T 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 (*)

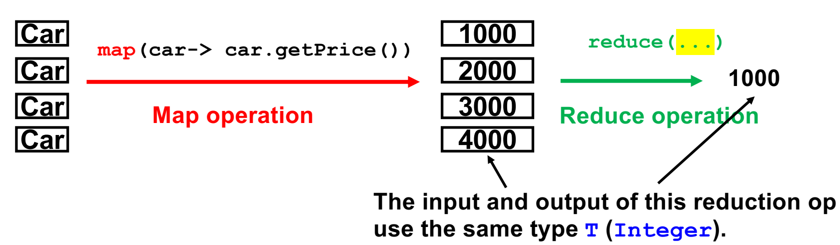

```
• T result = aStream.reduce(T initValue, (T result, T elem)-> {...});

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

- `result` : *result holder*
  - 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;} );
```



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 applying a LE on stream elements is **NOT** guaranteed.
  - Even though stream elements are ordered.
    - A stream's elements are ordered if its source collection is ordered (e.g., `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;} );
```

1000
2000
3000
4000
`reduce(...)` → 1000
Reduce operation

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

3000
4000
3000
2000
`reduce(...)` → 1000
Reduce operation

result = 0
result = 3000
result = 3000 (3000 < 4000)
result = 1000 (1000 < 3000)
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
BiFunction<U,T>	U, T	U	
BinaryOperator<U>	U, U	U	

- 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: T)
Output: # of auto makers (long: U)

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

- `result` : *result holder*
  - 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){
        result++;
    }
    long carMakerNum = result;
    ```

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


Important Notes

- The order of applying a LE on stream elements is **NOT** guaranteed.
  - Even though stream elements are ordered.
    - A stream's elements are ordered if its source collection is ordered (e.g., `List`).

- Reduction and combination operators must be **associative**.

- 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,x,y = 0; i++;            // i==1
    int i,x,y = 0; int x = i++;    // i==1, x==0
                                   // assignment of 0 first
    int i,x,y = 0; int y = ++i;    // i==1, y==1
                                   // increment of 0 first
    ```

- Just return `finalResult` (the first parameter) in the second LE unless you use a parallel stream in a multi-threaded app.

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

- When using a default (i.e. non-parallel, sequential) stream with a single thread, you never need to do anything extra in the second LE.
  – Just return what's at hand now, which is contained in the first parameter of the second LE, as the final map-reduce result.

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)
```

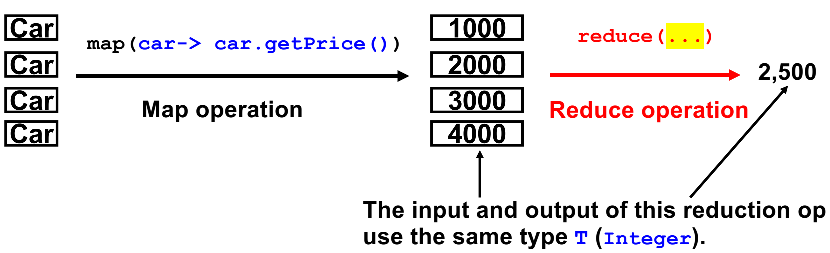
  – Use the 2nd version if you need a custom initial value.

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

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


Exercise: Average Car Price

```
• Integer averagePrice
    = cars.stream()
          .map( car -> car.getPrice() )
          .reduce( ... )
          .get();
```



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 different types, use the 3rd version:

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

Let's Try to Use the 1st Version of reduce()

```
• Integer averagePrice
    = cars.stream()
          .map( car -> car.getPrice() )
          .reduce( (average, price)-> {...} )
          .get();
```

```
Iterator<Iterator> it = prices.iterator();
Integer average = it.next();// first element
while(it.hasNext()){         // for each remaining elem
    Integer price = it.next();
    average = accumulate(average, price);}
```

Desirable algorithm

1000	price=1000	average=1000
2000	price=2000	average=1500
3000	price=3000	average=2000
4000	price=4000	average=2500


```
• Integer averagePrice
    = cars.stream()
          .map( car -> car.getPrice() )
          .reduce( (average, price)->{(average+price)/2} )
          .get();
```

Desirable algorithm

1000	price=1000	average=1000
2000	price=2000	average=1500
3000	price=3000	average=2000
4000	price=4000	average=2500

1000	price=1000	average=1000
2000	price=2000	average=(average+price)/2 = 1500
3000	price=3000	average=(average+price)/2 = 2250
4000	price=4000	average=(average+price)/2 = 3125


```
• Integer averagePrice
    = cars.stream()
          .map( car -> car.getPrice() )
          .reduce( (average, price)-> {(average+price)/2} )
          .get();
```

This (yellow-highlighted) LE DOES NOT work.

1000	price=1000	average=1000
2000	price=2000	average=(average+price)/2 = 1500
3000	price=3000	average=(average+price)/2 = 2250
4000	price=4000	average=(average+price)/2 = 3125

It should have worked like this:

1000	price=1000	average=1000
2000	price=2000	average=(average*1+price)/2 = 1500
3000	price=3000	average=(average*2+price)/3 = 2000
4000	price=4000	average=(average*3+price)/4 = 2500


```
• Integer averagePrice
    = cars.stream()
          .map( car -> car.getPrice() )
          .reduce( (average, price)-> {...} )
          .get();
```

An algorithm to be implemented:

1000	price=1000	average=1000
2000	price=2000	average=(average*1+price)/2 = 1500
3000	price=3000	average=(average*2+price)/3 = 2000
4000	price=4000	average=(average*3+price)/4 = 2500

To calculate the average correctly, the lambda expression requires:

(1) the number of the stream elements that have been examined AND

(2) the average of the elements that have been examined

```
• Integer averagePrice
    = cars.stream()
          .map( car -> car.getPrice() )
          .reduce( (..., price)-> {...} )
          .get();
```

1000	price=1000	average=1000
2000	price=2000	average=(average*1+price)/2 = 1500
3000	price=3000	average=(average*2+price)/3 = 2000
4000	price=4000	average=(average*3+price)/4 = 2500

To calculate the average correctly, the lambda expression requires:

(1) the number of the stream elements that have been examined AND

(2) the average of the elements that have been examined

Since `reduce()` takes only one parameter as a result holder, we pack the two data into a single parameter value. The simplest strategy here is to use an array.

The parameter (result holder) to be passed to the LE:

```
int[2]: [# of elements that have been examined,
         the average of those elements]
```

Result holder: array

Type of stream elements: Integer

The 1st version of reduce() uses the same type for the result holder and stream elements.

Cannot use the 1st version of reduce() anymore. In fact, the 2nd version does the same. We need to use the 3rd version.

Let's Use the 3rd Version of reduce()

```
Integer averagePrice
  = cars.stream()
        .map( car -> car.getPrice() )
        .reduce( new int[2],          // int[2]: [# of elems that have been examined,
             (result, price)->{...    //         the average of those elems]
                               ...
                               return result;}),
             (finalResult, intermediateResult)->finalResult
           )[1];
```

```
int[] result = new int[2];
for(Integer price: prices){
     result = accumulate(result, price);}
```

1000
2000 reduce(...) → [4, 2500]
3000
4000

Input: `Integer` **Output:** `int[2]`


```
Integer averagePrice
  = cars.stream()
        .map( car -> car.getPrice() )
        .reduce( new int[2],          // int[2]: [# of elems that have been examined,
             (result, price)->{...    //         the average of those elems]
                               ...
                               return result;}),
             (finalResult, intermediateResult)->finalResult
           )[1];
```

1000
2000 reduce(...) → [4, 2500]
3000
4000

Input: `Integer` **Output:** `int[2]`

1000 ([0, 0], 1000) → (0*0 + 1000)/(0++)
return [1,1000]

2000 ([1, 1000], 2000) → (1*1000 + 2000)/(1++)
return [2,1500]

3000 ([2, 1500], 3000) → (2*1500 + 3000)/(2++)
return [3,2000]

4000 ([3, 2000], 3000) → (3*2000 + 4000)/(3++)
return [4,2500]

Exercise

- Given a list of **`Car`**s,
  - Find the lowest and highest price.
    - Use the 2nd version of **`reduce()`**.
  - Compute the average price.
    - Use the 3rd version of **`reduce()`**.

HW 2

- A result holder can be a class instance
  - as well as a native-type value and an array.
- Compute the average car price with a class instance as a result holder (not an array as a result holder)
  - ```
    public class CarPriceResultHolder {
        private int numCarExamined;
        private double average;
        … }
    ```
  - Your map-reduce code:
    - ```
      double averagePrice
         = cars.stream()
               .map( car -> car.getPrice() )
               .reduce( new CarPriceResultHolder(),
                        (result, price)->{...
                                          ...
                                          return result;}),
                        (finalResult, intermediateResult)->finalResult
                      ).getAverage();
      ```