Building a Stream

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

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


Important Methods to Build a Stream

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

Important Methods to Build a Stream

- `of(T... values)`: a static method of `Stream`
  - Builds a stream with values (not a collection) as source data
    - `T`: Type of Stream elements (same as `T` in `Stream<T>`)
    - `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.
```


- `concat(Stream<T> a, Stream<T> b)`: a static method of `Stream`
  - Concatenates two streams into a single (new) stream.

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

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

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

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

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

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

```
Stream<Double> randomNums = Stream.generate( ()-> Math.random() )
                                  .limit(100);
// First 100 random numbers are selected 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 a given `UnaryOperator` to the initial element `seed` iteratively.
    - Generated elements: seed, f(seed), f(f(seed)), …

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

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

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

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

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

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

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

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

```
Integer seed = 0;
LinkedList<Integer> integers = new LinkedList<>();
integers.add(seed);
Integer i = seed;
for(;;){
  i = i+1;
  integers.add(i);
}
```

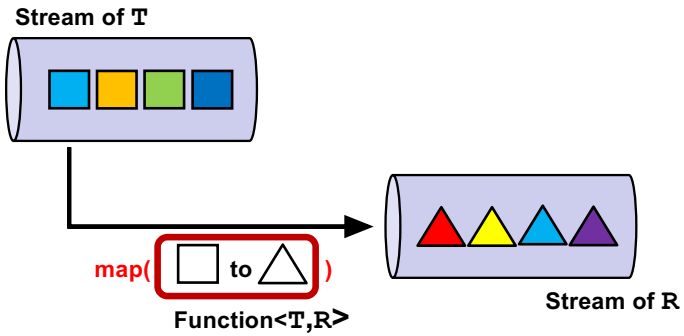

Other Important Methods in Stream

- `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 does 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.
```

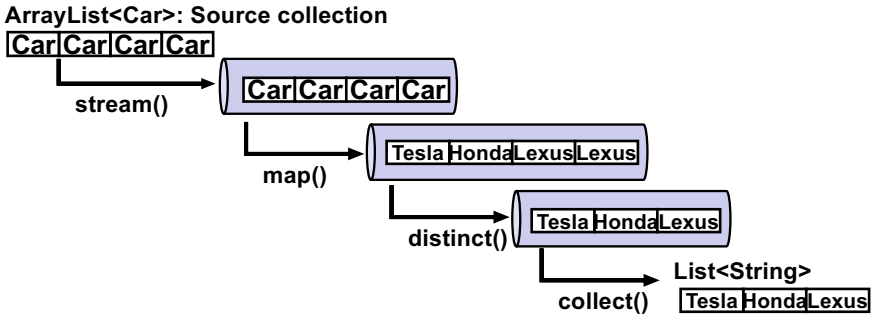
	Params	Returns	Example use case
Function<T,R>	T	R	Get the price (R) from a Car object (T)

	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)



- `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 a given `Comparator` and returns the sorted stream.

```
List<Float> prices =
  cars.stream()
      .map( (Car car)-> car.getPrice() )
      .distinct()
      .sorted( (Float price1, Float price2)-> price1-price2 )
      .collect( Collectors.toList() );
```


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

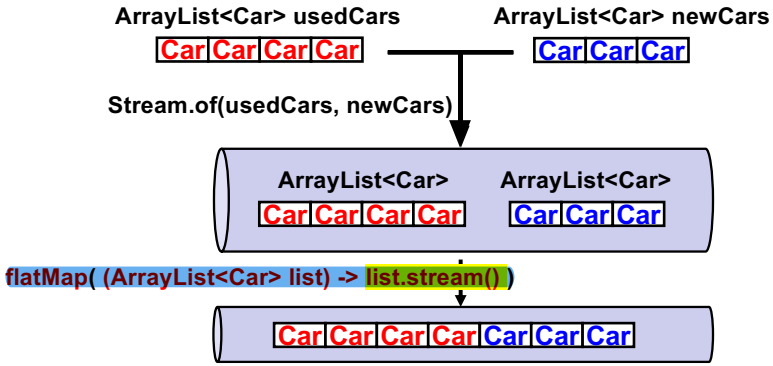
```
List<Car> sortedCars =
  cars.stream()
      .sorted( (Car car1, Car car2)->
                          car1.getPrice()-car2.getPrice()) )
      .collect( Collectors.toList() );
```

```
List<Car> sortedCars =
  cars.stream()
      .sorted( Comparator.comparing((Car car)-> car.getPrice()) )
      .collect( Collectors.toList() );
```

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

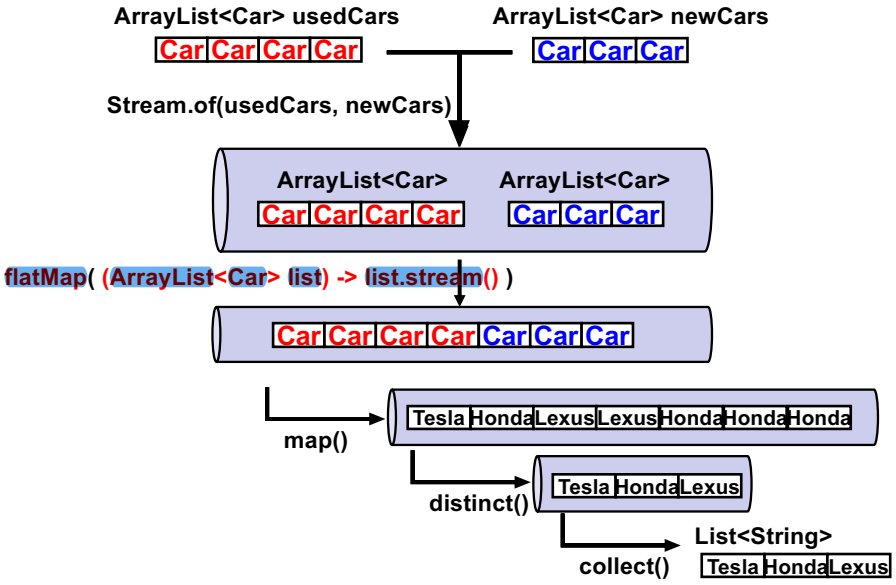
- `flatMap(Function<T,R>)`: *intermediate* operation
  - Accepts a `Function` that converts each element of a steam to a separate steam
  - Concatenates all the converted streams into a single stream.
  - `R` must be a stream.

```
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());
```




```
ArrayList<Car> usedCars = ...
ArrayList<Car> newCars = ...
List<String> makes =
            Stream.of(usedCars, newCars)
                  .flatmap( (ArrayList<Car> list)-> list.stream() )
```

17

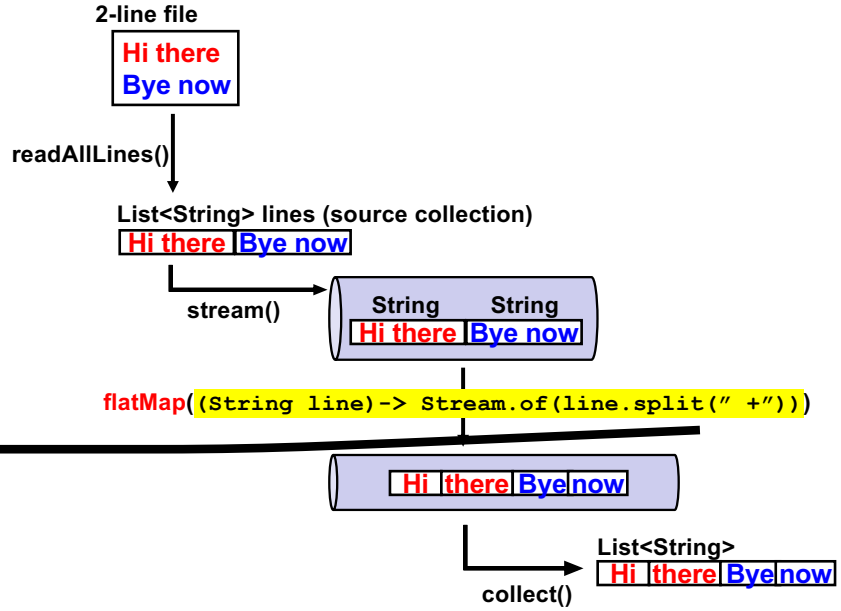


18

```
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 slides
- `String.split()`
  - takes a regular expression and returns an array of Strings.
  - `" +"` means splitting a string (`line`) with a space as a delimiter.
    - e.g., "Hi there" → ["Hi", "there"]

19



20

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

```
Car searchResult =
    cars.stream()
        .filter( (Car car)-> !car.hadAccidents() )
        .min( Comparator.comparing( (Car car)->car.getPrice() ))
        .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() ); }
```


- `collect(Collector)`: *terminal* operation
  - Collects a set of elements from a stream and returns the data set 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`: provides various static factory methods that return `Collector` objects.
    - `Collectors.toList()`
      - Returns a `Collector` object that collects stream elements and transforms them to a `List`.

- `Collectors`: provides various 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( Function<T,R>, Function<T,U> )`
    - A returned `Collector` 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>`

- `Collectors.toMap(Function<T,R>,Function<T,U>)`
  - A returned `Collector` transforms a stream of `T` to a `Map<R,U>`
  - ```
    Map<Integer, Car> idToCar =
        cars.stream()
            .collect(Collectors.toMap(
                                (Car car)->car.getId(),
                                (Car car)->car)) );
    ```
  - Transforms a stream of `Car`s to a `Map<Integer,Car>`

- **`Collectors`**: provides various static factory methods that return `Collector` objects.
  - `Collectors.toList()`
  - `Collectors.toSet()`
  - `Collectors.toMap()`
  - `Collectors.toCollection(...)`
    - Can state a specific collection class.
    - `Collectors.toCollection(ArrayList::new)`
    - `Collectors.toCollection(LinkedList::new)`
    - `Collectors.toCollection(HashMap::new)`
    - `Collectors.toCollection(TreeMap::new)`


- ```
  String concat = cars.stream()
                      .map( (Car car)-> car.getId() )
                      .collect( Collectors.joining(", ") );
  ```
  - `Collectors.joining()`
    - Returns a `Collector` object that concatenates input elements into a `String`, separated by a given delimiter.
- ```
  Map<String, List<Car>> carsGroupedByMakers =
          cars.stream()
              .collect(Collectors.groupingBy(
                          (Car car)->car.getMaker());
  ```
  Here in key value pair we only return LIST of car
- ```
  Map<String, Double> avgPriceByMakers =
          cars.stream()
              .collect(Collectors.groupingBy(
                          (Car car)->car.getMaker(),
                          Collectors.averagingDouble(
                              (Car car)->car.getPrice());
  ```
  If you want values in another type eg. double we use the 2nd part
  - `Collectors.groupingBy()`
    - Returns a `Collector` object that groups input elements into a ~~`String`, separated by a given delimiter.~~ Based on a given criterion.


- ```
  Map<Boolean, List<Car>> carsGroupedByPriceThreshold =
          cars.stream()
              .collect(Collectors.partitioningBy(
                          (Car car)->car.getPrice()>5000);
  ```
- ```
  Map<~~String~~, Map<Integer,Car>> avgPriceByMakers =
          cars.stream()
              .collect(Collectors.~~groupingBy~~partitioningBy(
                          (Car car)->car.getPrice()>5000,
                          Collectors.toMap(
                              (Car car)->car.getId(),
                              (Car car)->car);
  ```
  Boolean
  Map contains mapss
  Boolean. T — Maps Map Key-Values
  F — Map Key-values
  - `Collectors.partitioningBy()`
    - Returns a `Collector` object that separates input elements to two groups with a criterion.

- `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));
```

  - `forEach()` does NOT retain the order of elements,
    - even though the source collection (`cars`) is ordered.
  - All the other methods of `Stream` retains the order of elements, if the source collection is ordered.

API Methods that Return Streams

- Since 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 files and directories.
    - 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 postsCount =
    lines.filter( (String line)-> line.contains("POST") )
         .count();
}
```

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

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 R" means any sub type (subclass) of R.

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


HW 1

- Recall a HW in CS680: Sorting `Car`s with `Collections.sort()` and 4 LEs.
  - 4 LEs implemented 4 different ordering policies: price-, year-, mileage-, and domination count-based sorting.
- Do the same (i.e., sort `Car`s) with Stream API.
  - Implement 4 different sorting policies
- In each sorting policy,
  - Separate `Car`s to the "HIGH" and "LOW" groups with a certain threshold.
    - You can set any threshold for each soring policy.
  - For each group,
    - Get average, highest and lowest values (e.g. prices).
    - Get the the number of cars