Reduce Operation

- `Steam` provides "ready-made" reduce operations for common data processing tasks.
  - e.g. `count()`, `max()`, `min()`, `sum()`, `average()`

- Use `reduce()` if you implement your own (custom) 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)`

1st Version of `reduce()`

- `Optional<T> reduce(BinaryOperator<T> accumulator)`
  - Takes a reduction function (`accumulator`) as a LE.
    - Applies it on stream elements (`T`) one by one.
  - Returns the reduced value (`T`).

  - `T result = aStream.reduce( (T result, T elem)-> {...} ).get();`
    - » `result`: result holder
    - » `elem`: next available element

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

1st Version of `reduce()`

- `Optional<T> reduce(BinaryOperator<T> accumulator)`
  - Takes a reduction function (`accumulator`) as a LE.
    - Applies it on stream elements (`T`) one by one.
  - Returns the reduced value (`T`).

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

  - ```
    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);
    }
    ```

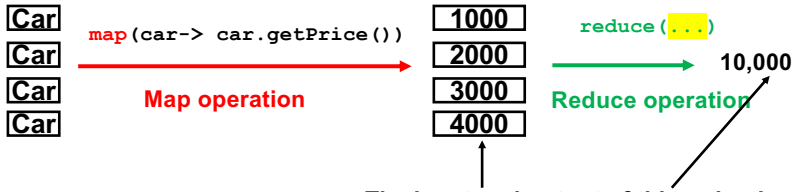

- `T result = aStream.reduce( (T result, T elem)-> {...} ).get();`

- ```
  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` : *result holder*
  - is *initialized* with the first element.
  - is *updated* in each iteration of the loop by
    - Getting accumulated with the next element (`elem`) with `accumulate()`

- A reduce operation (`accumulator`) is implemented as an anonymous version of `accumulate()`.
  - `accumulator`'s code block == `accumulate()`'s method body

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



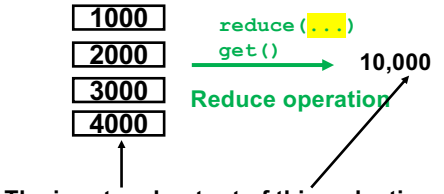
result = 1000
result = result + 2000 // 3,000
result = result + 3000 // 6,000
result = result + 4000 // 10,000

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

Important Notes (1)

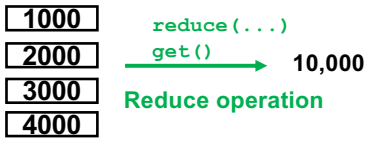
```
LinkedList<Car> cars = ...;
Integer totalValue
  = cars.stream()
        .map( (Car car)-> car.getPrice() )
        .reduce((Integer result, Integer price)->{result+price})
        .get();
```

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



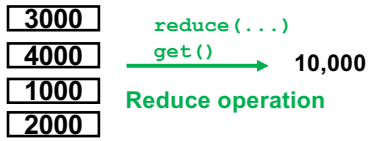
Important Notes (2)

- The order of applying a LE on stream elements is **NOT** guaranteed.
  - Even though stream elements are ordered.
    - A stream's elements are ordered if its source collection is ordered (e.g., `List`).
- A reduction function 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)
  - Gives the same result regardless of the way the operands are grouped.
  - e.g., Numerical sum, numerical product, string concatenation, max, min, matrix product, set union, set intersection, logical AND, logical OR, logical XOR, etc.
- Non-associative operators
  - e.g., Numerical subtraction, numerical division, power, logical NOR, logical NAND, etc.
    - (10 - 5) - 2 = 3 V.S. 10 - (5 - 2) = 7
    - (10/5)/2 = 1 V.S. 10/(5/2) = 4
    - (10^5)^2 V.S. 10^(5^2)

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 (`accumulator`) as the second parameter.
    - Applies the function on stream elements (`T`) one by one.
  - Returns the reduced value (`T`).

- `T result = aStream.reduce(T initValue, (T result, T elem)-> {...});`
  - » `result`: result holder
  - » `elem`: next available element

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

- `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 (`accumulator`) as the second parameter.
    - Applies the function on stream elements (`T`) one by one.
  - Returns the reduced value (`T`).

- `T result = aStream.reduce(T 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 (*)

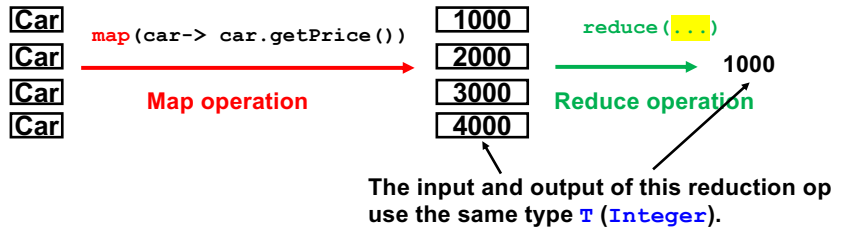
- `T result = aStream.reduce(T initValue, (T result, T elem)-> {...});`

- `T result = initValue;`
  `for(T element: collection){`
  `  result = accumulate(result, element);`
  `}`

- `result` : *result holder*
  - is *initialized* with `initValue`.
  - is *updated* in each iteration of the loop by
    - Getting accumulated with the next element (`elem`) with `accumulate()`

- A reduce operation (`accumulator`) is implemented as an anonymous version of `accumulate()`.
  - `accumulator`'s code block == `accumulate()`'s method body

- `Integer minPrice`
  `= 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;
}
```

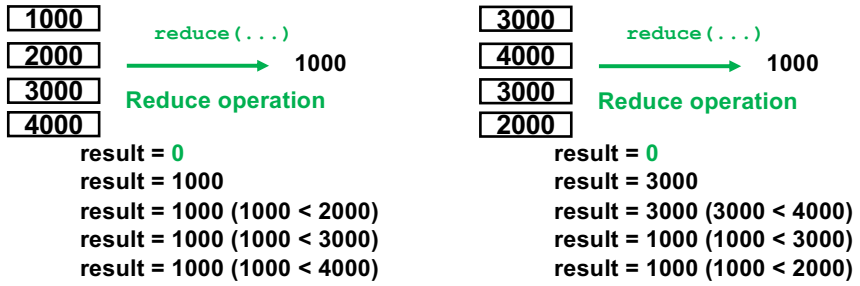
- With `min()` in the Stream API

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


Important Note

- The order of applying a LE on stream elements is **NOT** guaranteed.
  - Even though stream elements are ordered.
    - A stream's elements are ordered if its source collection is ordered (e.g., `List`).
- A reduction operator 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;} );
```



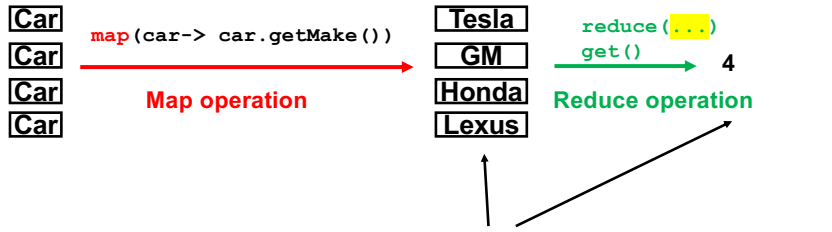
3rd Version of `reduce()`

- `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 (`accumulator`) as the second parameter.
    - Applies the function on stream elements (`T`) one by one.
  - Takes a combination function (`combiner`) as the third parameter.
    - Applies the function on *intermediate* reduction results (`U`) one by one.
  - Returns the final (combined) result (`U`).
  - Useful when stream elements (`T`) and a reduced value (`U`) use different types.

	Params	Returns	Example use case
BiFunction<U,T>	U, T	U	
BinaryOperator<U>	U, U	U	

- 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: T)
Output: # of auto makers (long: U)

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

```
U result = initValue;
for(T element: collection){
    result = accumulate(result,element);
}
```

- `result` : *result holder*
  - is *initialized* with `initValue`.
  - is *updated* in each iteration of the loop by
    - Getting accumulated with the next element (`elem`) with `accumulate()`

- A reduce operation (`accumulator`) is implemented as an anonymous version of `accumulate()`.
  - `accumulator`'s code block == `accumulate()`'s method body

- 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){
        result++;
    }
    long carMakerNum = result;
    ```

- With `count()` in the Streams API
  - ```
    long carMakerNum = cars.stream()
                            .map( (Car car)-> car.getMake() )
                            .count();
    ```


Important Notes

- The order of applying a LE on stream elements is **NOT** guaranteed.
  - Even though stream elements are ordered.
    - A stream's elements are ordered if its source collection is ordered (e.g., `List`).

- Reduction and combination operators must be **associative**.

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

- `reduce()` executes `result = ++result;`

- Just in case, note that:
  - ```
    int i,x,y = 0; i++;          // i==1
    int i,x,y = 0; int x = i++; // i==1, x==0
                                 // assignment of 0 first
    int i,x,y = 0; int y = ++i; // i==1, y==1
                                 // increment of 0 first
    ```
    ```

- Just return `finalResult` (the first parameter) in the second LE unless you use a parallel stream in a multi-threaded app.

```
– long carMakerNum =
                cars.stream()
                    .map( (Car car)-> car.getMake() )
                    .reduce(0,
                            (result,carMaker)-> ++result,
                            (finalResult,intermidiateResult)->finalResult);
```

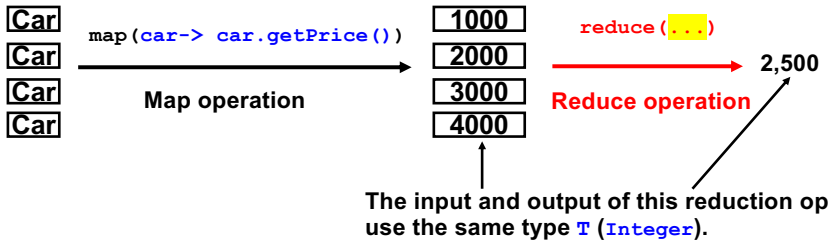
- When using a default (i.e. non-parallel, sequential) stream with a single thread, you never need to do anything extra in the second LE.
  - Just return what's at hand now, which is contained in the first parameter of the second LE, as the final map-reduce result.

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 a custom initial value.
- If the input (stream elements) and output (reduced result) use different types, use the 3rd version:
  - `U  reduce(U initVal, BiFunction<U,T> accumulator, BinaryOperator<U> combiner)`

Exercise: Average Car Price

```
• Integer averagePrice
    = cars.stream()
          .map( car -> car.getPrice() )
          .reduce( ... )
          .get();
```



3 Versions of reduce()

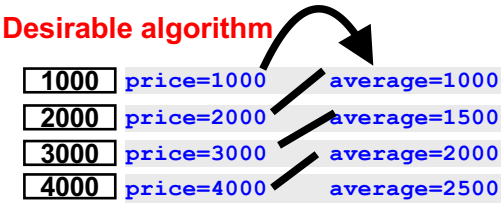
- If the input (stream elements) and output (reduced result) use the same type, use the 1st or 2nd version:
  - `Optional<T>  reduce(BinaryOperator<T> accumulator)`
  - `T  reduce(T initVal, BinaryOperator<T> accumulator)`
  - Use the 2nd version if you need a custom initial value.
- If the input (stream elements) and output (reduced result) use different types, use the 3rd version:
  - `U  reduce(U initVal, BiFunction<U,T> accumulator, BinaryOperator<U> combiner)`

Let's Try to Use the 1st Version of reduce()

```
• Integer averagePrice
    = cars.stream()
          .map( car -> car.getPrice() )
          .reduce( (average, price)-> {...} )
          .get();
```

```
Iterator<Iterator> it = prices.iterator();
Integer average = it.next();// first element
while(it.hasNext()){         // for each remaining elem
    Integer price = it.next();
    average = accumulate(average, price);}
```

Desirable algorithm




```
• Integer averagePrice
    = cars.stream()
          .map( car -> car.getPrice() )
          .reduce( (average, price)->{(average+price)/2} )
          .get();
```

Desirable algorithm

```
1000 price=1000    average=1000
2000 price=2000    average=1500
3000 price=3000    average=2000
4000 price=4000    average=2500
```

```
1000 price=1000    average=1000
2000 price=2000    average=(average+price)/2 = 1500
3000 price=3000    average=(average+price)/2 = 2250
4000 price=4000    average=(average+price)/2 = 3125
```



```
• Integer averagePrice
    = cars.stream()
          .map( car -> car.getPrice() )
          .reduce( (average, price)-> {(average+price)/2} )
          .get();
```

This (yellow-highlighted) LE DOES NOT work.

```
1000 price=1000    average=1000
2000 price=2000    average=(average+price)/2 = 1500
3000 price=3000    average=(average+price)/2 = 2250
4000 price=4000    average=(average+price)/2 = 3125
```

It should have worked like this:

```
1000 price=1000    average=1000
2000 price=2000    average=(average*1+price)/2 = 1500
3000 price=3000    average=(average*2+price)/3 = 2000
4000 price=4000    average=(average*3+price)/4 = 2500
```



```
• Integer averagePrice
    = cars.stream()
          .map( car -> car.getPrice() )
          .reduce( (average, price)-> {...} )
          .get();
```

An algorithm to be implemented:

```
1000 price=1000    average=1000
2000 price=2000    average=(average*1+price)/2 = 1500
3000 price=3000    average=(average*2+price)/3 = 2000
4000 price=4000    average=(average*3+price)/4 = 2500
```

To calculate the average correctly, the lambda expression requires:
(1) the number of the stream elements that have been examined AND
(2) the average of the elements that have been examined

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

1000	price=1000	average=1000
2000	price=2000	average=(average*1+price)/2 = 1500
3000	price=3000	average=(average*2+price)/3 = 2000
4000	price=4000	average=(average*3+price)/4 = 2500

To calculate the average correctly, the lambda expression requires:
(1) the number of the stream elements that have been examined AND
(2) the average of the elements that have been examined

Since `reduce()` takes only one parameter as a result holder, we pack the two data into a single parameter value. The simplest strategy here is to use an array.

The parameter (result holder) to be passed to the LE:

```
int[2]: [# of elements that have been examined,
         the average of those elements]
```

Result holder: array
Type of stream elements: Integer

The 1st version of reduce() uses the same type for the result holder and stream elements.

Cannot use the 1st version of reduce() anymore. In fact, the 2nd version does the same. We need to use the 3rd version.

Let's Use the 3rd Version of reduce()

```
Integer averagePrice
  = cars.stream()
        .map( car -> car.getPrice() )
        .reduce( new int[2],                  // int[2]: [# of elems that have been examined,
                 (result, price)->{...        //          the average of those elems]
                                   ...
                                   return result;}),
                 (finalResult, intermediateResult)->finalResult
                )[1];
```

```
int[] result = new int[2];
for(Integer price: prices){
    result = accumulate(result, price);}
```

1000
2000
3000
4000

reduce(...) → [4, 2500]

Input: `Integer` **Output:** `int[2]`


```
Integer averagePrice
  = cars.stream()
        .map( car -> car.getPrice() )
        .reduce( new int[2],                  // int[2]: [# of elems that have been examined,
                 (result, price)->{...        //          the average of those elems]
                                   ...
                                   return result;}),
                 (finalResult, intermediateResult)->finalResult
                )[1];
```

1000
2000
3000
4000

reduce(...) → [4, 2500]

Input: `Integer` **Output:** `int[2]`

1000 ([0, 0], 1000) → (0*0 + 1000)/(0++)
return [1,1000]

2000 ([1, 1000], 2000) → (1*1000 + 2000)/(1++)
return [2,1500]

3000 ([2, 1500], 3000) → (2*1500 + 3000)/(2++)
return [3,2000]

4000 ([3, 2000], 3000) → (3*2000 + 4000)/(3++)
return [4,2500]

Exercise

- Given a list of **`Car`**s,
  - Find the lowest and highest price.
    - Use the 2nd version of **`reduce()`**.
  - Compute the average price.
    - Use the 3rd version of **`reduce()`**.

HW 2

- A result holder can be a class instance
  - as well as a native-type value and an array.
- Compute the average car price with a class instance as a result holder (not an array as a result holder)
  -
```
public class CarPriceResultHolder {
    private int numCarExamined;
    private double average;
Getter methods ... }
```
  - Your map-reduce code:
    -
```
double averagePrice
  = cars.stream()
        .map( car -> car.getPrice() )
        .reduce( new CarPriceResultHolder(),
                 (result, price)->{...
                                   ...
                                   return result;}),
               (finalResult, intermediateResult)->finalResult
      ).getAverage();
```

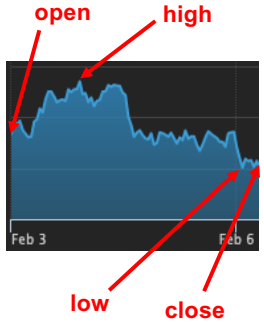

HW 3

- Understand what I explained about *Observer* in the first lecture (c.f. 1st lec note).
  - I explained how you are expected to use the Observable class and the Observer interface that I gave you in CS680.
  - I also explained how many students mis-used them in CS680.
- Download DJIA's daily historical data for the past 5 days and save it as a CSV file.
  - e.g., https://www.wsj.com/market-data/quotes/index/DJIA/historical-prices

DJIA Daily Historical Data

- 4 data are extracted from price changes in a day
  - "Open" price
    - The price when the market opens at 9am
  - "Close" price
    - The price when the market closes at 4pm
  - "High" price
    - The highest price during the day.
  - "Low" price
    - The lowest price during the day.

DATE	OPEN	HIGH	LOW	CLOSE
02/08/23	34132.90	34161.65	33899.79	33949.01
02/07/23	33769.78	34240.00	33634.10	34156.69
02/06/23	33874.44	33962.84	33683.58	33891.02
02/03/23	33926.30	34179.58	33813.86	33926.01
02/02/23	34129.30	34145.14	33814.78	34053.94



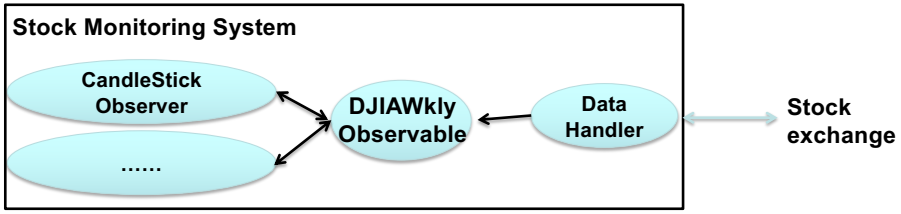
DJIA Weekly Summary

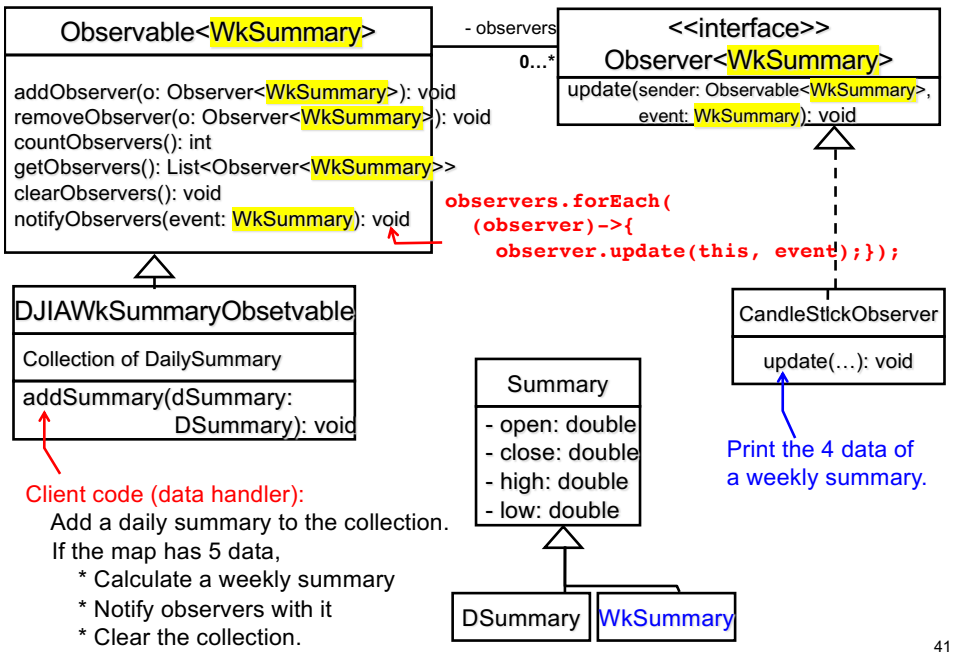
- The 5-day historical data can be used to produce a weekly summary:
  - Open
    - The 9am-price on the 1st day
  - Close
    - The 4pm-price on the last (5th) day

  High
  The highest price during 5days

  Low
  The lowest price during 5 days.

Calculation and Notification of Weekly Summary





- Make sure to use *Observer* properly.
  - c.f. Lec note 1
- Have your client code
  - Parse a CSV with Stream API to create 5 instances of DSummary.
  - Add DSummary instances to DJIAWkSummaryObservable one by one.
  - Calculate weekly summary data with Stream API
- Have your observer print out 4 data of a weekly summary.
- [Optional] Work on the calculation and notification of monthly summary data.

Tips

- Parsing a CSV file to **`List<List<String>>`**
  - By stripping commas and double quotations.

```
Path path = Paths.get(...);
try( Stream<String> lines = Files.lines(path) ){
  List<List<String>> csv =
      lines.map( line -> {
        return Stream.of( line.split(",") )
                     .map(value->value.substring(...))
                     .collect( Collectors.toList() ); })
           .collect( Collectors.toList() );
} catch (IOException ex) {}
```

- Tweak this code to get **`List<List<Double>>`**
  - You could choose to use native-type double.


```
// Stream to array conversion
//
String[] arr = stream.toArray(String[]::new);

// Array to stream conversion
//
Stream<String> stream = Arrays.stream(arr);

// Stream to list conversion
//
List<String> list = stream.collect(Collectors.toList());

// List to stream conversion
//
Stream<String> stream = list.stream();

// List to array conversion
//
String[] arr = list.toArray(new String[0]);

// Array to list conversion
//
List<String> list = Arrays.asList(arr);
```


Appendix: NIO-based File/Path Handling and Try-with-resources Statement

(1) Dealing with File/Directory Paths in NIO

- **`java.nio.Paths`**
  - A utility class (i.e., a set of static methods) to create a path in the file system.
    - Path: A sequence of directory names
      - Optionally with a file name in the end.
  - A path can be *absolute* or *relative*.
    - `Path absolute = Paths.get("/Users/jxs/temp/test.txt");`
    - `Path relative = Paths.get("temp/test.txt");`
- **`java.nio.Path`**
  - Represents a path in the file system.
  - Given a path, *resolve* (or determine) another path.
    - `Path absolute = Paths.get("/Users/jxs/");`
      `Path another = absolute.resolve("temp/test.txt");`
    - `Path relative = Paths.get("src");`
      `Path another = relative.resolveSibling("bin");`

Just in Case: Passing a Variable # of Parameters to a Method

- `Paths.get()` can receive a variable number of parameter values (1 to many values)
  - c.f. Java API documentation
  - `Paths.get(String first, String... more)`
    - `Paths.get("temp/test.txt");` // relative path
    - `Paths.get("temp", "test.txt");` // relative path
    - `Paths.get("/", "Users", "jxs");` // absolute path
  - `String... More` → Can receive zero to many String values.
  - Introduced in Java 5 (JDK 1.5)

- Parameter values are handled with an array.
  - ```
    class Foo{
        public void varParamMethod(String... strings){
            for(int i = 0; i < strings.length; i++){
                System.out.println(strings[i]); } } }
    ```
  - ```
    Foo foo = new Foo();
    foo.varParamMethod("U", "M", "B");
    ```
- **`String... Strings`** is a syntactic sugar for **`String[] strings`**.
  - Your Java compiler transforms the above code to:
    - ```
      class Foo{
          public void varParamMethod(String[] Strings){
              for(int i = 0; i < strings.length; i++){
                  System.out.println(strings[i]); } } }
      ```
    - ```
      Foo foo = new Foo();
      String[] strs = {"U", "M", "B"};
      foo.varParamMethod(strs);
      ```


Reading and Writing into a File w/ NIO

- `java.nio.file.Files`
  - A utility class (i.e., a set of static methods) to process a file/directory.
  - Reading a byte sequence and a char sequence from a file
    - ```
      Path path = Paths.get("/Users/jxs/temp/test.txt");
      byte[] bytes = Files.readAllBytes(path);
      String content = new String(bytes);
      ```
    - ```
      List<String> lines = Files.readAllLines(path);
      for(String line: lines){
            System.out.println(line); }
      ```
  - Writing into a file
    - `Files.write(path, bytes);`
    - `Files.write(path, content.getBytes());`
    - `Files.write(path, bytes, StandardOpenOption.CREATE);`
    - `Files.write(path, lines);`
    - `Files.write(path, lines, StandardOpenOption.WRITE);`
    - `StandardOpenOption: CREATE, WRITE, APPEND, DELETE_ON_CLOSE, etc.`

NIO (java.nio) v.s. Traditional I/O (java.io)

- NIO provides simpler or easier-to-use APIs.
  - Client code can be more concise and easier to understand.
- NIO:
  - ```
    Path path = Paths.get("/Users/jxs/temp/test.txt");
    byte[] bytes = Files.readAllBytes(path);
    String content = new String(bytes);
    ```
- java.io:
  - ```
    File file = ...;
    FileInputStream fis = new FileInputStream(file);
    int len = (int)file.length();
    byte[] bytes = new byte[len];
    fis.read(bytes);
    fis.close();
    String content = new String(bytes);
    ```

NIO (java.nio) v.s. Traditional I/O (java.io)

- NIO:
  -
```
Path path = Paths.get("/Users/jxs/temp/test.txt");
List<String> lines = Files.readAllLines(path);
```
- java.io:
  -
```
int ch=-1, i=0;
ArrayList<String> contents = new ArrayList<String>();
StringBuffer strBuff = new StringBuffer();
File file = ...;
InputStreamReader reader = new InputStreamReader(
                               new FileInputStream(file));
while( (ch=reader.read()) != -1 ){
    if( (char)ch == '\n' ){       //**line break detection
        contents.add(i, strBuff.toString());
        strBuff.delete(0, strBuff.length());
        i++;
        continue;
    }
    strBuff.append((char)ch);
}
reader.close();
```

** The perfect (platform independent) detection of a line break should be more complex.
Unix: '\n', Mac: '\r', Windows: '\r\n' c.f. BufferedReader.readLine()

NIO (java.nio) v.s. Traditional I/O (java.io)

- NIO:
  -
```
Path path = Paths.get("/Users/jxs/temp/test.txt");
List<String> lines = Files.readAllLines(path);
```
- java.io (a bit simplified version):
  -
```
int ch=-1, i=0;
ArrayList<String> contents = new ArrayList<String>();
StringBuffer strBuff = new StringBuffer();
File file = ...;
FileReader reader = new FileReader(file); //***
while( (ch=reader.read()) != -1 ){
    if( (char)ch == '\n' ){  //** Line break detection
        contents.add(i, strBuff.toString());
        strBuff.delete(0, strBuff.length());
        i++;
        continue;
    }
    strBuff.append((char)ch);
}
reader.close();
```

*** FileReader: A convenience class for reading character files.

Files in Java NIO

- readAllBytes(), readAllLines()
  - Read the whole data from a file without buffering.
- write()
  - Write a set of data to a file without buffering.
- When using a large file, it makes sense to use **`BufferedReader`** and **`BufferedWriter`** with **`Files`**.
  -
```
Path path = Paths.get("/Users/jxs/temp/test.txt");
BufferedReader reader = Files.newBufferedReader(path);
while( (line=reader.readLine()) != null ){
    // do something
}
reader.close();
```
  -
```
BufferedWriter writer = Files.newBufferedWriter(path);
writer.write(...);
writer.close();
```


Just in case: Buffering

- At the lowest level, read/write operations deal with data *byte by byte*, or *char by char*.
  - File access occurs *byte by byte*, or *char by char*.
- Inefficient if you read/write a lot of data.
- Buffering allows read/write operations to deal with data in a coarse-grained manner.
  - Chunk by chunk, not byte by byte or char by char
  - Chunk = a set of bytes or a set of chars
    - The size of a chunk: 512 bytes by default, but configurable

Getting Input/Output Streams from `Files`

- Input and output streams can be obtained from **`Files`**.
  - `Path path = Paths.get("/Users/jxs/temp/test.txt");`
    `InputStream is = Files.newInputStream(path);`
    - `is` contains an instance of ChannelInputStream, which is a subclass of InputStream.
    - Make sure to call `is.close()` in the end.
- Can decorate the input/output stream with filters.
  - `ZipInputStream zis = new ZipInputStream(`
    `Files.newInputStream(path) );`
    - Make sure to call `zis.close()` in the end.

Never Forget to Call close()

- Need to call `close()` on each input/output stream (or its filer) in the end.
  - Must-do: Follow the *Before/After* design pattern.
    - In Java, use a *try-catch-finally* or *try-finally* statement.

```
» Open a file here.
  try{
      Do something with the file here.
      Throw an exception if an error occurs.
  }catch(...){
      Error-handling code here.
  }finally{
      Close the file here.
  }
```

  - Note: No need to call `close()` when using `readAllBytes()`, `readAllLines()` and `write()` of `Files`.


```
- Path path = Paths.get("/Users/jxs/temp/test.txt");
  BufferedReader reader = Files.newBufferedReader(path);
  try{
    while( (line=reader.readLine()) != null ){
        // do something
    }
  }catch(IOException ex){
    ... // Error handling
  }finally{
    reader.close();
  }
```


(2) Try-with-resources Statement

- Allows you to skip calling close() explicitly in the finally block.
  - *Try-catch-finally*

```
- Open a file here.
  try{
      Do something with the file here.
  }catch(...){
      Handle errors here.
  }finally{
      Close the file here.
  }
```

  - *Try-with-resources*

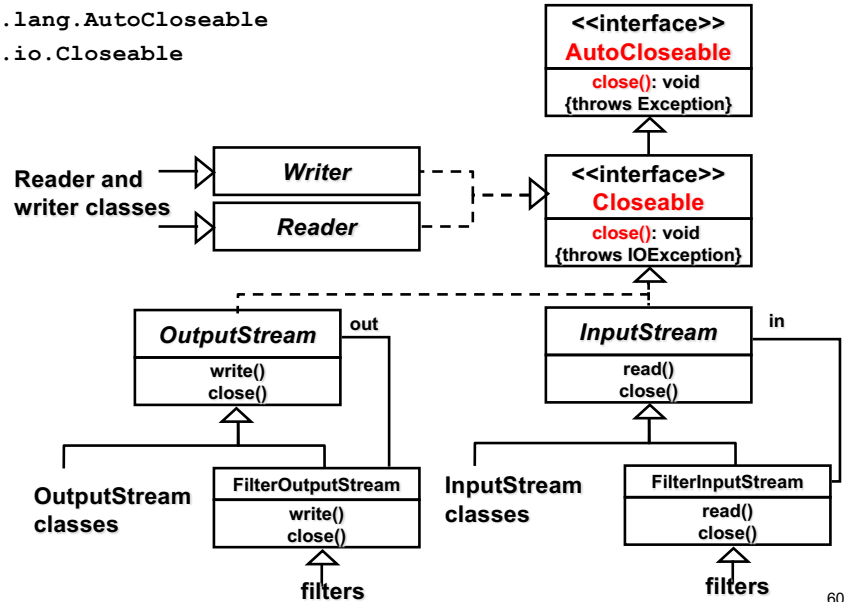
```
• try ( Open a file here ){
      Do something with the file here.
  }
```

Try to use this in homework
code should not be ancient java coding style else you'll get 0
as given. in Tips

- `close()` is automatically called on a resource used for reading or writing to a file, when exiting a try block.
  - ```
    try( BufferedReader reader =
            Files.newBufferedReader( Paths.get("test.txt")) ){
        while( (line=reader.readLine()) != null ){
            // do something }
     }
    ```
    - No explicit call of close() on `reader` in the finally block. `reader` is expected to implement the `AutoCloseable` interface.
  - ```
    try( BufferedReader reader = Files.newBufferedReader(...);
         PrintWriter writer = new PrintWriter(...) ){
        while( (line=reader.readLine()) != null ){
            // do something
            writer.println(...); }
    }
    ```
    - Can specify multiple resources in a try block. close() is automatically called on all of them. They all need to implement `AutoCloseable`.

AutoCloseable Interface

- `java.lang.AutoCloseable`
- `java.io.Closeable`



- Recap: No need to call `close()` when using `readAllBytes()`, `readAllLines()` and `write()` of `Files`.
  - Those methods internally use the try-with-resources statement to read and write to a file.

Try-with-resources-Catch-Finally

- Catch and finally blocks can be attached to a try-with-resources statement.
  - ```
    try( BufferedReader reader =
            Files.newBufferedReader( Paths.get("test.txt")) ){
        while( (line=reader.readLine()) != null ){
            // do something. This part may throw an exception.}
    }catch(...){
        //This block runs if the try block throws an exception.
    }finally{
        ...
        //No need to do reader.close() here.
    }
    ```
    - The catch and finally blocks run (if necessary) AFTER close() is called on `reader`.
    ```