General-Purpose Functional Interfaces

- Java 8 has a lot of functional interfaces.
  - e.g., Comparator<T>: Used for specific purposes such as calling Collections.sort().
- In addition to *special-purpose* functional interfaces, Java 8 has *general-purpose* ones that can be 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 Handling/Processing with Streams

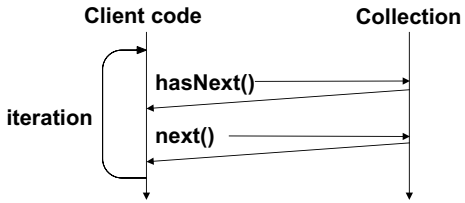
- Java 8 made major improvements to the Collections API by
  - Adding new static and default methods in existing interfaces
    - e.g., `Iterable.forEach()`
  - Adding streams.
    - `java.util.stream.Stream<T>`
      - Contains many methods that take care of common operations to be performed on a `Collection`.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

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*.
  - "How" is often emphasized than "what." Or, "what" is often obscured by "how."

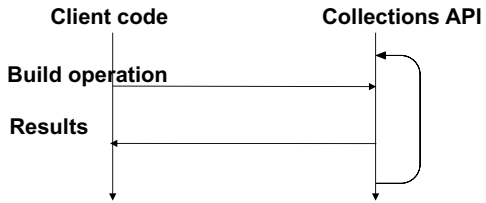
```
int count = 0;
Iterator<Car> it = carList.iterator();
while( iterator.hasNext() ){
    Car car = iterator.next();
    if( car.getPrice() < 5000 ) count++; }
```

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

New Way of Collection Access

- ***Internal*** iteration:
  - `stream()`: Performs 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 exists 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; Keeping only `Car` objects whose prices are lower than $5000
  - Count the number of `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();
```

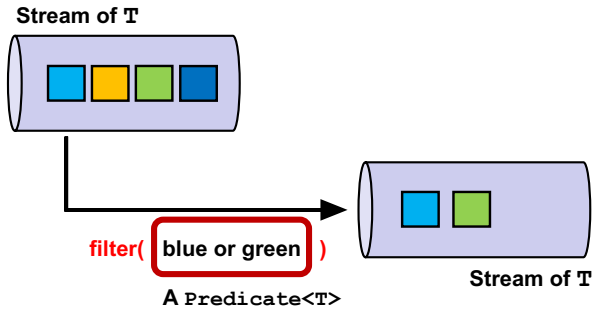
Streams and Collections

- Interface `Collection<E>`
  - `default Stream<E> stream()`
    - Returns a stream 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.

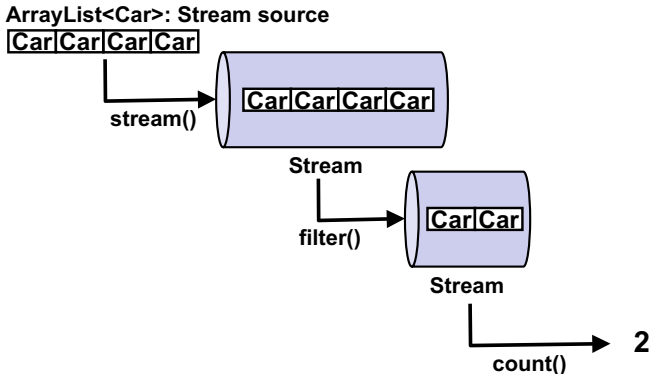
-
```
long count = carList.stream()
                    .filter( (Car car)-> car.getPrice()<5000 )
                    .count();
```


	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.



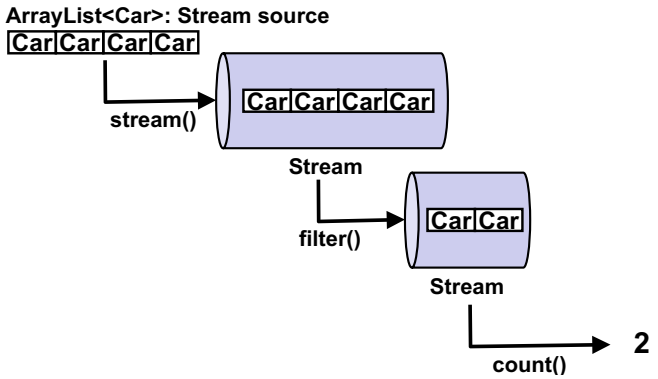
- Common Structure of stream pipelines
  - Build a stream on a collection source
  - Perform zero or more *intermediate* operations
    - An intermediate operation returns a Stream.
  - Perform a *terminal* operation
    - A terminal operation returns non-Stream value or void.

-
```
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 never occur.
- ```
  long count = carList.stream()
                      .filter((Car car)->{
                                System.out.println(car.getPrice());
                                car.getPrice()<5000} );
  ```
  - Nothing is printed out.
  - The filtering operation never occur.
  ```

Exercise

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

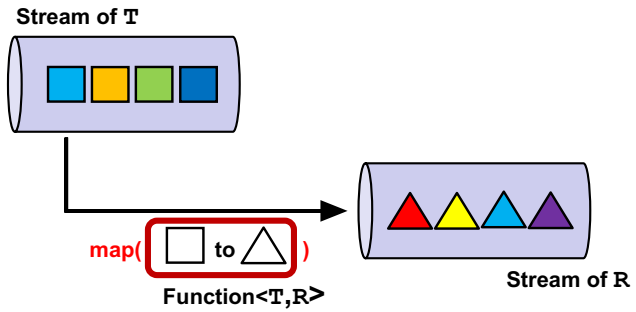
Important Methods of Stream

- `of(T... values)`: a static method
  - Builds a stream with `values` as a stream source

```
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.
```

- `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 a stream of values (one by one).
    - Returns another stream of new values.

	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 (*)

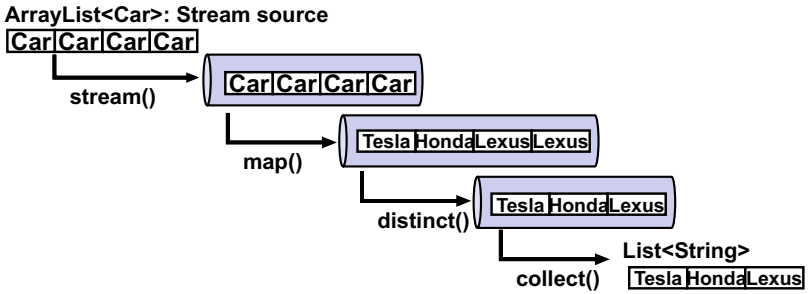


- `collect(Collector)`: *terminal* operation
  - Collects a set of values from a stream and returns it with a particular collection type.

```
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.
```

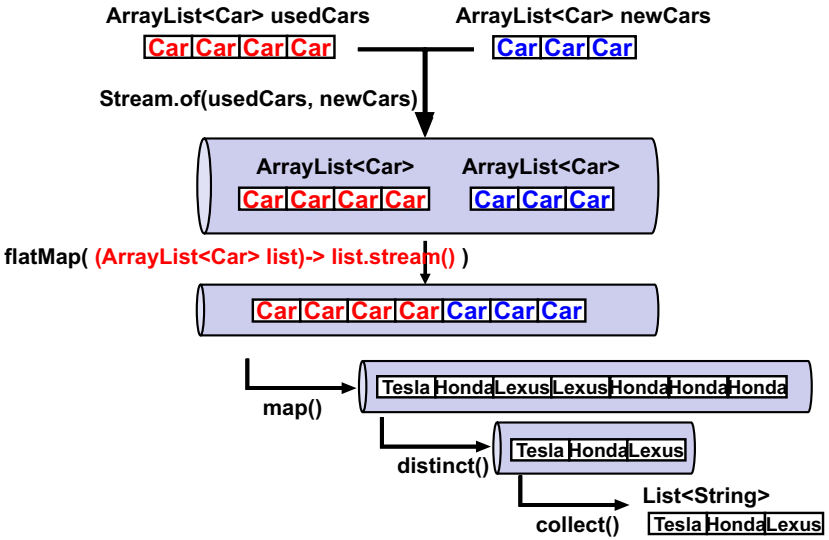
  - `Collectors`: Accumulates values in a stream and transforms the accumulated ones into a particular type
    - `Collectors.toList()`
    - `Collectors.toSet()`
    - `Collectors.toMap()`

- `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



- `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());
```

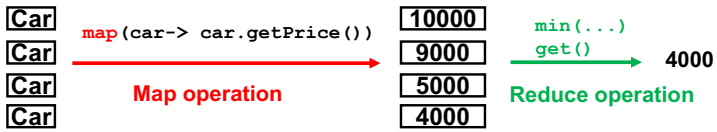
- `max(Comparator<T>)`: *terminal* operation
  - Returns the maximum value according to the provided `Comparator`.
- `min(Comparator<T>)`: *terminal* operation
  - Returns the maximum value according to the provided `Comparator`.
-
```
Integer p = cars.stream()
                .filter( (Car car)-> !car.hadAccidents() )
                .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`.

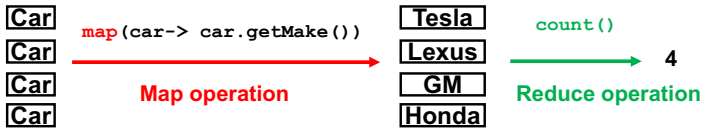
Map-Reduce Data Processing Pattern

- 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 reduce operations for common data processing logic.
  - e.g. `count()`, `max()`, `min()`
- Use `reduce()` when you would like to define your own reduce operation
  - `T reduce(T, BinaryOperator<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)
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.

A Generalized Form of Reduce Operations

- With the Stream API
  - ```
    Integer price = cars.stream()
                    .map((Car car)-> car.getPrice())
                    .min(Comparator.comparing(price-> price ))
                    .get();
    ```
- 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;
    }
    ```
- Generalized form of reduce op in a traditional style
  - ```
    T result = initValue;
    for(T element: collection){
        result = accumulate(result, element);
    }
    ```


- Generalized form of reduce op in a traditional style
  - ```
    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 of the collection through accumulate()
- Reduce operations can be implemented in this form by varying initValue and accumulate().

- Generalized form of reduce op with the Stream API
  - ```
    T result = aStream.reduce(initValue, (result, element)-> ... );
    ```
  - ```
    T result = initValue;
    for(T element: collection){
        result = accumulate(result, element);
    }
    ```
- `T reduce(T, BinaryOperator<T>)`
  - Takes a lambda expression as the second parameter.
    - The body of `accumulate()` is expressed in the LE.

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

- 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;
}
```


HW 3-1

- Implement your own min() and max() 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;} );
```

- Due: March 15 midnight

reduce()

- Use `reduce()` when you would like to define your own reduce operation
  - `T reduce(T, BinaryOperator<T>)`
    -

```
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;} );
```

  - `U reduce(U, BiFunction<U,T>, BinaryOperator<U>)`

	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.

- Think of implementing `count()` yourself with `reduce()`.
  - Example:

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

  - `count()` takes a stream of auto makers and returns the number of them.
    - The input and output use different types (String and long).

A Generalized Form of Reduce Operations

- With the Stream API
  - ```
    long carMakerNum = cars.stream()
                            .map( (Car car)-> car.getMake() )
                            .count();
    ```
- In a traditional style
  - ```
    List<String> carMakers = ...
    int result = 0;
    for(String carMaker: carMakers){
     if(carMaker != null){
            result++;
     }
    }
    ```
- Generalized form of reduce op in a traditional style
  - ```
    U result = initValue;
    for(T element: collection){
        result = accumulate(result, element);
    }
    ```


- Generalized form of reduce op in a traditional style
  - ```
    U result = initValue;
    for(T element: collection){
        result = accumulate(result,element);
    }
    ```
  - result (U)
    - is *initialized* with initValue.
    - is *updated* in each iteration of the loop by
      - Getting accumulated with each element of the collection (T) through accumulate()
        - » accumulate() returns a U value.
- Reduce operations can be implemented in this form by varying initValue and accumulate().

- Generalized form of reduce op with the Stream API
  - ```
    U result = aStream.reduce(initValue,
                              (result, element)-> ... );
                              (result, intermediateR)->... );
    ```
  - ```
    U result = initValue;
    for(T element: collection){
        result = accumulate(result,element);
    }
    ```
- `U reduce(U, BiFunction<U,T>, BinaryOperator<U>)`
  - Takes lambda expressions as the second and third parameters.
    - The body of `accumulate()` is expressed as the second param.
    - The third param: Just return result.
      - More details to be explained in the rest of this semester.

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

- With `count()` in the Streams API
  - ```
    long carMakerNum = cars.stream()
                            .map( (Car car)-> car.getMake() )
                            .count();
    ```
- With `reduce()` in the Streams API
  - ```
    int carMakerNum = cars.stream()
                           .map( (Car car)-> car.getMake() )
                           .reduce(0,
                                   (result,carMaker)-> ++result
                                   (result,intermediateR)->result);
    ```
- In traditional style
  - ```
    List<String> carMakers = ...
    int result = 0;
    for(String carMaker: carMakers){
     if(carMaker != null){
            result++;
     }
    }
    ```
    ```

- With `reduce()` in the Stream API
  -
```
int carMakerNum = cars.stream()
                      .map( (Car car)-> car.getMake() )
                      .reduce(0,
                              (result,carMaker)-> ++result
                              (result,intermediateR)->result);
```

- `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
```


HW 3-2

- Implement count() with reduce() for a stream of Car objects.
- Compute the average car price with reduce().
- Use the second variant of reduce().
- Due: March 15 midnight

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