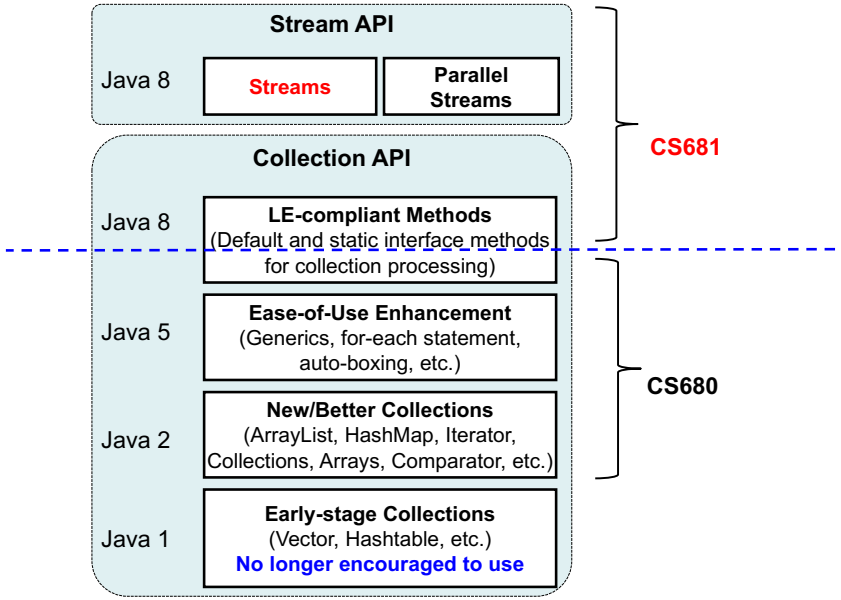
Stream API

Collection and Stream APIs



General-Purpose Functional Interfaces

- Since its version 8, Java has a lot of ***functional interfaces***.
  - e.g., `Comparator<T>`: Used for a specific purpose:
    - Comparison of two data (objects)
    - c.f. `Collections.sort()`
- In addition to *special-purpose* functional interfaces, Java has *general-purpose* ones that can be generally used in many scenarios or for many purposes.

Important General-Purpose Functional Interfaces

	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.

Important General-Purpose Functional Interfaces

	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) Comparator.comparing(), Map.computeIfAbsent(), Map.computeIfPresent()
Consumer<T>	T	void	Print out a collection element (T). Iterable.forEach()
Predicate<T>	T	boolean	Has this car (T) had an accident? Collection.removeIf()
Supplier<T>	NO	T	A factory method. Create a Car object and return it.
UnaryOperator<T>	T	T	Logical NOT (!) List.replaceAll()
BinaryOperator<T>	T, T	T	Multiplying two numbers (*)
BiFunction<U,T>	U, T	R	Return TRUE (R) if two params (U and T) match. Map.compute()

Collection Processing with LEs

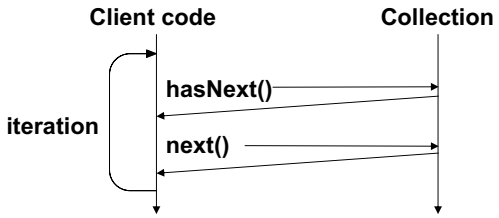
- Java 8 made major improvements to the Collection API by
  - Adding new static and default methods in existing interfaces
    - e.g., `Iterable.forEach()`
  - Adding **streams**.
    - `java.util.stream.Stream<T>`
      - Has many methods that take care of common operations to be performed on a `Collection`.

Traditional Way of Collection Processing

- ***External*** iteration: Iterate over a collection and performs an operation on each element in turn.

```
Iterator<ArrayList> iterator = strList.iterator();
while( iterator.hasNext() ) {
  System.out.print(iterator.next()); }
```

- Iteration occurs outside of a collection.
- Need to write a boilerplate code whenever you need to iterate over the collection.



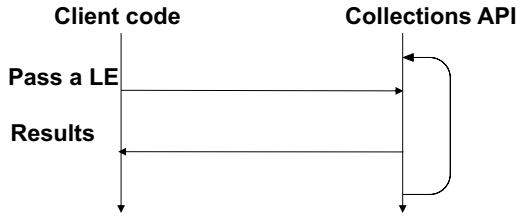
- The loop mixes up *what you want to do on a collection* and *how you do it*.
  - May not be that maintainable because "what" is often obscured by "how." ("How" is often emphasized too much than "what.")

```
Iterator<ArrayList> iterator = strList.iterator();
while( iterator.hasNext() ) {
  System.out.print(iterator.next()); }
```

- Inherently serial
  - Hard to make it concurrent/parallel.

New Way of Collection Processing

- ***Internal*** iteration:
  - `Iterable.forEach()`
    - Does not return an `Iterator` that externally controls the iteration
    - Creates an equivalent object, which works inside of a collection.
      - A collection internally uses the iterator-like object to perform iteration
  - `strList.forEach( (String i)->System.out.println(i) )`

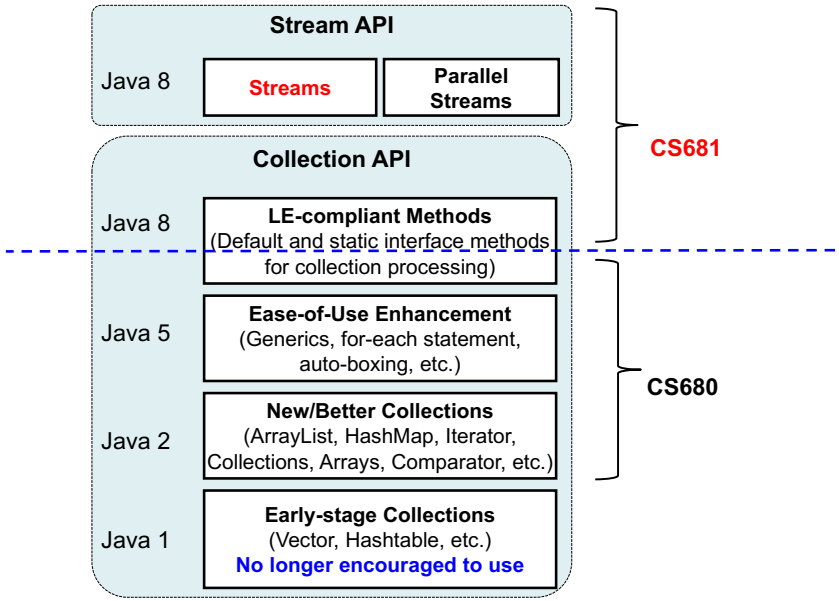


- Client code simply states "what" you want to do on a collection. "How" is hidden.
  - Collection processing looks more declarative, not procedural.
    - c.f. SQL statements

Collection Processing with LEs

- Java 8 made major improvements to the Collection API by
  - Adding new static and default methods in existing interfaces
    - e.g., `Iterable.forEach()`
  - Adding **streams**.
    - `java.util.stream.Stream<T>`
      - Has many methods that take care of common operations to be performed on a `Collection`.

Collection and Stream APIs

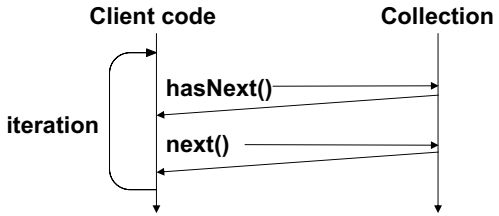


Traditional Way of Collection Access

- ***External*** iteration: Iterate over a collection and performs an operation on each element in turn.

```
int count = 0;
Iterator<Car> it = carList.iterator();
while( iterator.hasNext() ){
    Car car = iterator.next();
    if( car.getPrice() < 5000 ) count++; }
```

- Iteration occurs outside of a collection.
- Need to write a lot of boilerplate code whenever you need to iterate over the collection.



- The loop mixes up *what you want to do on a collection* and *how you do it*.
  - May not be that maintainable because "what" is often obscured by "how." ("How" is often emphasized too much than "what.")

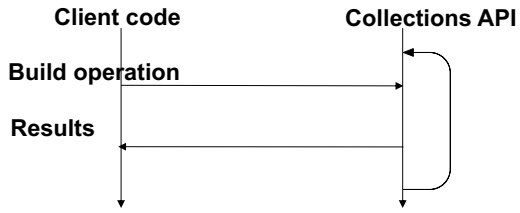
```
int count = 0;
Iterator<Car> it = carList.iterator();
while( iterator.hasNext() ){
    Car car = iterator.next();
    if( car.getPrice() < 5000 ) count++; }
```

- Inherently serial
  - Hard to make it concurrent/parallel.

New Way of Collection Access

- ***Internal*** iteration:
  - `stream()`: Plays a similar role to the call of `iterator()`
    - Does not return an `Iterator` that externally controls the iteration
    - Returns an equivalent object, a `Stream`, which works inside of a collection.
    - Helps build a complex operation on a collection.

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



- Client code simply states "what" you want to do on a collection. "How" is hidden.
  - Get a stream.
  - Filter the stream to keep the `Car` objects that are less expensive than $5000
  - Count the number of those `Car` objects in the steam.

- Stream API does NOT modify the elements of the source collection.

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

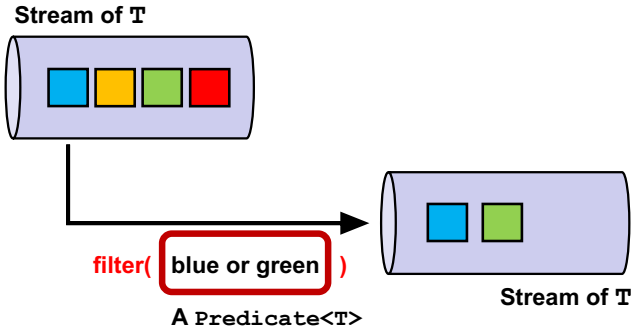
Streams and Collections

- Interface `Collection<E>`
  - `default Stream<E> stream()`
    - Returns a stream that uses this collection as its data 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.

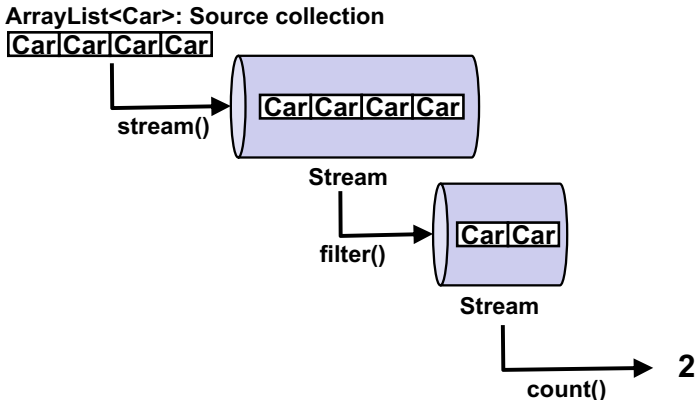
```
long count = carList.stream()
                    .filter( (Car car)-> car.getPrice()<5000 )
                    .count();
```


	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.



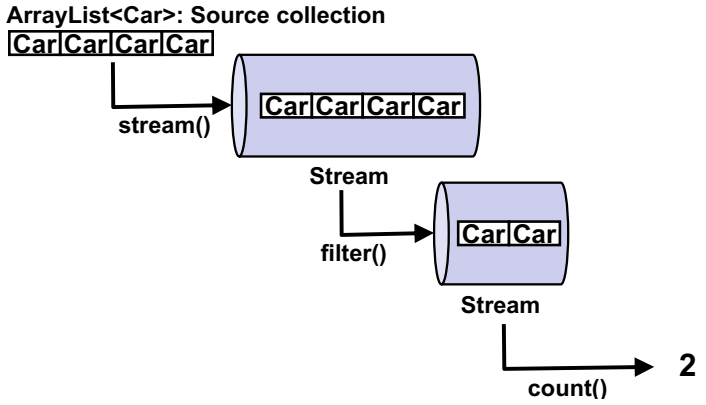
- Common steps to pipeline streams
  - Build a stream on a source collection
  - Perform zero or more *intermediate* operations
    - Each intermediate operation returns a Stream.
  - Perform a *terminal* operation
    - A terminal operation returns non-Stream value or void.

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

How Many Traversals?

- Traditional
  - ```
    int count = 0;
    Iterator<Car> it = carList.iterator();
    while( iterator.hasNext() ){
        Car car = iterator.next();
        if( car.getPrice() < 5000 ) count++;
    }
    ```
  - Traversing the list once .
- New
  - ```
    long count = carList.stream()
                        .filter( (Car car)-> car.getPrice()<5000 )
                        .count();
    ```
  - Traversing the list twice?
  - No, only once!

- Useful to intuitively understand the structure and behavior of a stream pipeline.
  - Real/internal traversal execution is a bit different.



Lazy and Eager Operations

- All intermediate operations are *lazy*.
- All terminal operations are *eager*.
- `filter()`: *intermediate* operation (lazy)
  - Does NOT perform filtering immediately when it is called.
  - Just prepares the filtering task and *delays* the task's execution until a terminal operation is invoked.
- `count()`: *terminal* operation (eager)
  - Is executed immediately when it is called.
- ```
  long count = carList.stream()
                      .filter( (Car car)-> car.getPrice()<5000 )
                      .count();
  ```


- No intermediate operations are executed until a terminal operation is called.
- ```
  long count = carList.stream()
                      .filter( (Car car)-> car.getPrice()<5000 );
  ```
  - The filtering operation is never executed.
- ```
  long count = carList.stream()
                      .filter((Car car)->{
                                System.out.println(car.getPrice());
                                car.getPrice()<5000} );
  ```
  - Nothing is printed out.
  - The filtering operation is never executed.
  ```

Important Methods of Stream

- `of(T... values)`: a static method of `Stream`
  - Builds a stream with values (not a collection) as source data

```
List<String> collected = Stream.of("u", "m", "b")
                    .map((String str)-> str.toUpperCase())
                    .collect( Collectors.toList() );
// a list of "U", "M" and "B" is returned.
```

```
String[] strs = {"u", "m", "b"};
List<String> collected = Stream.of(strs)
                    .map((String str)-> str.toUpperCase())
                    .collect( Collectors.toList() );
// a list of "U", "M" and "B" is returned.
```


Important Methods of Stream

- `of(T... values)`: a static method of `Stream`
  - Builds a stream with values (not a collection) as source data

```
List<String> collected = Stream.of("u", "m", "b")
                    .map((String str)-> str.toUpperCase())
                    .collect( Collectors.toList() );
// a list of "U", "M" and "B" is returned.
```

```
String[] strs = {"u", "m", "b"};
List<String> collected = Stream.of(strs)
                    .map((String str)-> str.toUpperCase())
                    .collect( Collectors.toList() );
// a list of "U", "M" and "B" is returned.
```


- `generate(Supplier<T> s)`: a static method of `Stream`
  - Returns an infinite stream in which each element is generated by repeatedly applying the provided `Supplier`.

```
Stream<Double> randomNums = Stream.generate( ()-> Math.random() );
// an infinite number of random numbers are generated.
```

-
```
Stream<Double> randomNums = Stream.generate( Math::random );
// an infinite number of random numbers are generated.
```

```
Stream<Double> randomNums = Stream.generate( ()-> Math.random() )
                                  .limit(100);
// First 100 random numbers are taken out and returned.
```

	Params	Returns	Example use case
Supplier<T>	NO	T	A factory method. Create a Car object and return it.

- *Method references* in lambda expressions
  - *object::method*
    - `System.out::println` (`System.out` contains an instance of `PrintStream`.)
    - `(int x) -> System.out.println(x)`
  - *Class::staticMethod*
    - `Math::max`
    - `(double x, double y) -> Math.max(x, y)`
  - *Class::method*
    - `Car::getPrice`
    - `(Car car)-> car.getPrice()`
    - `Car::setPrice`
    - `(Car car, int price)-> car.setPrice(price)`

- `iterate(T seed, UnaryOperator<T> f)`: a static method of `Stream`
  - Returns an infinite stream produced by iteratively applying the provided `UnaryOperator` to the initial element `seed`.
    - Generated elements: seed, f(seed), f(f(seed)), …

```
Stream<Integer> integers =
    Stream.iterate( 0, (Integer i)-> i.intValue()+1 );
// Infinite sequence of 0, 1, 2, 3...
```

```
Stream<Integer> oddNums =
    Stream.iterate( 1, (Integer i)-> i.intValue()+2 );
// Odd numbers: 1, 3, 5, 7...
```

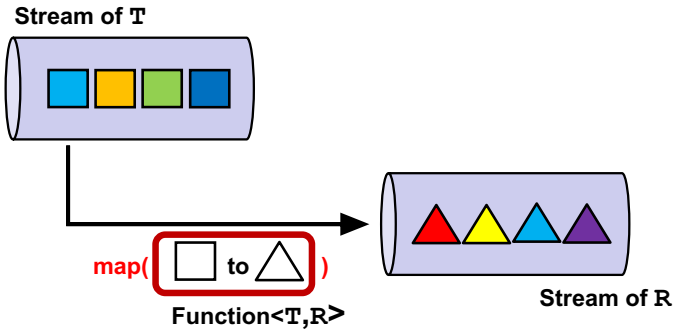
```
Stream<Integer> oddNums =
    Stream.iterate( 1, (Integer i)-> i.intValue()+2 );
        .skip(2);
// First 3 odd numbers are removed: 5, 7...
```

	Params	Returns	Example use case
UnaryOperator<T>	T	T	Logical NOT (!)

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

```
List<String> collected = Stream.of("u", "m", "b")
                    .map((String str)-> str.toUpperCase())
                    .collect( Collectors.toList() );
// a list of "U", "M" and "B" is returned.
```


	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 (*)

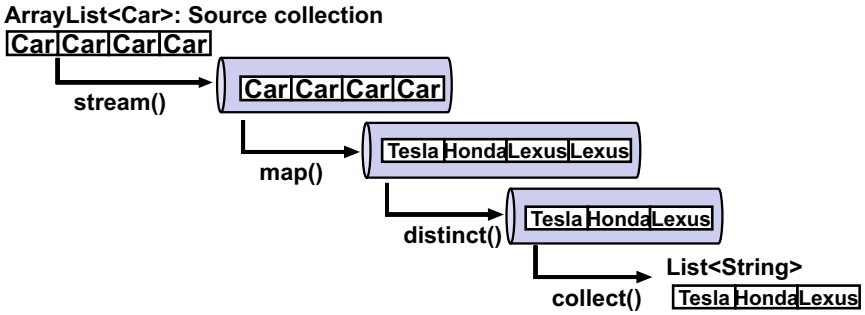


- `collect(Collector)`: *terminal* operation
  - Collects a set of values from a stream and returns it with a particular collection type.

```
List<String> collected = Stream.of("u", "m", "b")
                    .map((String str)-> str.toUpperCase())
                    .collect( Collectors.toList() );
// a list of "U", "M" and "B" is returned.
```

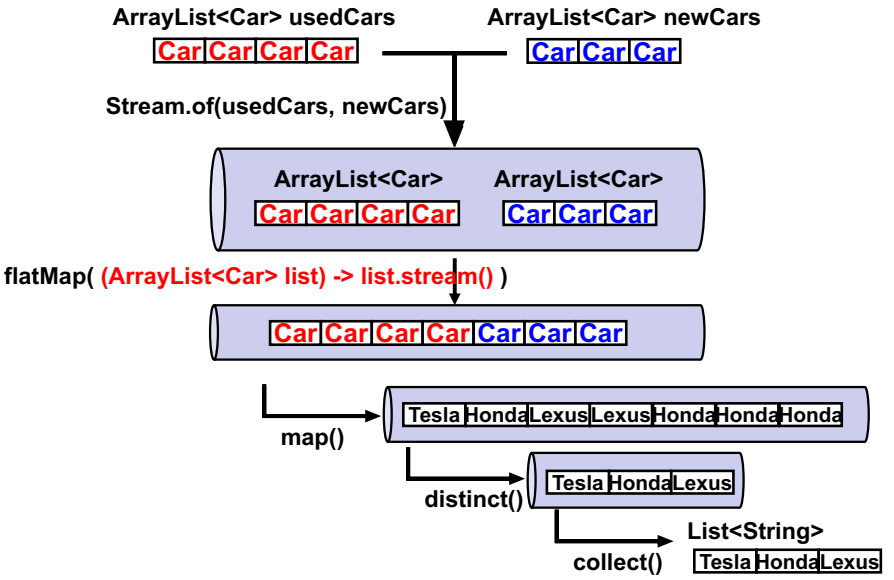
  - `Collectors`: Accumulates values in a stream and transforms the accumulated ones into a particular type
    - `Collectors.toList()`
    - `Collectors.toSet()`
    - `Collectors.toMap()`
    - `Collectors.toCollection(...)`
      - Can state a specific collection type/class.
      - `Collectors.toCollection(ArrayList::new)`
      - `Collectors.toCollection(TreeSet::new)`
      - `Collectors.toCollection(TreeMap::new)`

- `map(Function<T,R>)`: *intermediate* operation (cont'd)
  -
```
List<String> makes = cars.stream()
                          .map( (Car car)-> car.getMake() )
                          .distinct()
                          .collect(Collectors.toList());
```
- `distinct()`: *intermediate* operation
  - Removes redundant elements and returns a stream consisting of distinct elements



- `sorted(Comparator)`: *intermediate* operation
  - Sorts the elements of this stream according to the provided `Comparator` and returns the sorted stream.
  -
```
List<String> makes =
  cars.stream()
      .map( (Car car)-> car.getMake() )
      .distinct()
      .sorted( Comparator.comparing((Car car)-> car.getPrice()) )
      .collect( Collectors.toList() );
```
  - Higher-order function, comparing(): c.f. CS680

- `flatMap(Function<T,R>)`: *intermediate* operation
  - Converts each element of a steam to a (separate) steam and then…
  - Concatenates all the (converted) streams into a single stream.
  - `R` must be a stream.
  -
```
ArrayList<Car> usedCars = ...
ArrayList<Car> newCars = ...
List<String> makes = Stream.of(usedCars, newCars)
                           .flatmap( list-> list.stream() )
                           .map( (Car car)-> car.getMake() )
                           .distinct()
                           .collect(Collectors.toList());
```

- `max(Comparator<T>)`: *terminal* operation
  - Returns the maximum value according to the provided `Comparator`.
- `min(Comparator<T>)`: *terminal* operation
  - Returns the minimum value according to the provided `Comparator`.

```
Integer p = cars.stream()
                .filter( (Car car)-> !car.hadAccidents() )
                .map( (Car car)-> car.getPrice() )
                .filter( (Integer price)-> price<5000 )
                .max( Comparator.comparing( (Integer price)-> price ))
                .get();
```

- `max()` and `min()` returns `Optional<T>`.
  - An `Optional` represents a value that may or may not exist.
    - It does not exist if `max()` or `min()` is called on an empty steam.

- `get()` of `Optional<T>`
  - If this `Optional` contains a value, returns the value.
  - Otherwise, throws `NoSuchElementException`.

- `isPresent()` of `Optional<T>`
  - Checks if this `Optional` contains a value.

```
Optional<Integer> p =
    cars.stream()
        .filter( (Car car)-> !car.hadAccidents() )
        .map( (Car car)-> car.getPrice() )
        .filter( (Integer price)-> price<5000 )
        .max( Comparator.comparing( (Integer price)-> price ));
if( p.isPresent() ){
    System.out.println( p.get() ); }
```


- `forEach(Consumer<T>)`: *terminal* operation
  - Applies an LE on each stream element.

```
cars.stream()
    .map( (Car car)-> car.getMake() )
    .distinct()
    .forEach( (String autoMaker)-> System.out.println(autoMaker));
```


Methods that Return Streams

- Since its version 8, Java API has many methods that return streams.
- `java.nio.file.Files`
  - A utility class (i.e., a set of static methods) to process a file/directory.
    - Java NIO: c.f. CS680
  - `lines(Path path)`: Reads all lines from a file as a Stream.

```
Path path = Paths.get("/Users/jxs/temp/log.txt");
try( Stream<String> lines = Files.lines(path) ){
  long posts =
    lines.filter( (String line)-> line.contains("POST") )
         .count();
}
```

    - Try-with-resources statement: c.f. CS680

Exercise

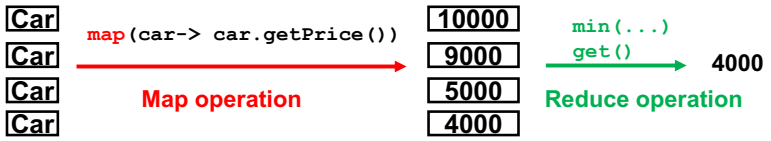
- Experience major methods in the Stream API
  - e.g., With a CS680 HW in which you implemented the class Car and sorted Car objects.

Map-Reduce Data Processing Pattern

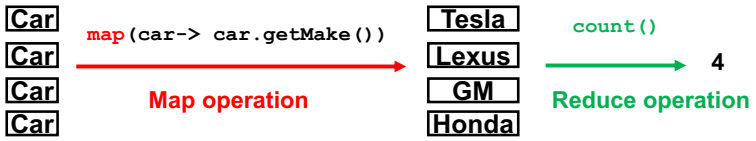
Map-Reduce Data Processing Pattern

- Intent
  - Obtain/generate a single value from a dataset through the *map* and *reduce* operations.
  - *Map* operation
    - 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()`


```
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 "ready-made" reduce operations for common data processing logic.
  - e.g. `count()`, `max()`, `min()`

- Use `reduce()` if you 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.

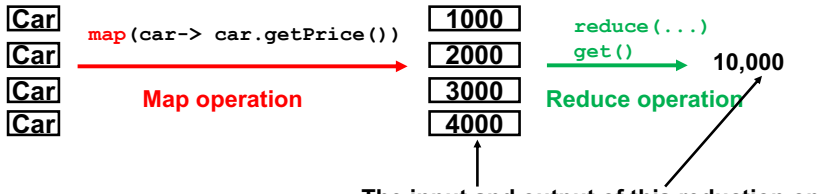
1st Version of reduce()

- `Optional<T> reduce(BinaryOperator<T> accumulator)`
  - Takes a reduction function as a LE.
    - Performs it on each stream element (`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 an element (e.g. the first element).
  - is *updated* in each iteration of the loop by
    - Getting accumulated with the next element through accumulate()
- Reduce operations can be implemented in this form by varying accumulate().

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



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

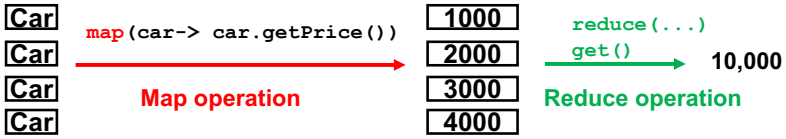
result = 1000
result = result + 2000 // 3,000
result = result + 3000 // 6,000
result = result + 4000 // 10,000

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

Important Note

- The order of stream elements is **NOT** guaranteed.
- A reduction operator (i.e. LE body) 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)
  - e.g., Numerical sum, numerical product, string concatenation, max, min, union, product set, etc.
- Non-associative operators
  - e.g., Numerical subtraction
    - (10 - 5) – 2 = 3
    - 10 - (5 - 2) = 7

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.
    - Performs 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 (*)

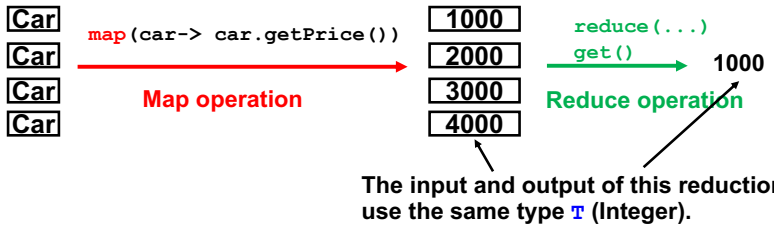
- `T result = aStream.reduce(initValue, (T result, T elem)-> {...} );`

- ```
  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 totalValue
      = 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;
    }
    ```

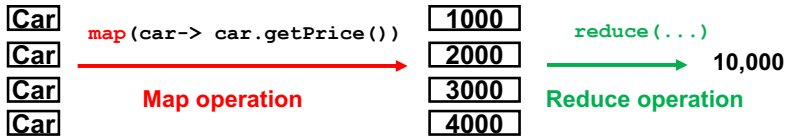
- With **`min()`** in the Stream API

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


Important Note

- The order of stream elements is **NOT** guaranteed.
- A reduction operator (i.e. LE body) 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 = 1000
result = 1000 (1000 < 2000)
result = 1000 (1000 < 3000)
result = 1000 (1000 < 4000)

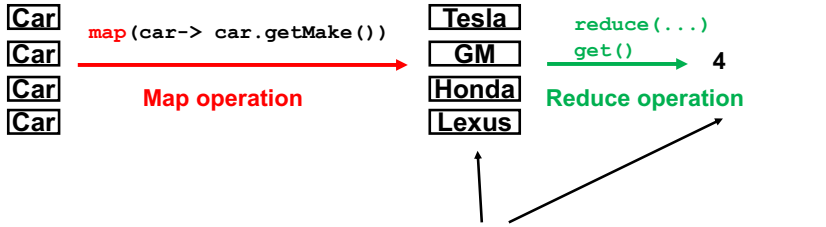
result = 0
result = 3000
result = 3000 (3000 < 4000)
result = 1000 (1000 < 3000)
result = 1000 (1000 < 2000)

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 (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	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();`



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().

- 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,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
```


- 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)-> {++result} );
                (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().

- A parallel stream uses the second LE (combination function) to safely process stream elements in a parallel/concurrent manner.
  - To be discussed in the 2nd half of this semester.

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

-
```
U result1 = initValue;
for(T element: collection1){
  result1 = accumulate(result1,element);
}
finalResult = finalResult + result1;
```

```
U result2 = initValue;
for(T element: collection2){
  result2 = accumulate(result2,element);
}
finalResult = finalResult + result2;
```


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 some 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)
```

Just In Case…

- `Stream<t> sorted(Comparator<? super T> comparator)`
  - "? super T" means any super type (super class) of T.

- `Stream<R> map(Function<? Super T, ? extends R> mapper)`
  - "? super T" means any super type (super class) of T.
  - "? extends T" means any sub type (subclass) of T.

```
Object result = cars.stream()
                    .map( (Object car)-> car.toString() )
                    .reduce( String result,
                             (result, str)-> result + str );
```


collect() and Collectors

- `collect(Collector)`: *terminal* operation
  - Collects a set of values from a stream and returns it with a particular collection type.
- `Collecters`: accumulates values in a stream and transforms the accumulated ones into a particular type

```
Object result = cars.stream()
                    .map( (Object car)-> car.toString() )
                    .reduce( String result,
                             (result, str)-> result + str );
```

```
Object result = cars.stream()
                    .map( (Object car)-> car.toString() )
                    .collect( Collectors.joining() );
```

```
Object result = cars.stream()
                    .map( (Object car)-> car.toString() )
                    .collect( Collectors.joining(", ") );
```


- `collect(Collector)`: *terminal* operation

```
List<String> collected = Stream.of("u", "m", "b")
                               .map((String str)-> str.toUpperCase())
                               .collect( Collectors.toList() );
// a list of "U", "M" and "B" is returned.
```

- `Collectors`: Accumulates values in a stream and transforms the accumulated ones into a particular type
  - `Collectors.toList()`
  - `Collectors.toSet()`
  - `Collectors.toMap(...)`
  - `Collectors.toCollection(...)`
    - Can state a specific collection type/class.
    - `Collectors.toCollection(ArrayList::new)`
    - `Collectors.toCollection(TreeSet::new)`
    - `Collectors.toCollection(TreeMap::new)`

- `toMap(Function<T,R>,Function<T,U>)`
  - Transforms a stream of `T` to a `Map<R,U>`

```
Map<Integer, String> idToAutoMaker =
    cars.stream()
        .collect( Collectors.toMap( (Car car)-> car.getId(),
                                    (Car car)-> car.getMake() );
```

  - Transforms a stream of `Car`s to a `Map<Integer,String>`

```
Map<Integer, Person> idToCar =
    cars.stream()
        .collect( Collectors.toMap( (Car car)->car.getId(),
                                    (Car car)->Function.identity(car) );
```

  - Transforms a stream of `Car`s to a `Map<Integer,Person>`

HW 3-1

- Implement your own min(), max() and count() with reduce() for a stream of Car objects.
  -
```
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;} );
```


HW 3-2

- Compute the average car price with reduce().
- Use the 3rd version of reduce().
- Due: Oct 4 (Thu) midnight

-
```
int average =
  cars.stream()
      .map( (Car car)-> car.getPrice() )
      .reduce( new int[3],
               (arr, price)->{ ...
                               ...
                               return arr; },
               (arr, intermediateArr)->{ return arr; }
             )[2];
```

- int[3]:
  - 0th element: Total number of cars that have been examined so far.
  - 1st element: Sum of car prices that have been examined so far.
  - 2nd element: Average of car prices that have been examined so far.