Parallel Streams

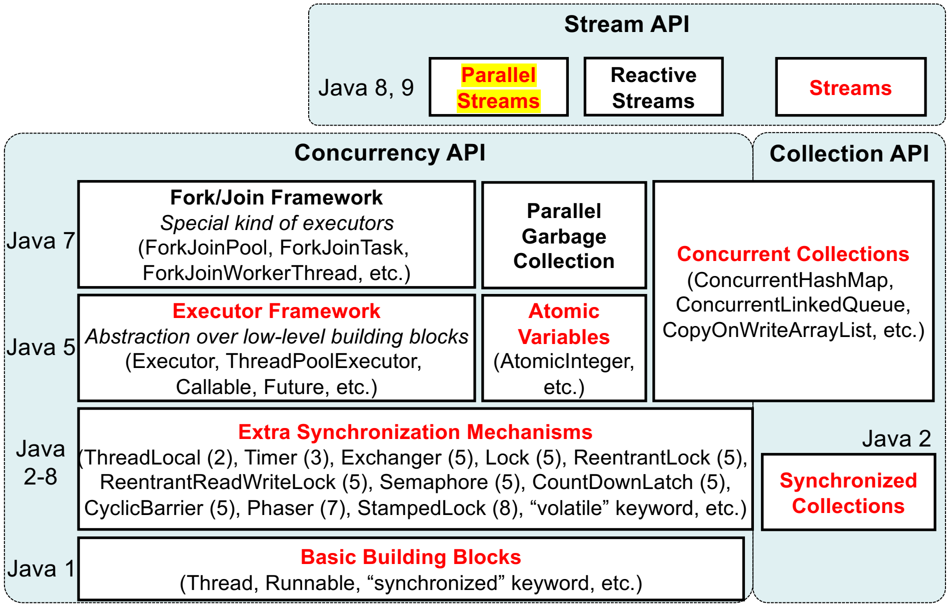
Streams and Collections

- Interface `Collection<T>`
  - `default Stream<T> stream()`
    - Returns a stream with this collection as its source.

- `java.util.stream.Stream<T>`
  - `Stream<T> filter(Predicate<T> predicate)`
    - Returns a stream consisting of the elements of this stream that match a given predicate (i.e. filtering criterion).
  - `long count()`
    - Returns the count of elements in this stream.

```
• long count = carList.stream()
                      .filter( (Car car)-> car.getPrice()<5000 )
                      .count();
```

Concurrency API in Java



Stream API

Java 8, 9 | Parallel Streams | Reactive Streams | Streams

Concurrency API | Collection API

Java 7 | **Fork/Join Framework** *Special kind of executors* (ForkJoinPool, ForkJoinTask, ForkJoinWorkerThread, etc.) | **Parallel Garbage Collection**

Java 5 | **Executor Framework** *Abstraction over low-level building blocks* (Executor, ThreadPoolExecutor, Callable, Future, etc.) | **Atomic Variables** (AtomicInteger, etc.)

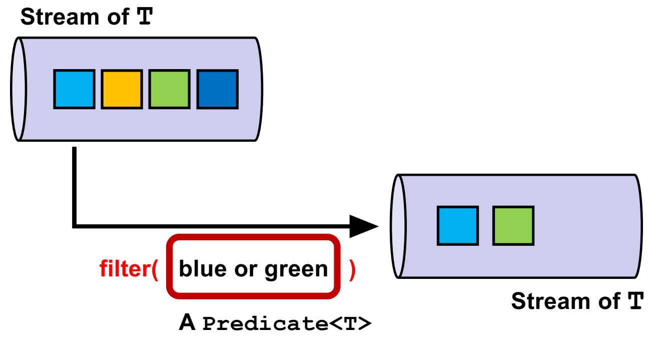
Concurrent Collections (ConcurrentHashMap, ConcurrentLinkedQueue, CopyOnWriteArrayList, etc.)

Java 2-8 | **Extra Synchronization Mechanisms** (ThreadLocal (2), Timer (3), Exchanger (5), Lock (5), ReentrantLock (5), ReentrantReadWriteLock (5), Semaphore (5), CountDownLatch (5), CyclicBarrier (5), Phaser (7), StampedLock (8), "volatile" keyword, etc.)

Java 2 | **Synchronized Collections**

Java 1 | **Basic Building Blocks** (Thread, Runnable, "synchronized" keyword, etc.)

	Params	Returns	Example use case
Function<T,R>	T	R	Get the price (R) from a Car object (T) Generate a function (R) from another (T)
Consumer<T>	T	void	Print out a collection element (T)
Predicate<T>	T	boolean	Has this car (T) had an accident?
Supplier<T>	NO	T	A factory method. Create a Car object and return it.
UnaryOperator<T>	T	T	Logical NOT (!)
BinaryOperator<T>	T, T	T	Multiplying two numbers (*)

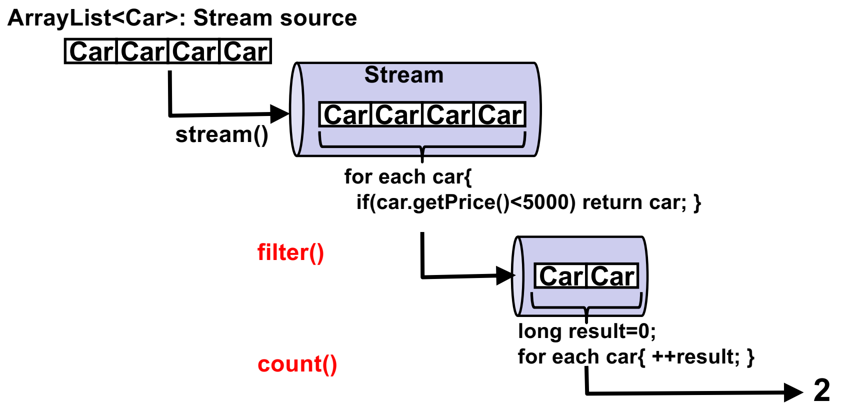


Stream Pipeline

- Multiple streams can be pipelined.

```
  – long count = carList.stream()
                           .filter( (Car car)-> car.getPrice()<5000 )
                           .count();
```

- Streams do NOT modify their source collection.



- Common structure of stream pipelines
  - Build a stream on a collection source
  - Perform zero or more *intermediate* operations
    - An intermediate operation returns a Stream.
    - e.g. `filter()`
  - Perform a *terminal* operation
    - A terminal operation returns non-Stream value or void.
    - e.g., `count()`

```
•  long count = carList.stream()
                           .filter( (Car car)-> car.getPrice()<5000 )
                           .count();
```

Concurrent/Parallel Stream Processing

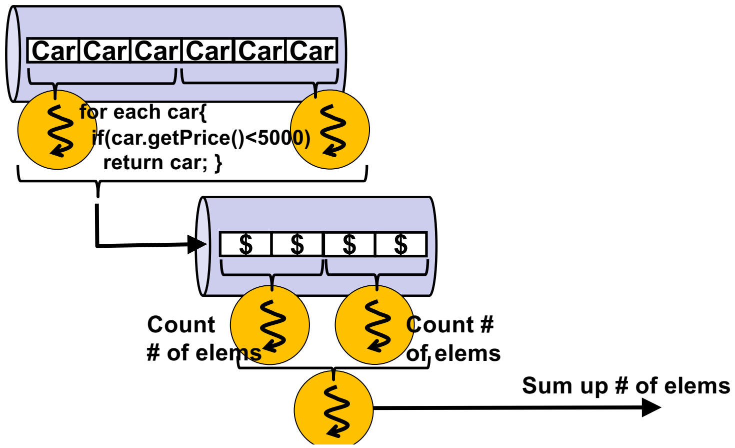
```
•  long count = carList.stream()
                           .filter( (Car car)-> car.getPrice()<5000 )
                           .count();

•  long count = carList.stream()
                           .parallel()
                           .filter( (Car car)-> car.getPrice()<5000 )
                           .count();
```

- `stream()` creates a *sequential* stream by default.
  - Executes all operations sequentially (with a single thread)

- `parallel()` turns a sequential stream to a *parallel* stream.
  - Executes all operations concurrently (with extra, multiple threads)

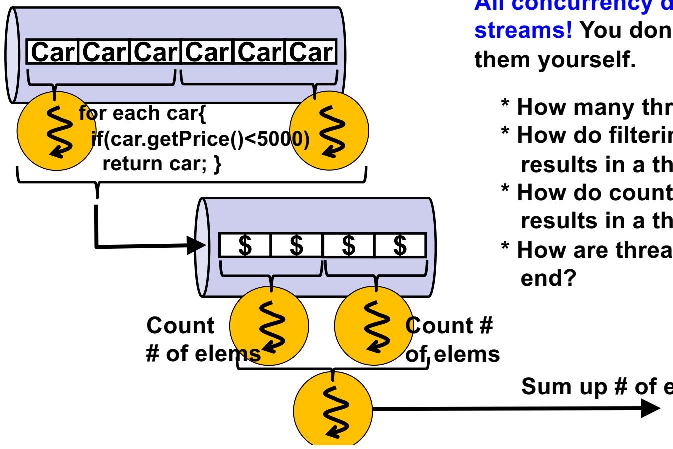
```
•  long count = carList.stream()
                           .parallel()
                           .filter( (Car car)-> car.getPrice()<5000 )
                           .count();
```

Parallel stream



-
```
long count = carList.stream()
                    .parallel()
                    .filter( (Car car)-> car.getPrice()<5000 )
                    .count();
```

Parallel stream



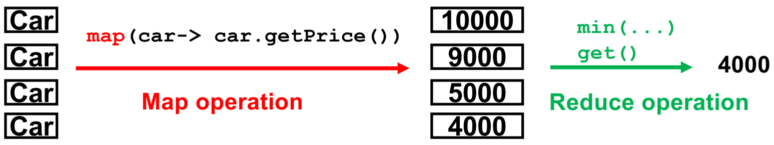
All concurrency details are hidden in streams! **You don't have to deal with them yourself.**

*** How many threads are used?**
*** How do filtering tasks combine results in a thread-safe way?**
*** How do counting tasks combine results in a thread-safe way?**
*** How are threads terminated in the end?**

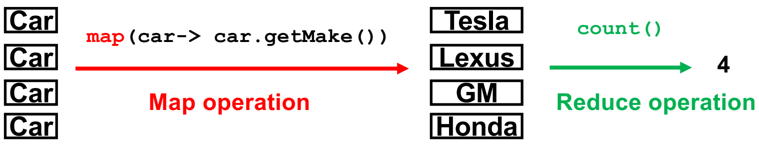
Map-Reduce Data Processing Pattern

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

```
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();
```



reduce()

- Steam API provides reduce operations for common data processing logic.
  - e.g. `count()`, `max()`, `min()`
- Use `reduce()` when you would like to implement your own 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)
```

	Params	Returns	Example use case
Function<T,R>	T	R	Get the price (R) from a Car object (T) Generate a function (R) from another (T)
Consumer<T>	T	void	Print out a collection element (T)
Predicate<T>	T	boolean	Has this car (T) had an accident?
Supplier<T>	NO	T	A factory method. Create a Car object and return it.
UnaryOperator<T>	T	T	Logical NOT (!)
BinaryOperator<T>	T, T	T	Multiplying two numbers (*)
BiFunction<U,T>	U, T	R	Return TRUE (R) if two params (U and T) match.

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 (as a LE) as the second parameter.
    - Applies the function on each stream element (`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 (*)

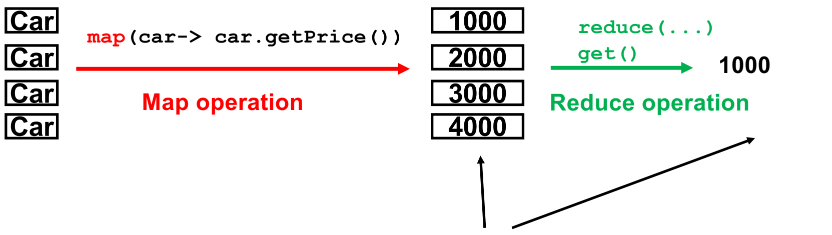
- `T result = aStream.reduce(initValue, (T result, T elem)-> {...} );`

-
  ```
  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 each element through accumulate()

- Reduce operations can be implemented in this form by varying initValue and accumulate().

-
  ```
  Integer lowestPrice
      = 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).

- With `min()` in the Stream API

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

- 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;
  }
```

3rd Version of reduce()

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

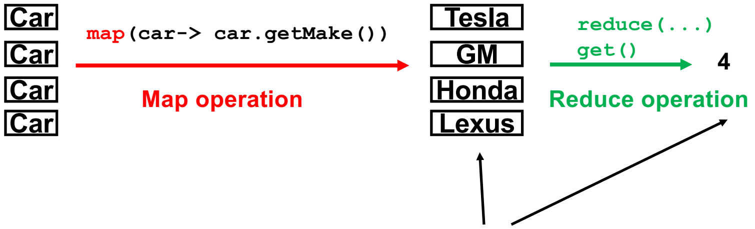
- Takes the initial value (`U`) for the reduced value (i.e. reduction result) as the first parameter.
- Takes a reduction function (as a LE) as the second parameter.
  - Performs the function on each stream element (`T`) one by one.
- Takes a combination function (as a LE) as the third parameter.
  - Performs the function on each intermediate reduction result (`U`).
- 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	R	Return TRUE (R) if two params (U and T) match.

- Suppose you are implementing `count()` yourself with `reduce()`.

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



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 each element through accumulate()

- Reduce operations can be implemented in this form by varying initValue and accumulate().

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

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

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

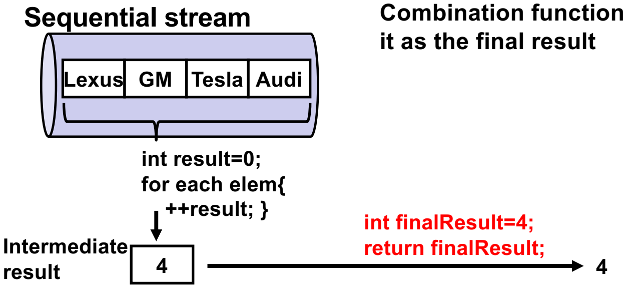
- Reduce operations can be implemented in this form by varying initValue and accumulate().

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

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

With a sequential stream, there is only one intermediate result.

Combination function should return it as the final result



- With **`count()`** in the Streams API

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

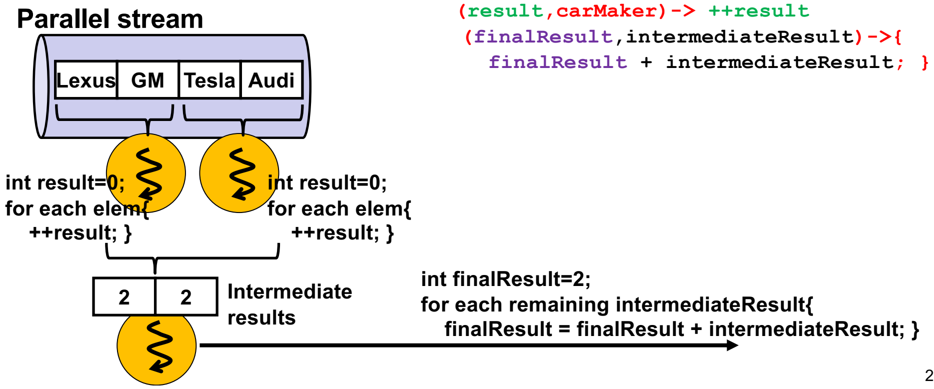
  -
    ```
    int carMakerNum =cars.stream()
                         .map( (Car car)-> car.getMake() )
                         .reduce(0,
                                (result,carMaker)-> ++result
                                 (finalResult,intResult)->finalResult);
    ```
- **`reduce()`** executes **`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
    ```

reduce() on a Parallel Stream

- A parallel stream uses the second LE (combination function) to combine all intermediate results.

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

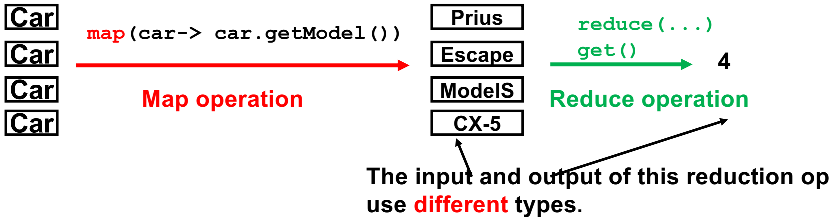


HW 19

- Implement a new data field and getter method in the class `Car`

```
– class Car{
    private String model;
    public String getModel();
    ... }
```

- Implement this map-reduce operation with
  - 3rd version of `reduce()`.
  - Parallel streams



The input and output of this reduction op use different types.

Arrays.parallelSort()

- **`Arrays`**
  - Utility class to process arrays of primitive type values and arrays of objects.
  - `sort()`
    - c.f. CS680
  - `parallelSort()`
    - Parallel (multi-threaded) version of `sort()`

```
– String fileContent = new String(
                           Files.readAllBytes(Paths.get(”test.txt”)));
  String[] words = fileContent.split("\\s+"); // White space as a
  Arrays.parallelSort(words);                 // delimiter

– Car[] cars = ...;
  Arrays.parallelSort( cars, (Car c1, Car c2)->{
                             (int)c1.getPrice()-c2.getPrice()));
```

Collections

- Utility class to process collections
  - `sort()`
    - c.f. CS680
  - `parallelSort()` is NOT available.

- To sort a collection in parallel (i.e., with multiple threads), create a `stream` of it, and then call `parallel()` and `sorted()`.

```
– List<Float> prices =
      cars.stream()
          .parallel()
          .sorted( (Car c1, Car c2)->{...} )
          .collect( Collectors.toList() );
```

HW 20

- Revise your HW 3 solution by using parallel streams
  - HW 3: Sort `Car` instances with sequential streams
    - Use 4 different ordering policies
    - c.f. `Stream.sorted()` and `Stream.collect()`
- Sort `Car` instances with parallel streams
  - Use 4 different ordering policies
  - c.f. `Stream.parallel()`

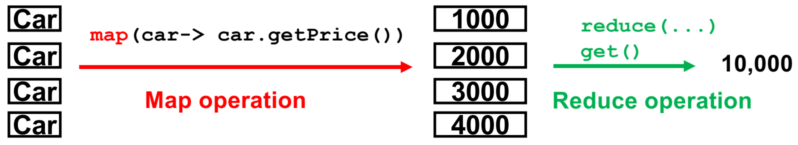
Thread Safety Issues in Parallel Streams

- Each lambda expression to be executed on a stream has to be associative and stateless.
  - If it’s not associative, the end result may be wrong.
  - If it’s not stateless (i.e. if it’s stateful), it may cause race conditions.

Associative Property of LEs

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

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

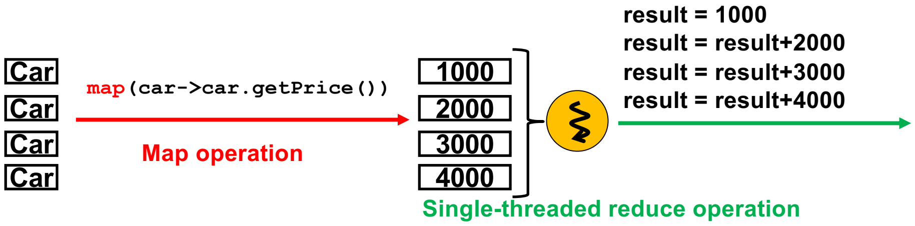
result = 3000
result = result + 4000 // 7000
result = result + 1000 // 8000
result = result + 2000 // 10000

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

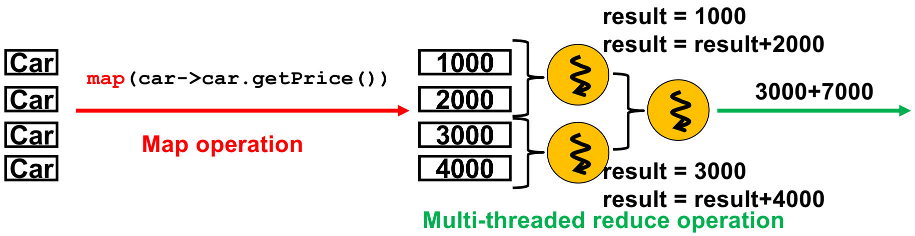
- Associative operator
  - (x **op** y) **op** z = x **op** (y **op** z)
  - ((a **op** b) **op** c) **op** d = (a **op** b) **op** (c **op** d)
  - e.g., Numerical sum, numerical product, string concatenation, max, min, union, product set, etc.
- Non-associative operators
  - e.g., Numerical subtraction,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

- A reduction function must be **associative**.

  -
    ```
    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
result = 1000
result = result+2000
result = result+3000
result = result+4000
Single-threaded reduce operation

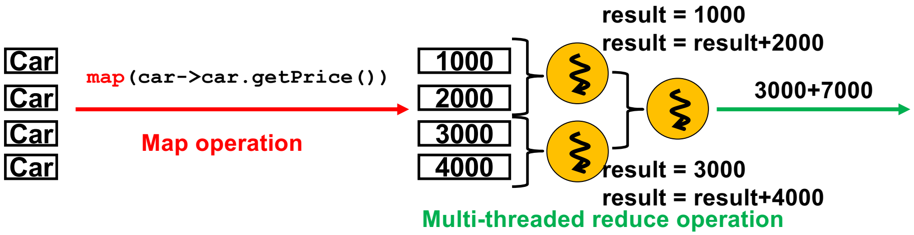


Car
Car
Car
Car
map(car->car.getPrice())
Map operation
1000
2000
3000
4000
result = 1000
result = result+2000
3000+7000
result = 3000
result = result+4000
Multi-threaded reduce operation

Stateless Property of LEs

- A reduction function must be **stateless**.

  -
    ```
    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
result = 1000
result = result+2000
3000+7000
result = 3000
result = result+4000
Multi-threaded reduce operation

- A LE should not use a shared variable (state).
  - Multiple threads should not access a shared variable.

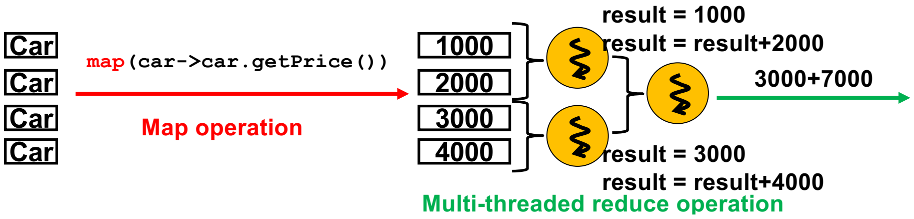
- `result`: local variable → No worries about race conditions
- Access to stream elements: thread safe → No worries about race conditions

- This LE is NOT **stateless**. (It is stateful.)

  -
    ```
    Integer totalValue;
    cars.stream().map( (Car car)-> car.getPrice() )
                 .forEach( (Integer value)->{totalValue+value});
    ```
  - Uses a shared variable (state): `totalValue`.
    - Multiple threads share it; Race conditions occurs.

- This LE is **stateless**.

  -
    ```
    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
result = 1000
result = result+2000
3000+7000
result = 3000
result = result+4000
Multi-threaded reduce operation