Welcome to CS681

Tue Thu 5:30pm to 6:45pm

W-2-0200

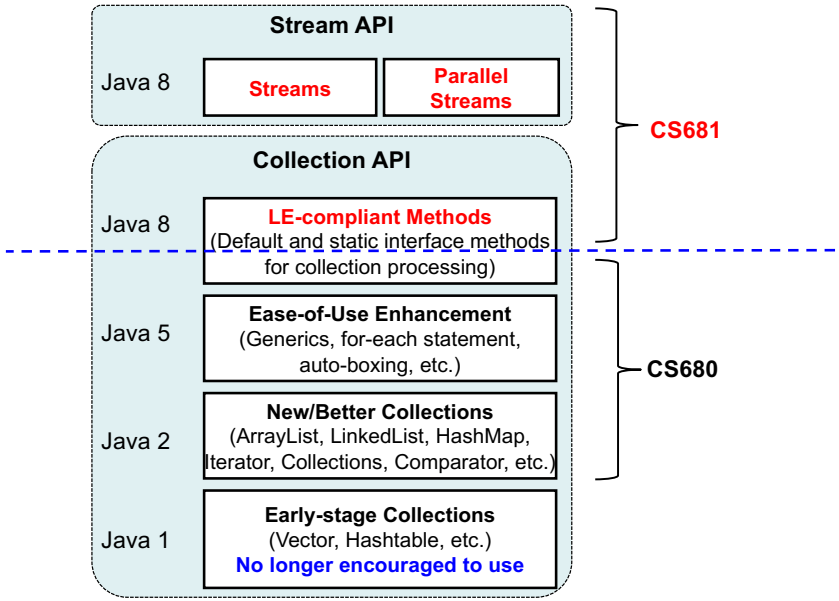
Course Topics: Advanced Software Engineering

- Functional programming with lambda expressions
- Concurrent programming (multi-threading)

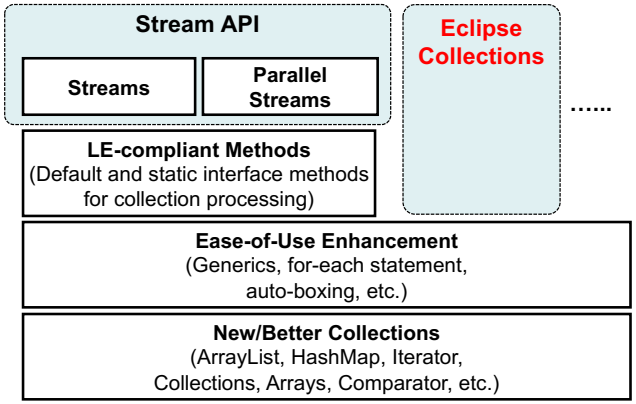
Course Topic #1

- Functional programming with lambda expressions
  - 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



Extra Utilities



- Eclipse Collections
  - https://www.eclipse.org/collections/
  - https://github.com/eclipse/eclipse-collections-kata
  - Used to be Goldman Sachs (GS) Collections
    - https://github.com/goldmansachs/gs-collections

Course Topic #2

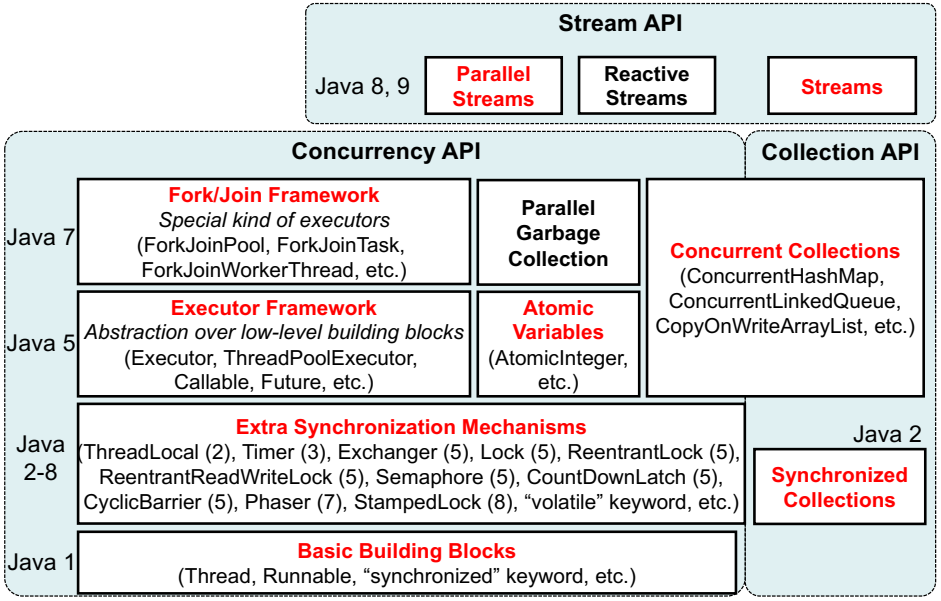
- Concurrent programming (multi-threading)
  - Mechanisms, data structures, libraries and frameworks for concurrency (multi-threading)
  - Concurrent object-oriented design patterns
    - e.g., Concurrent Singleton, Concurrent Observer, Concurrent Visitor, Concurrent MapReduce, Producer-Consumer
  - 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 computing 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 a variety of software.
  - Not only "special" applications but also "normal" apps can enjoy, or even require, concurrency.
    - e.g., Web apps, smartphone/tablet apps
  - **Goals**: *responsiveness* and *performance* improvement
    - Fine-grained multi-tasking
    - Better I/O handling
    - Multi-core CPUs

Concurrency API in Java



Course Work

- Lectures
- Homework
  - Reading
  - Coding (in Java)
- Individual project (TBD/A)

Grading

- Homework (60%)
- Project deliverables (30%)
- Quizzes (10%)
  - Occasionally, in lectures
  - May not have quizzes at all, depending on the class size
- No midterm and final exams.

- Some/many HWs have submission deadlines.
  - Do your best to meet the deadlines.
    - If you regularly meet them, you will get some extra points.
  - You can miss the deadlines. You DO NOT have to notify me that you will miss or have missed a deadline.
    - Up to a week late or so: No problem. I favor better code than timeliness. Focus on your work, not making excuses to me.
    - Beyond that: Depends.
- In principle, you cannot replace your HW solution after you submit it.

My Email Addresses

- Questions → **jxs@cs.umb.edu**
  - I regularly check this account.
- HW solutions → **umasscs681@gmail.com**
  - I occasionally check this account.
    - Once a week or so.

Your Email Address

- Send your (preferred) email address to **jxs@cs.umb.edu**.
  - I will use that address to email you lecture notes, announcements, etc.
  - You will use that address to submit your HW solutions.
- No need to do this, if you took CS680 in Fall 2018.
  - I will use an email address that you used in CS680.

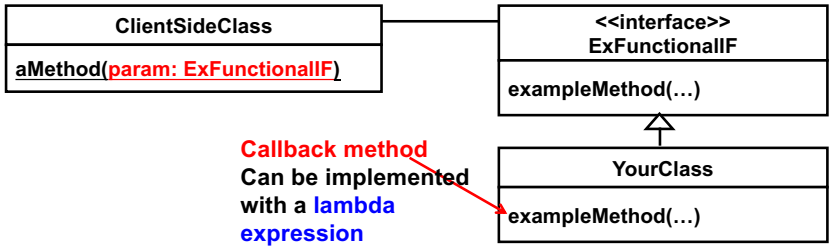
Recap: Lambda Expressions in Java

Benefits of Using Lambda Expressions

- Can make your code more concise (less repetitive)
  - *Callback* functions/methods as lambda expressions
    - e.g., Comparator, event handling (e.g., Observer), GUI
- Can enjoy the power of functional programming
  - e.g., higher-order functions
- Can gain a new way to process collections
  - Newly-added *default methods* that accept lambda expressions
  - "Internal" iteration as opposed to traditional "external" iteration
  - Collection streams
    - Enables Map-Reduce data processing (a topic in CS681)
- Can simplify concurrent programming (multi-threading)
  - Repeatedly and concurrently executed code (block) as a lambda expression (a topic in CS681)

Where/When to Use Lambda Expressions

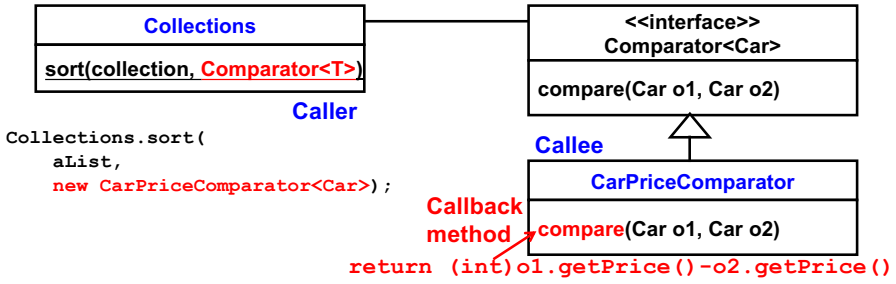
- There is a functional interface:
  - An interface that has an abstract method
  - You are expected to define a class that implements the interface with the body of the abstract method.
- There is a method that accepts a parameter that is typed with that functional interface.
  - It will invoke the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



- Many design patterns follow this structure.
  - You can use lambda expressions to implement those design patterns.
  - In CS680
    - Used lambda expressions to implement *Strategy*
  - What else?
    - *Observer*, *Command*, etc. etc.

Callback Method in Comparator

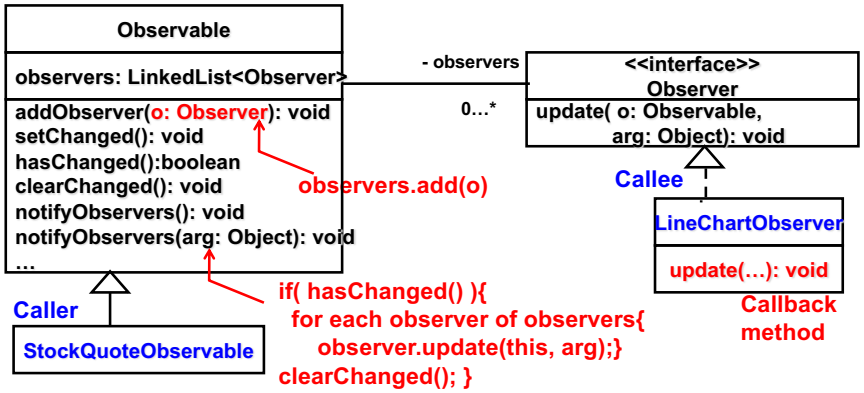
- Two key players
  - A "callee" class that implements a callback method
    - You implement it, but you do not call the method directly.
  - A "caller" class that will call the callback method in the future
    - Someone else has implemented it.
- Interaction between a callee and a caller
  - You make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

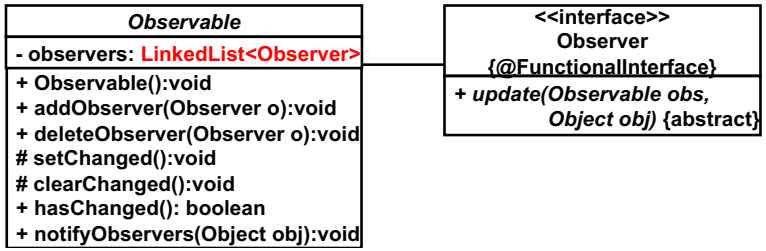
Another Example: *Observer*



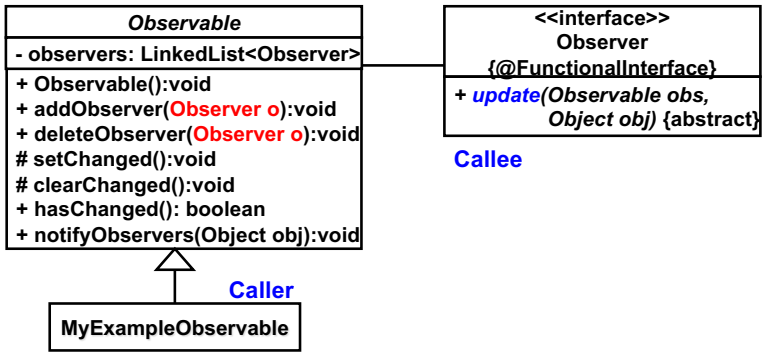

```
anObservable.addObserver( new LineChartObserver() );
...
anObservable.notifyObservers( ... );
```


HW 1: Implement *Observer* with LEs

- Define your own `Observable` (class) and `Observer` (interface)
  - DO NOT reuse `java.util.Observable` and `java.util.Observer`
  - Define `Observable` as an abstract class.
  - Define `Observer` as a functional interface.
    - `update()` as the only abstract method in that interface.
  - Implement all the methods that are available for `java.util.Observable`
    - c.f. Java API doc for expected behaviors/responsibilities of the methods
  - Use `LinkedList` to hold all registered observers.
    - c.f. CS680 slides on "ArrayList v.s. LinkedList"



- Use a lambda expression to implement `update()` and pass it to `addObserver()` and `deleteObserver()`.
  - In CS680, you defined a class that implements `Observer` and implemented `update()` in that class.



- To test your own `Observable` and `Observer`, use the classes that you implemented as the subclasses of `java.util.Observable` in CS680.
  - e.g., **`StockQuoteObservable`** and **`DJIAQuoteObservable`**

- Example client code

```
StockQuoteObservable observable = new StockQuoteObservable(...);
observable.addObserver( (Observable o, Object obj)->
                              {System.out.println(...);} );
observable.addObserver( ... );
observable.setQuote(...);
observable.notifyObservers(...);
```

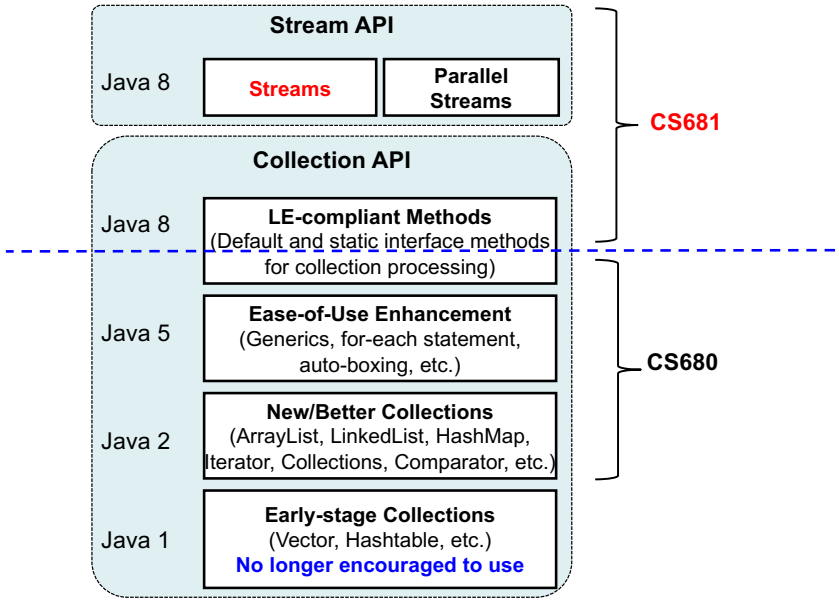
- Submission Requirements
  - Follow the submission requirements in CS680.
    - Use Ant.
    - Submit .java files and an Ant script (e.g. build.xml).
    - Never send me binary code.
    - Avoid attaching a .zip file. Use another archive file format.
  - No need to do unit testing.
  - Due: February 14 (Thu) midnight
- Follow the same requirements for all subsequent HWs

Stream API

Collection Processing with LEs

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

Collection and Stream APIs

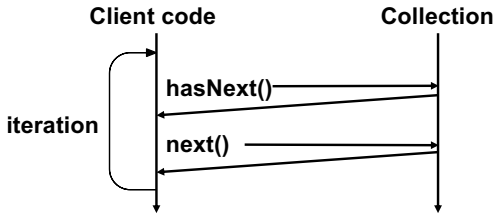


Traditional Way of Collection Processing

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

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

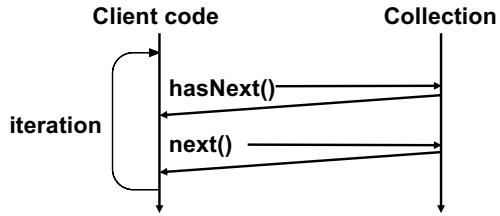
- Iteration occurs outside of a collection.
- Need to write a boilerplate code whenever you need to iterate over the collection.



- ***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++; }
```

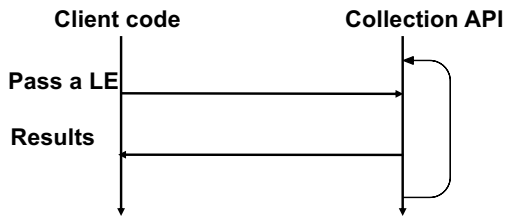
- Iteration occurs outside of a collection.
- Need to write a lot of boilerplate code whenever you need to iterate over the collection.



- The loop mixes up *what you want to do on a collection* and *how you do it*.
  - Not that easy to understand and maintainable because "what" is often obscured by "how." ("How" is often emphasized too much than "what.")
- Inherently serial
  - Hard to make it concurrent/parallel.

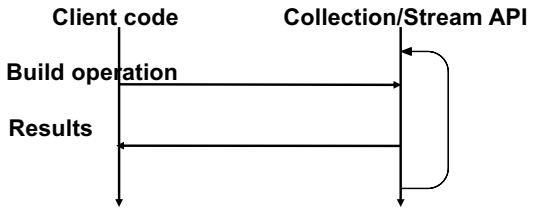
New Way of Collection Processing

- ***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) )`



- ***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 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 those `Car` instances in the steam.

General-Purpose Functional Interfaces

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

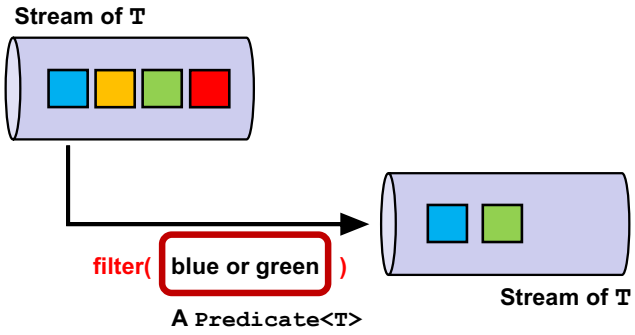
	Params	Returns	Example use case
Function<T,R>	T	R	Get the price (R) from a Car object (T) Generate a function (R) from another (T)
Consumer<T>	T	void	Print out a collection element (T)
Predicate<T>	T	boolean	Has this car (T) had an accident?
Supplier<T>	NO	T	A factory method. Create a Car object and 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.

	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()

Streams and Collections

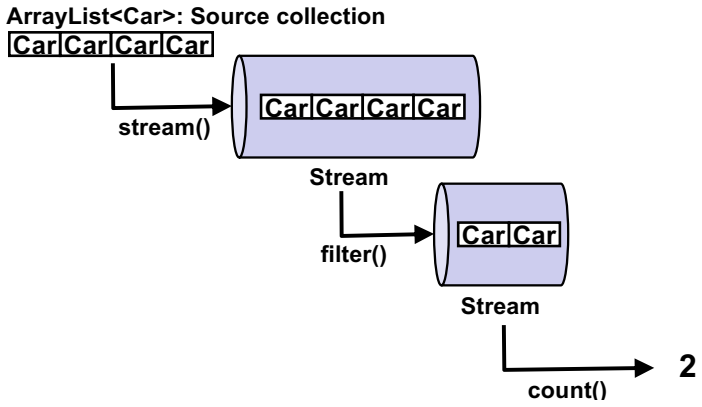
- Interface `Collection<E>`
  - `default Stream<E> stream()`
    - Returns a stream that uses this collection as its data source.
- `java.util.stream.Stream<T>`
  - `Stream<T> filter(Predicate<T> predicate)`
    - Returns a stream consisting of the elements of this stream that match a given predicate (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();
```


	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 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++;
}
```

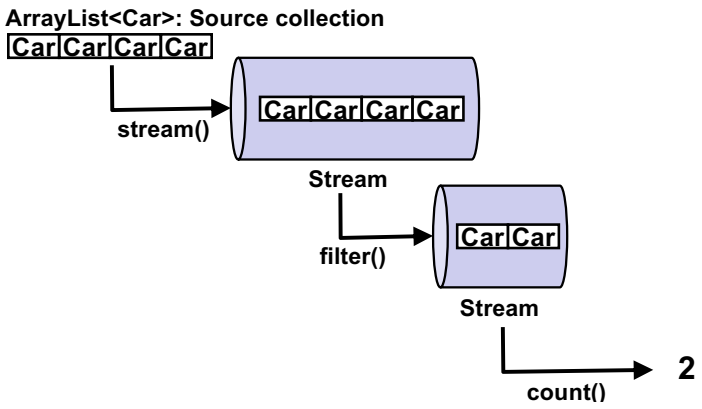
  - Traversing the list once .

- New

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

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

- This kind of conceptual diagram are 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 is never executed.

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

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