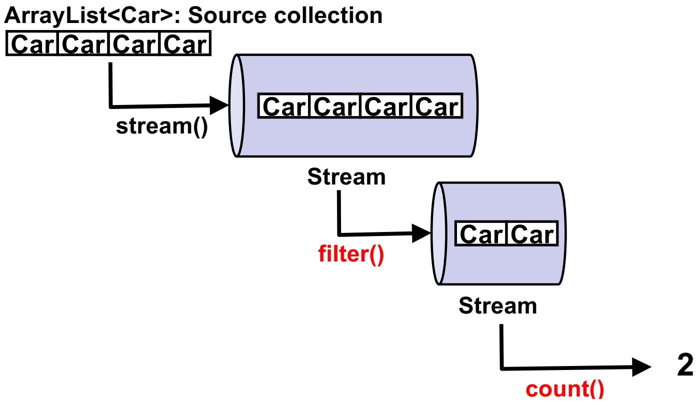
Recap: Stream Pipeline

- Multiple streams can be pipelined.

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

- Streams do NOT modify their source collection.



ArrayList<Car>: Source collection

Car Car Car Car

stream()

Car Car Car Car

Stream

filter()

Car Car

Stream

count()

2

- 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();
```

Important Methods of Stream

- `of(T... values)`: a static method of `Stream`
  - Builds a stream with values (not a collection) as source data
    - `T... values` is a syntactic sugar for `T[] 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.


– 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 applying the provided `Supplier` repeatedly.

```
– 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 applying the provided `UnaryOperator` to the initial element `seed` iteratively.
    - 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 2 odd numbers are removed: 5, 7...
```

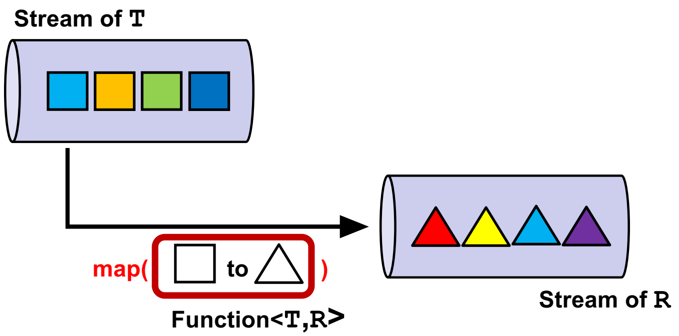
	Params	Returns	Example use case
UnaryOperator<T>	T	T	Logical NOT (!)

- `map(Function<T,R>)` : *intermediate* operation
  - Performs a stream-to-stream transformation
    - Takes a `Function` 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.

```
• 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)

	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)



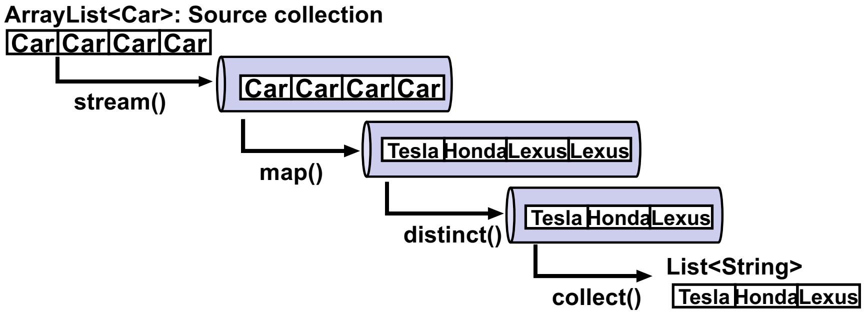
- `collect(Collector)`: *terminal* operation
  - Collects a set of elements from a stream and returns that with a particular collection type.

```
– List<String> collected = Stream.of(“u”, ”m”, ”b”)
                                 .map((String str)-> str.toUpperCase())
                                 .collect( Collectors.toList() );
```

  - `Collectors`: provides static factory methods that return `Collector` objects.
    - `Collectors.toList()`
      - Returns a `Collector` object that collects stream elements and transforms them to a list.
    - `Collectors.toSet()`
    - `Collectors.toMap()`
    - `Collectors.toCollection(...)`
      - Can state a specific collection class.
      - `Collectors.toCollection(ArrayList::new)`
      - `Collectors.toCollection(TreeSet::new)`
      - `Collectors.toCollection(TreeMap::new)`

- `distinct()`: *intermediate* operation
  - Removes redundant elements and returns a stream consisting of distinct elements

```
– List<String> makes = cars.stream()
                           .map( (Car car)-> car.getMake() )
                           .distinct()
                           .collect(Collectors.toList());
```



- `sorted(Comparator)`: *intermediate* operation
  - Sorts stream elements according to the provided `Comparator` and returns the sorted stream.

```
– List<Float> prices =
    cars.stream()
        .map( (Car car)-> car.getPrice() )
        .distinct()
        .sorted( (Car o1, Car o2)->(int)o1.getPrice()-o2.getPrice() )
        .collect( Collectors.toList() );
```

- `sorted(Comparator)`: *intermediate* operation
  - Sorts stream elements according to the provided `Comparator` and returns the sorted stream.

```
– List<Float> prices =
    cars.stream()
        .map( (Car car)-> car.getPrice() )
        .distinct()
        .sorted( (Car o1, Car o2)->(int)o1.getPrice()-o2.getPrice() )
        .collect( Collectors.toList() );

– List<Float> prices =
    cars.stream()
        .map( (Car car)-> car.getPrice() )
        .distinct()
        .sorted( Comparator.comparing((Car car)-> car.getPrice()) )
        .collect( Collectors.toList() );
```

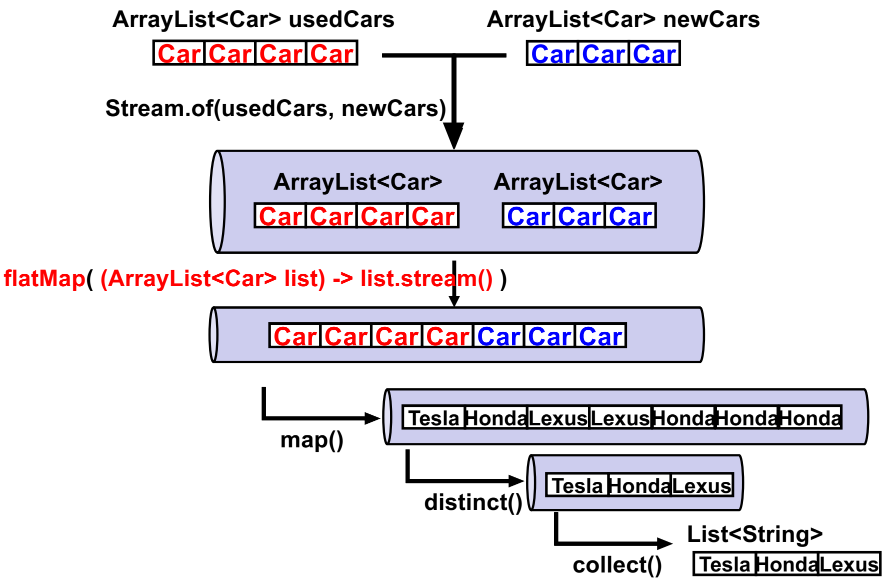
  - `comparing()`: higher-order function; c.f. CS680

- `concat(Stream<T> a, Stream<T> b)`: *intermediate* operation
  - Concatenates two streams into a single stream.

```
- ArrayList<Car> usedCars = ...
  ArrayList<Car> newCars = ...
  Stream<Car> cars =
                 Stream.concat(usedCars.stream(), newCars.stream());
```

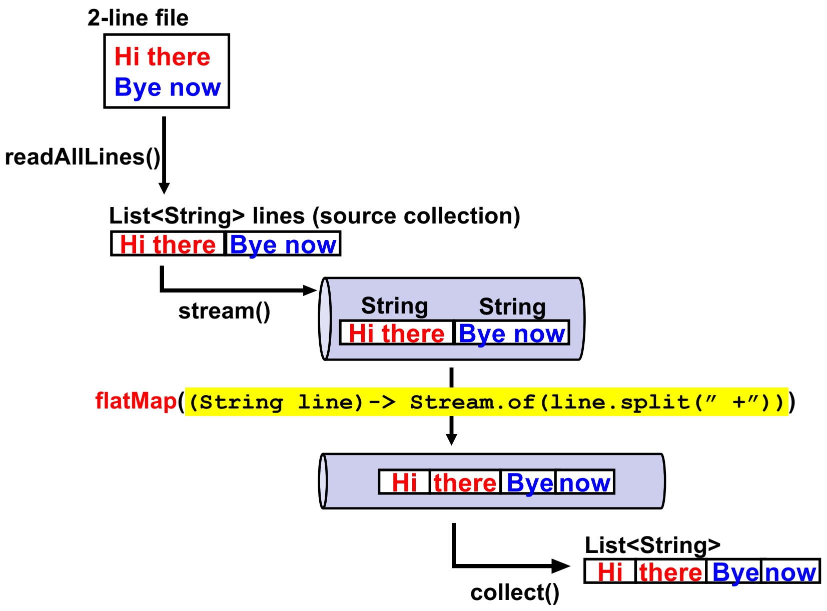
- `flatMap(Function<T,R>)`: *intermediate* operation
  - Converts each element of a steam to a separate steam and then…
  - Concatenates all the converted streams into a single stream.
  - `R` must be a stream.

```
- ArrayList<Car> usedCars = ...
  ArrayList<Car> newCars = ...
  List<String> makes =
              Stream.of(usedCars, newCars)
                    .flatmap( (ArrayList<Car> list)-> list.stream() )
                    .map( (Car car)-> car.getMake() )
                    .distinct()
                    .collect(Collectors.toList());
```



```
- Path path = Paths.get("/Users/jxs/temp/test.txt");
  List<String> lines = Files.readAllLines(path);
  List<String> words =
        lines.stream()
             .flatMap( (String line)-> Stream.of(line.split(" +")) );
             .collect(Collectors.toList());
```

- New I/O (NIO) : `java.nio`
  - Includes `Path`, `Paths`, `Files`, etc. c.f. CS680
- `String.split()`
  - takes a regular expression and returns an array of Strings.
  - `" +"` means splitting a string (`line`) with a space.
    - e.g., "Hi there" → ["Hi", "there"]



2-line file

Hi there
Bye now

readAllLines()

List<String> lines (source collection)

Hi there | Bye now

stream()

String String

Hi there | Bye now

flatMap((String line)-> Stream.of(line.split(" +")))

Hi | there | Bye | now

collect()

List<String>

Hi | there | Bye | now

- `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( price -> price<5000 )
                .max( Comparator.comparing( 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( price -> price<5000 )
          .max( Comparator.comparing( 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., Use the class Car that you implemented in CS680.

- Explore methods of Stream.

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 );
    ```