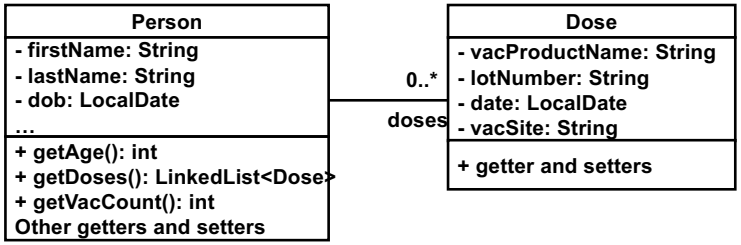
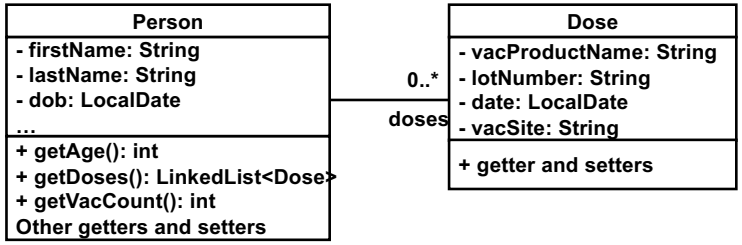
Exercise: Calculating Vaccination Rate

- Suppose you have a dataset about vaccination status in a certain community (e.g. town, college)
  - The dataset (in a DB or as a CSV file) has been parsed into a `LinkedList<Person>`.
  - Each `Person` has a `LinkedList<Dose>`.



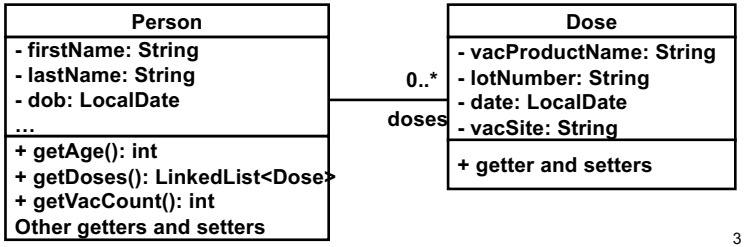
- Compute the vaccination rate of this community (`LinkedList<Person>`).
  - Rate = # of fully vaccinated 18+ yr-olds / total headcount

```
LinkedList<Person> people = ...;
long fullyVacCount = people.stream()
                           .filter( ... )
                           .count();
float vacRate = (float)fullyVacCount/people.size() * 100;
```



- Compute the vaccination rate of this community (`LinkedList<Person>`).
  - Rate = # of fully vaccinated 18+ yr-olds / total headcount

```
LinkedList<Person> people = ...;
long fullyVacCount = people.stream()
                           .filter((Person p)->
                                   {p.getAge()>=18 &&
                                    p.getVacCount()>=3})
                           .count();
float vacRate = (float)fullyVacCount/people.size() * 100;
```



Further Exercises

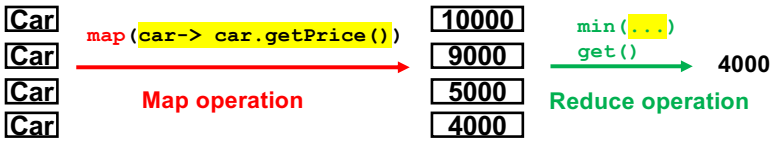
- Calculate the vaccination rate for
  - The people over 60 yrs old.
  - The people in between 40 and 60 yrs old.
  - The people in between 20 and 40 yrs old.
  - The people under 20 yrs old.
- Calculate the average # of vaccinations administered in each of the above age groups.
- Calculate the number of people who have never been vaccinated in each of the above age groups.

Map-Reduce Data Processing Pattern

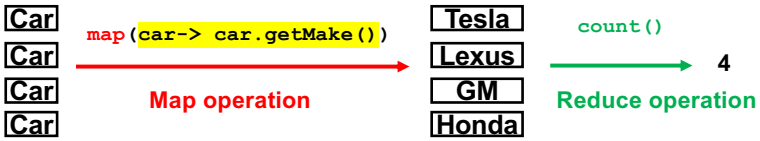
Map-Reduce Data Processing Pattern

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


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

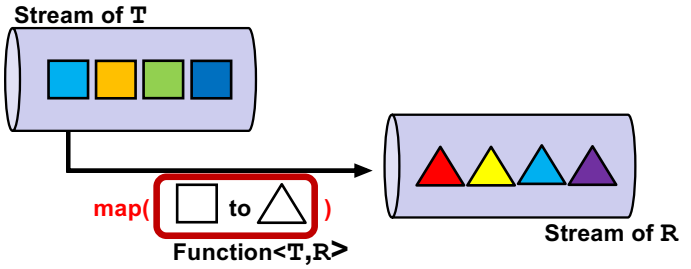



```
long carMakerNum = cars.stream()
                       .map( (Car car)-> car.getMake() )
                       .count();
```



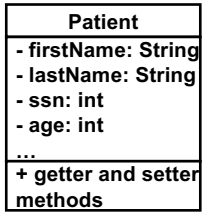
`Stream.map()`

- `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.

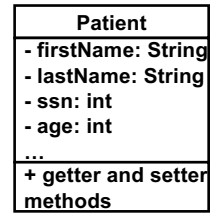


Exercise: Data Anonymization

- Suppose you have a (huge) dataset about patients with a certain disease and process it for a clinical/research study.
  - The original dataset (in a DB or as a CSV file) has been parsed into a `LinkedList<Patient>`.

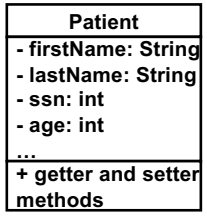


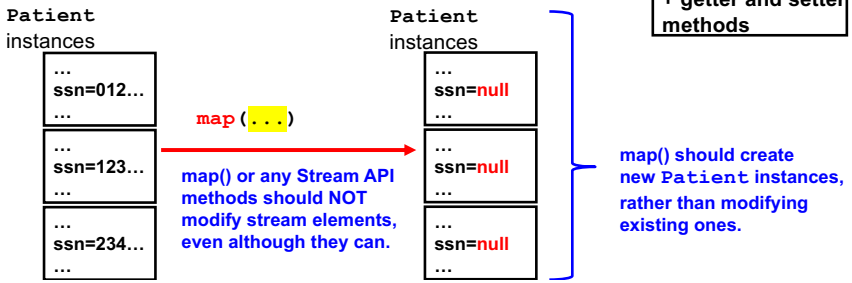
- Data processing for anonymization
  - Randomly choose a half of the patients as samples
    - e.g., Reducing the number of patients from 10K to 5K
  - Replace the first and last names and SSN of each patient with "null"
  - Replace the age of each patient with:
    - 50 if age >= 50
    - 21 if 21 <= age < 50
    - 0 if age < 21



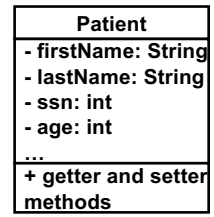

```
LinkedList<Patient> patients = ...;
LinkedList<Patient> patientSamples =
  patients.stream()
          .filter( (patient)->{Math.random()>0.5;} )
          .map(...)
          .collect(Collectors.toCollection(...));
```

- Data processing for anonymization
  - Randomly choose a half of the patients as samples
    - e.g., Reducing the number of patients from 10K to 5K
  - Replace the first and last names and SSN of each patient with "null"
  - Replace the age of each patient with:
    - 50 if age >= 50
    - 21 if 21 <= age < 50
    - 0 if age < 21





- Data processing for anonymization
  - Randomly choose a half of the patients as samples
    - e.g., Reducing the number of patients from 10K to 5K
  - Replace the first and last names and SSN of each patient with "null"
  - Replace the age of each patient with:
    - 50 if age >= 50
    - 21 if 21 <= age < 50
    - 0 if age < 21

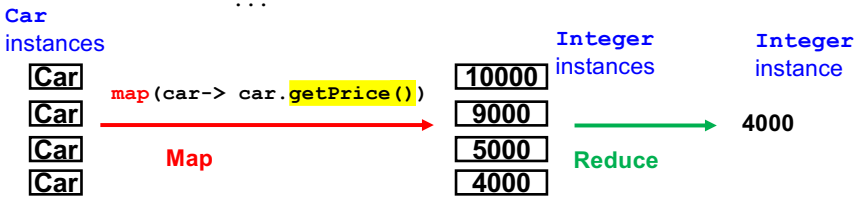



```
LinkedList<Patient> patients = ...;
LinkedList<Patient> patientSamples =
  patients.stream()
          .filter( (patient)->{Math.random()>0.5;} )
          .map((patient)->{if(patient.getAge()>=50){
                    new Patient(null, null, null, 50,...); }
                  else ...)
          .collect(Collectors.toCollection(...));
```

Auto-boxing and Auto-unboxing

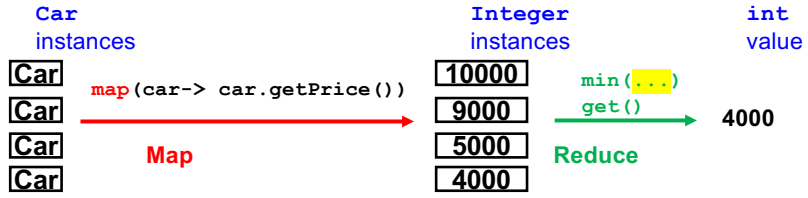
- Stream API performs auto-boxing and auto-unboxing whenever necessary and appropriate.

```
Integer price = cars.stream()
                .map( (Car car)-> car.getPrice() )
                ...
```



  - Even though `getPrice()` returns an `int` value, `map()` auto-boxes it to an `Integer` instance and creates a stream of `Integer` instances.
    - Reference types are used for stream elements, by default.


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



  - Even though `get()` return an `Interger` instance, it is auto-unboxed to an `int` value.

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

  - Even though `random()` returns a `double` value, `generate()` auto-boxes it to a `Double` instance and creates a stream of `Double` instances.
    - Reference types are assumed for stream elements, by default.

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

Stream of Native-type Values

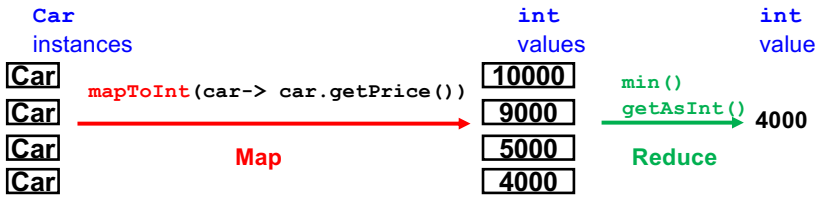
- You can create and use a stream of native-type values, if you want,
  - rather than a stream of class instances.

  - `IntStream` for a stream of `int` values
  - `LongStream` for a stream of `long` values
  - `DoubleStream` for a stream of `double` values

  - These special stream types implement useful arithmetic operations.
    - sum(), average()
    - min(), max(), sorted(): Does not take a Comparator.
    - summaryStatistics(): Returns a stats summary.

- Call `mapToInt()`, `mapToLong()` or `mapToDouble()` to get a stream of native-type values (i.e., unboxed values)

```
int minPrice = cars.stream()
                   .mapToInt( (Car car)-> car.getPrice() )
                   .min()
                   .getAsInt();
```



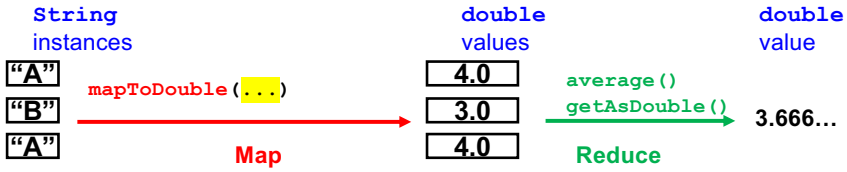
Exercise: GPA Calculation

```
HashMap<String, String> transcript = new HashMap<>();
transcript.put("CS680", "A");
transcript.put("CS681", "B");
transcript.put("CS682", "A");
```



```
HashMap<String, String> transcript = new HashMap<>();
transcript.put("CS680", "A");
transcript.put("CS681", "B");
transcript.put("CS682", "A");

double gpa = transcript.values()
                .stream()
                .mapToDouble((String grade)->{
                        if(grade.equals("A")){return 4.0;}
                        else if(...){...}
                        else if ... })
                .average()
                .getAsDouble();
```



Further Exercises

- Calculate the number of each grade.
- Calculate the GPA for undergraduate courses and graduate courses

Exercise: Order Queries

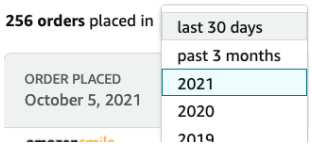
- Suppose an `Account` manages a list of `Orders`.
- How to write `Account`'s methods with Stream API?
  - `getOrdersInYr(int year)`

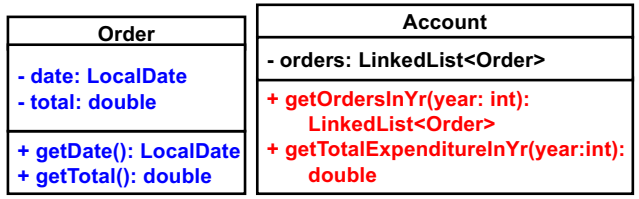
```
return orders.stream()
             .filter( ... )
             .collect(Collectors.toCollection(LinkedList::new));
```

  - `getTotalExpenditureInYr(int year)`

```
return this.getOrdersInYr(year).stream()
                               .mapToDouble( ... )
                               .sum();
```

- FYI:
  - `LocalDate today = LocalDate.now();`
  - `int year = today.getYear();`

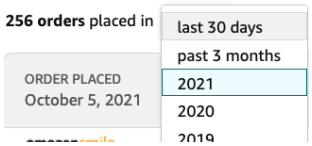


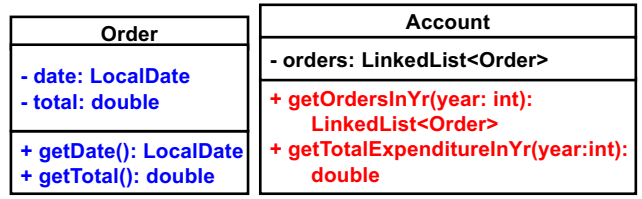



```
getOrdersInYr(int year):
  return orders.stream()
               .filter( (order)-> order.getDate().getYear()==year )
               .collect(Collectors.toCollection(LinkedList::new));

getTotalExpenditureInYr(int year):
  return this.getOrdersInYr(year).stream()
                                 .mapToDouble((order)-> order.getTotal())
                                 .sum();
```


- FYI:
  - `LocalDate today = LocalDate.now();`
  - `int year = today.getYear();`

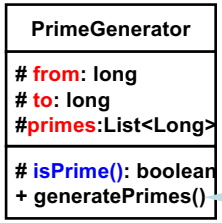




Further Exercise

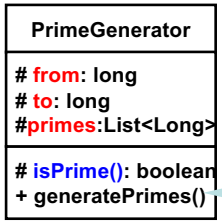
- Calculate the average total expenditure in each order.
- Calculate the total expenditure in each month over multiple years.

Exercise: Prime Number Generation



Without Stream API (c.f. CS680)

```
for(long n = from; n <= to; n++) {
        if( isPrime(n) ){ primes.add(n); } }
```



With Stream API

```
primes = LongStream.rangeClosed(from, to)
                   .filter((long n)->isPrime(n))
                   ...}
```

LongStream

- A stream of primitive `long` values
  - A stream that deals with `long` values directly, rather than `Long`
  - `range(long startInclusive, long endExclusive)`
    - Create a stream from `startInclusive` (inclusive) to `endExclusive` (exclusive) by an incremental step of 1.
  - `rangeClosed(long startInclusive, long endInclusive)`
    - Create a stream from `startInclusive` (inclusive) to `endInclusive` (inclusive) by an incremental step of 1.
- c.f. `DoubleStream` and `IntStream`

- `of()`, `generate()`, `iterate()`
  - Creates a `LongStream`
  - c.f. previous lecture note
- `sum()`, `average()`, `min()`, `max()`, `sorted()`, etc.
  - Useful numerical computation
- `boxed()`
  - Converts (or "box") `long` numbers (primitive values) to `Long` instances and
  - Returns a `Stream<Long>`
- c.f. `DoubleStream` and `IntStream`