Welcome to CS681

Tue Thu 5:30pm to 6:45pm

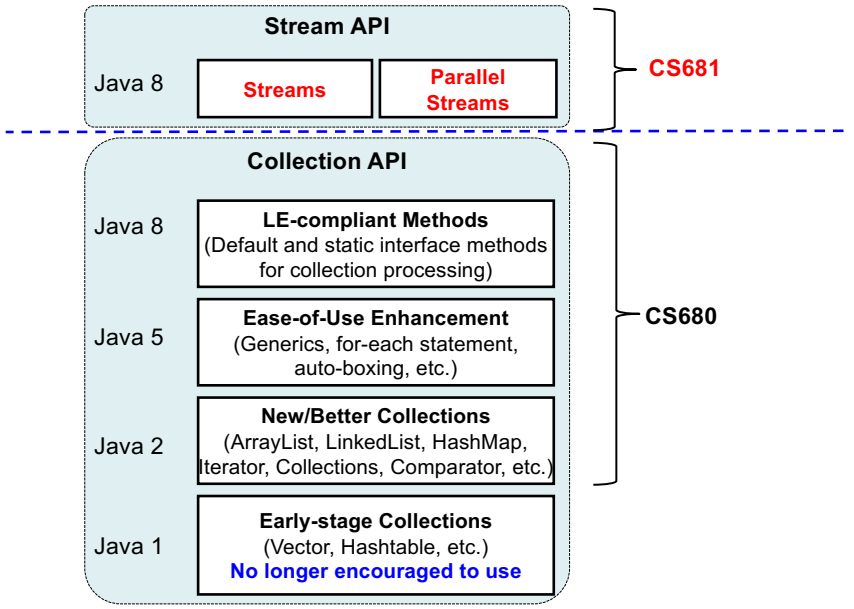
Course Topics: Advanced Software Engineering

- Functional programming with lambda expressions (LEs)
- Concurrent object-oriented programming
  - Multi-threading with Java

Course Topic #1

- Functional programming with lambda expressions (LEs)
  - Continuation from CS680
  - Collection processing with LEs
    - Stream API, which heavily uses LEs
    - Design patterns (data processing patterns); e.g., MapReduce

Collection and Stream APIs in Java



Please Note…

- You are expected to understand some basics about LEs in Java.
  - i.e., Topics covered in CS680
- CS681 will cover more advanced topics atop those basic understanding.
  - Take a look at CS680 lecture notes and refresh your memory about Java LEs.

Course Topic #2

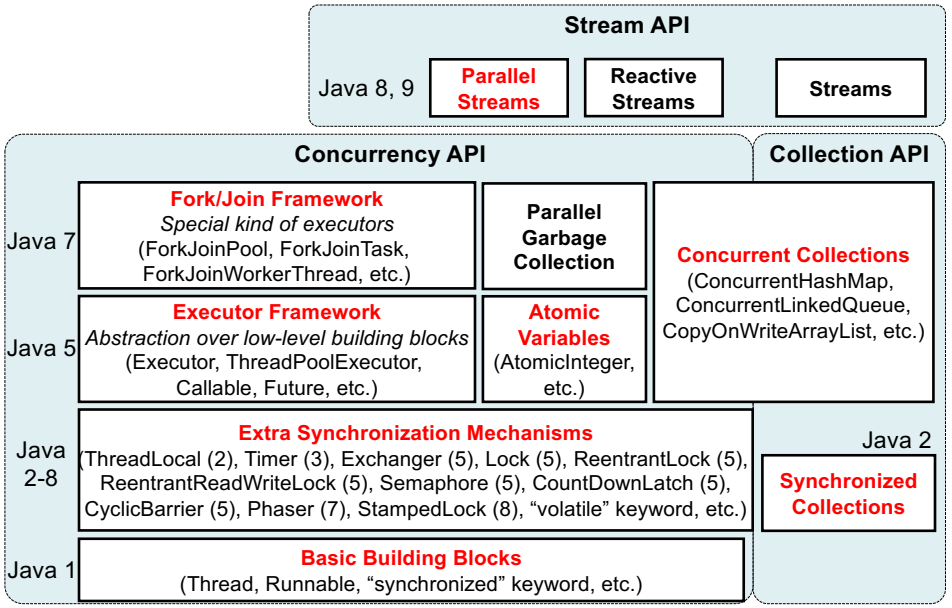
- Concurrent programming (multi-threading)
  - Mechanisms, data structures, APIs and frameworks for concurrency (multi-threading) in Java
  - Concurrent object-oriented design patterns
    - e.g., Concurrent *Singleton*, Concurrent *Observer*, Concurrent *Visitor*, Concurrent *Command*, Concurrent *MapReduce*, *Producer-Consumer*
  - Concurrency with LEs
    - With concurrency-aware collections, parallel streams, etc.

Concurrency as a Part of SE? Yes!

- Concurrency was for specific groups of software engineers to develop specific types of software.
  - Concurrency in programming: Since mid 60s
    - PL/I ("TASK" statement, '65), Concurrent Pascal ('75), Concurrent Smalltalk ('86), Objective-C ('86), Java ('96) , …, C# ('02), Erlang ('06), …, etc.
  - Concurrency in OSes: Since 70s
    - Multix ('70), Mach (cthreads, '85) , NeXTSTEP (cthreads, '86), OS/2 ('87), Solaris libthread ('93), Windows NT ('94), pthreads (POSIX threads '95), Windows ('95), Mac OS X ('01), Linux ('03), iOS, Android, etc.
  - Early applications
    - High-performance and real-time computing in…
      - defense, aviation, financial trading (algorithmic and high freq. trades), etc.

- Concurrency is now important and useful for most software engineers to develop any software.
  - Not only "special" applications but also "normal" apps can enjoy, or even require, concurrency.
    - e.g., Web apps, desktop apps, smartphone/tablet apps, IoT apps, etc.
  - **Goals**: *Responsiveness* and *performance* improvement
    - Perform fine-grained multi-tasking
    - Do Better I/O handling
    - Take advantage of multi-core CPUs

Concurrency API in Java



Course Work and Grading

- Course work
  - Lectures
  - Homework (coding in Java)
  - Quizzes
- Grading factors
  - Coding homework (90 to 95%)
  - Quizzes (5 to 10%)
- No (midterm and final) exams.

Homework

- For each HW, do your best to turn in your solution in a week or two.
  - Will give you extra points if you regularly turn in your solutions.
- There are 2 **FIRM** deadlines for HW submissions.
  - March 19 (Sun), midnight
    - Submit your solutions for the HWs given by early March.
      - I will NOT grade any late submissions.
  - End of the semester (mid May)
    - Specific date to be announced later.
    - Submit your solutions for the remaining HWs.

- Place your HW solutions at a private GitHub repo and share it with me.
  - My GitHub account is jxsboston.
- Notes
  - Upload source code (.java files) only, NOT binary code (.class, .jar files).
  - No need to do unit testing.
  - Use Ant for the first half of HWs (as you did in CS680).
  - Use Gradle for the remaining HWs.
    - Will teach how to use Gradle.
- Questions:
  - Use **Junichi.Suzuki@umb.edu**

Important Notice

- You must work on every single HW alone (by yourself).
  - You can discuss HW assignments with others. However, you must do your coding yourself.
- It is an academic crime to
  - Copy (or steal) someone else's code and submit it as your own work.
  - Allow someone else to copy (or steal) your code and submit it as his/her work.
    - Use a private repo to avoid this.
- You will end up with a serious situation if you commit this academic crime.
  - The University, College, Department and I have no mercy about it.

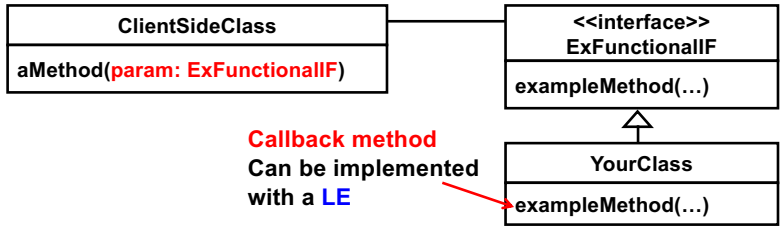
Recap: Lambda Expressions in Java

Benefits of Using Lambda Expressions

- Can make your code more concise (less repetitive)
  - *Callback* functions/methods as LEs
    - e.g., Comparator (*Strategy*), *Command*
- Can enjoy the power of functional programming
  - e.g., higher-order functions (e.g. `Comparator.comparing()`)
- Can gain a new way to process collections
  - "Internal" iteration as opposed to traditional "external" iteration
  - Collection streams (Stream API)
    - Enables Map-Reduce data processing
- Can simplify concurrent programming (multi-threading)
  - Repeatedly and concurrently executed code (block) as a LE

Where/When to Use Lambda Expressions

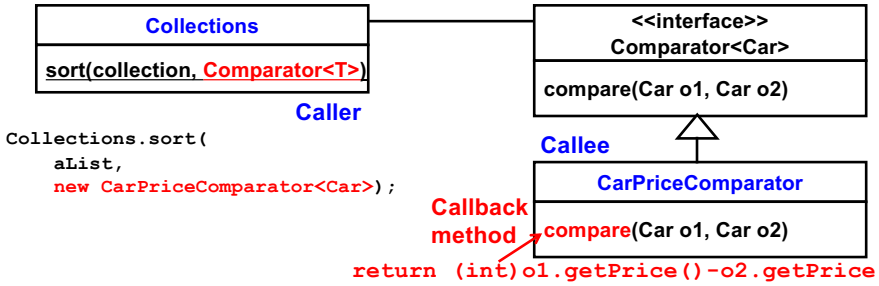
- A functional interface is defined.
  - An interface that has *exactly one* abstract method
  - You are expected to define a class that implements the functional interface by implementing the body of the abstract method.
- There is a method that accepts a parameter that is typed with that functional interface.
  - It will invoke the method that is implemented in your class.
    - The method (`exampleMethod()`) is often called "callback" method.



- Some design patterns follow this structure.
  - You can use LEs to implement callback methods in those design patterns.
  - e.g. *Strategy*, *Command*, *Observer*, etc.

Callback Method in Comparator

- Two key players
  - A "callee" class that implements a callback method
    - You implement it, but you do not call the method directly.
  - A "caller" class that will call the callback method in the future
    - Someone else has implemented it.
- Interaction between a callee and a caller
  - You make a callee class instance and pass it to a caller, so the caller can call the callback method in the future.



What does `Collections.sort()` do?

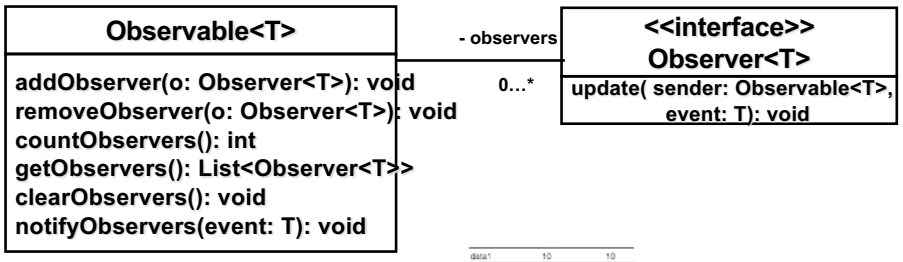
```
class Collections
    static ... sort(List<T> list, Comparator<T> c){
        for each pair (o1 and o2) of elements in list{
            int result = c.compare(o1, o2);
            if(result < 0){
                ...
            }else if(result > 0){
                ...
            }else if(result==0){
                ...
        } } }
```

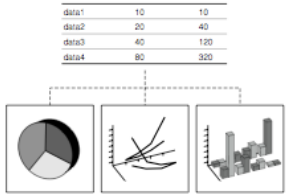

Another Example: *Observer*

- Intent
  - Event notification
    - Define a one-to-many dependency between objects so that, when one object changes its state, all its dependents are notified automatically
- a.k.a
  - Publish-Subscribe (pub/sub)
  - Event source - event listener
- Two key participants (classes/interfaces)
  - Observable (model, publisher or subject)
    - Propagates an event to its dependents (observers) when its state changes.
  - Observer (view and subscriber)
    - Receives events from an observable object.

Simplified Version of MS Azure SDK

- `edu.umb.cs680.Observable`
- `edu.umb.cs680.Observer`

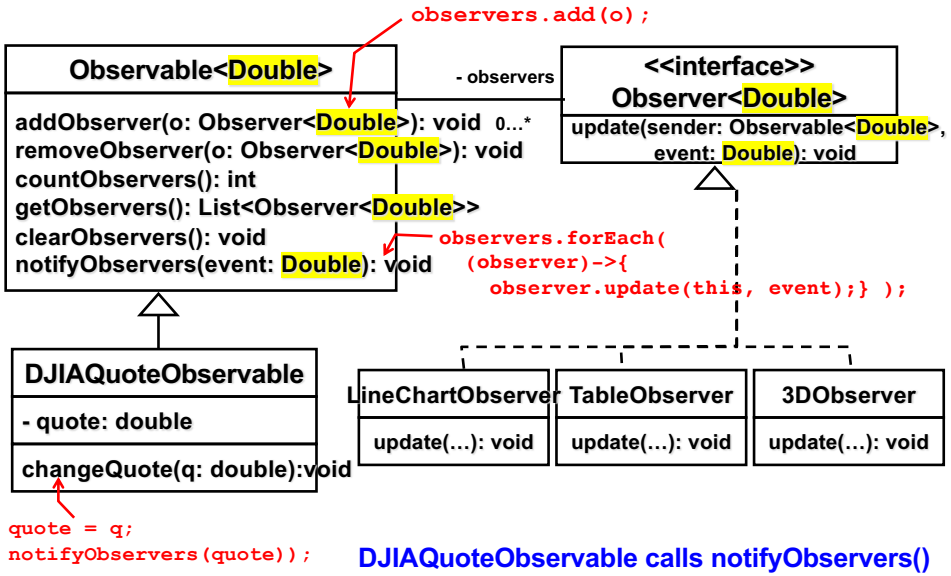




Exercise (Not HW)

- Read and understand `Observable<T>` and `Observer<T>`.
- Implement 2 event notification scenarios.

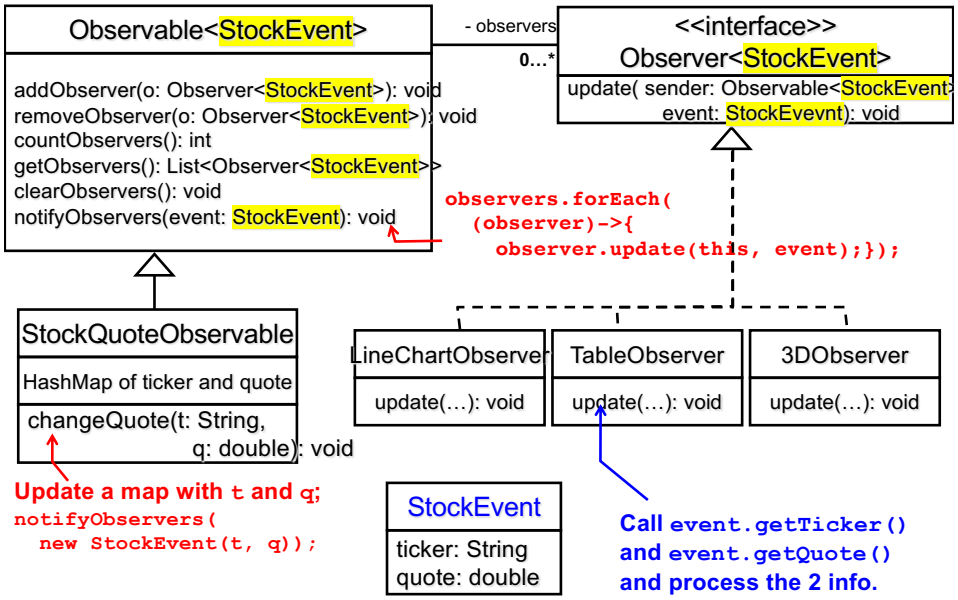
Event Notification #1



DJIAQuoteObservable calls notifyObservers() to sent out Double-typed events.

- DO NOT modify `Observable<T>` and `Observer<T>`.
  - Keep them as they are, and just "use" them.
- Define `DJIAQuoteObservable` (in `DJIAQuoteObservable.java`) as follows:
  - `public class DJIAObservable extends Observable<Double>{ ...}`
- Define `LineChartObserver` (in `LineChartObserver.java`) as follows:
  - `public class LineChartObserver extends Observer<Double>{ ...}`

Event Notification #2



- DO NOT modify `Observable<T>` and `Observer<T>`.
  - Keep them as they are, and just "use" them.

- Define `StockQuoteObservable` (in `StockQuoteObservable.java`) as follows:

```
public class StockObservable extends Observable<Stockevent>{
...}
```

- Define `LineChartObserver` (in `LineChartObserver.java`) as follows:

```
public class LineChartObserver extends Observer<StockEvent>{
...}
```


Exercise (Not HW)

- DO NOT modify `Observable<T>` and `Observer<T>`.
  - Keep them as they are, and just "use" them.

- Implement the 2 event notification scenarios by replacing the subclasses of Observer with LEs.
  - `Observer<T>` is a functional interface.

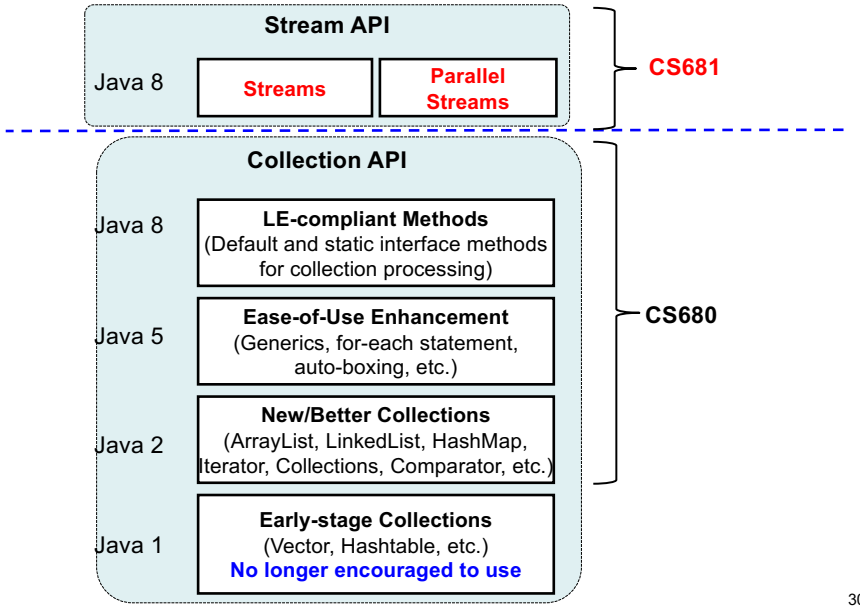
```
StockQuoteObservable<StockEvent> o = new StockQuoteObservable<>();
Observer updateQuoteLineChart =
    (StockQuoteObservable<StockEvent> sender, StockEvent event)->{
          {System.out.println(...);};
o.addObserver( quoteUpdateLineChart );
o.addObserver( ... );
o.addObserver( ... );

o.changeQuote( ... );
```


Stream API

New Way of Collection Processing with LEs

- Java 8 made major improvements to the Collection API by
  - Adding new static and default methods in existing interfaces
    - Those new methods allow you to pass LEs to your collection object.
    - e.g. `Iterable.forEach()`
    - c.f. CS680 lecture notes
  - Adding **streams**.
    - `java.util.stream.Stream<T>`
      - Has many methods that take care of common operations to be performed on collection elements (T).

Collection and Stream APIs in Java

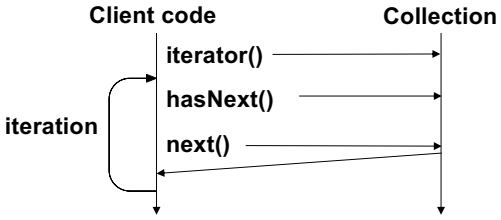


Traditional Way of Collection Processing

- ***External*** iteration: Iterate over a collection and perform an operation on each element in turn.

```
Iterator<String> iterator = strList.iterator();
while( iterator.hasNext() ) {
   System.out.println(iterator.next()); }
```

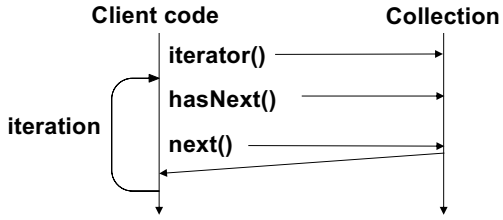
- Iteration runs outside of a collection.
- Need to write boilerplate code whenever you need to iterate over the collection.



- ***External*** iteration: Iterate over a collection and perform an operation on each element in turn.

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

  - Iteration runs outside of a collection.
  - Need to write a lot of boilerplate code whenever you need to iterate over the collection.

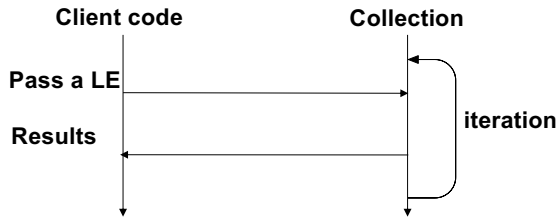


- The loop mixes up *what you want to do on a collection* and *how you do it*.
  - Not that easy to understand and maintain because "what" is often obscured by "how." ("How" is often emphasized too much than "what.")

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

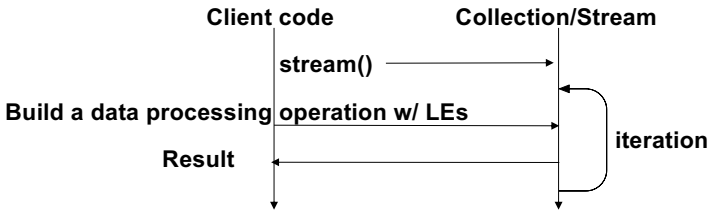
New Way of Collection Processing (1)

- ***Internal*** iteration:
  - `Iterable.forEach()`
    - Does not return an `Iterator` that externally controls the iteration
    - Creates an equivalent object, which works inside of a collection.
      - A collection internally uses the iterator-like object to perform iteration
  - `strList.forEach( (String i)->System.out.println(i) )`



New Way of Collection Processing (2)

- ***Internal*** iteration:
  - `stream()`: Plays a similar role to the call of `iterator()`
    - Does not return an `Iterator` that externally controls the iteration
    - Returns a `Stream`, which helps build a complex data processing operation on a collection and runs it by performing iteration.
  -
```
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.
  - Collection processing looks more declarative, not procedural.
    - c.f. SQL statements
  -
```
long count = carList.stream()
                    .filter( (Car car)-> car.getPrice()<5000 )
                    .count();
```

    - Get a stream that contains `Car` instances.
    - Filter the stream to keep the `Car` instances that are less expensive than $5000
    - Count the number of remaining `Car` instances in the steam.

Functional Interfaces

- Since its version 8, Java extensively uses ***functional interfaces*** in many APIs.
  - e.g., `Comparator<T>`: Used for a specific purpose:
    - Comparison of two data (objects)
    - c.f. `Collections.sort()`

- In addition to *special-purpose* functional interfaces, Java has *general-purpose* ones that can be generally used in many scenarios or for many purposes.

Important General-Purpose Functional Interfaces

Function<T,R>
Consumer<T>
Predicate<T>
Supplier<T>
UnaryOperator<T>
BinaryOperator<T>
BiFunction<U,T>

All available in `java.util.function`

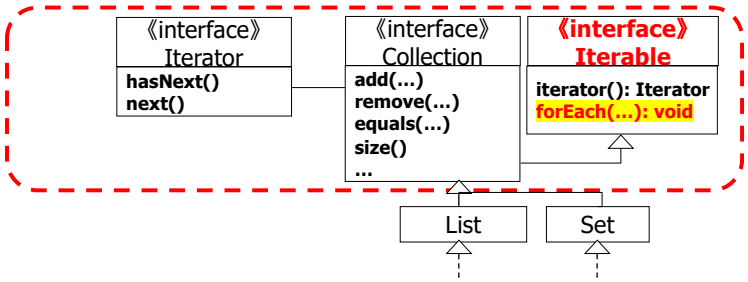
- Just like special-purpose functional interfaces…
- Each general-purpose functional interface helps you define and use a "function" as a form of lambda expression.
  - Anonymous
  - Zero or more inputs
  - Zero or more outputs

- To understand what inputs/outputs the function is supposed to take/return, check out the (only) abstract method of that functional interface.
  - e.g., `Comparator<T>`
    - `int compare(T o1, T o2)`

	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()

An Example General-Purpose Functional Interface: `Consumer`

- `Iterable<T>`
  - `default void forEach(Consumer<T> action)`
    - Applies a given function on each element of a collection that implements `Iterable`.



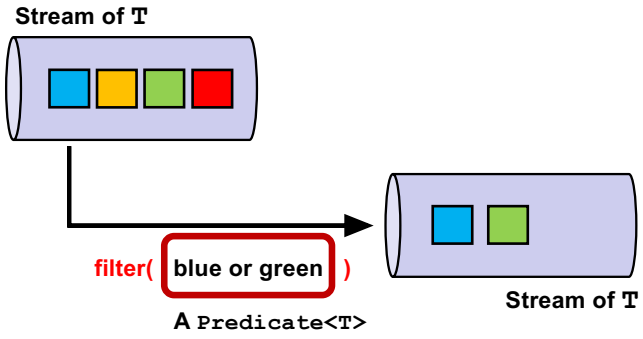

```
LinkedList<String> strList = ...
strList.forEach( (String s)->System.out.println(s) );
```

- `Consumer<T>`: Functional interface
  - Represents a function (LE) that accepts a parameter (T) and returns nothing.
    - The LE receives a collection element as a parameter and specifies an action to be applied to the collection element as its code block.

```
LinkedList<String> strList = ...
strList.forEach( (String s)->System.out.println(s) );
```

	Params	Returns	Example use case
Consumer<T>	T	void	Print out a collection element (T). Iterable.forEach()

	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 (*)



Streams and Collections

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

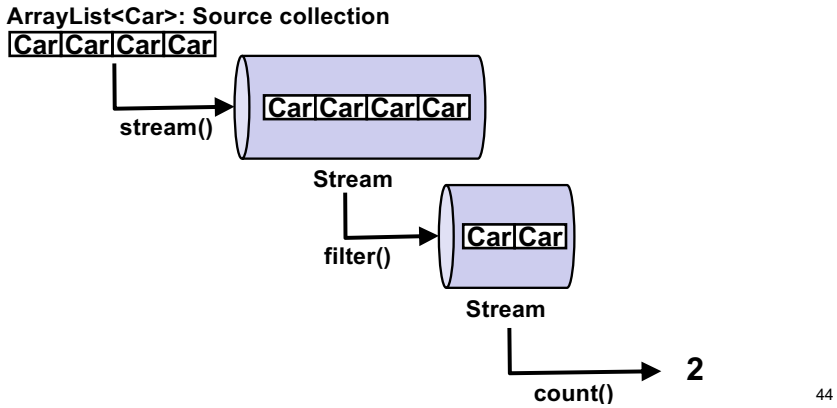
```
long count = carList.stream()
                    .filter( (Car car)-> car.getPrice()<5000 )
                    .count();
```


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 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();
```


How Many Traversals?

- Traditional

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

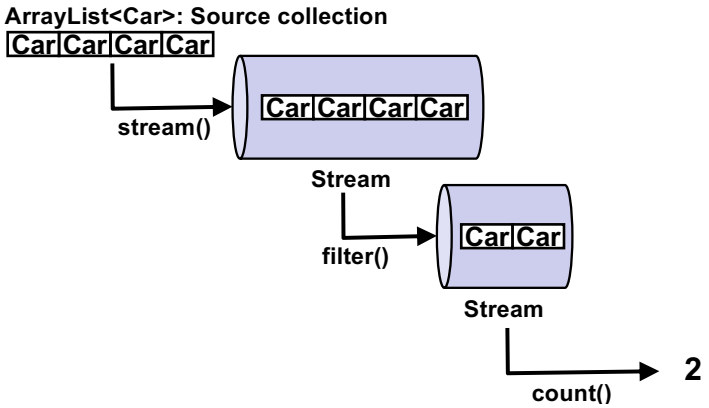
  - Traversing the list once .

- New

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

  - Traversing the list twice?
  - No, only once!

- This conceptual diagram is used to intuitively understand the structure and behavior of a stream pipeline.
  - Real traversal execution (internal iteration) is a bit different (more efficient)



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.

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

  - The filtering operation is never executed.

```
... = carList.stream()
            .filter((Car car)->{
                      System.out.println(car.getPrice());
                      return car.getPrice()<5000} );
```

  - Nothing is printed out.
  - The filtering operation is never executed.