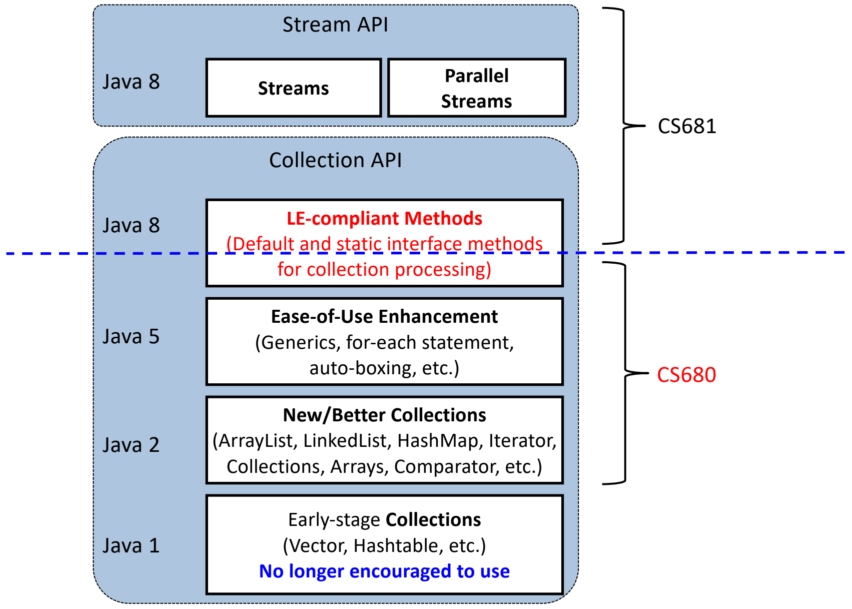
HW 15

- Implement the color adjustment/filtering example in Lecture Note ~~20~~ 18 in 2 versions
  - With the Strategy design pattern
  - With a lambda expression(s).

Notable Enhancements in Java 8

- Lambda expressions
  - Allow you to do *functional programming* in Java
- Static and default methods in interfaces
- Collection processing with lambda expressions (LEs)
  - Newly-added default and static interface methods to process collection elements with LEs
  - Stream API, which heavily uses LEs (CS 681)

Collection and Stream APIs in Java

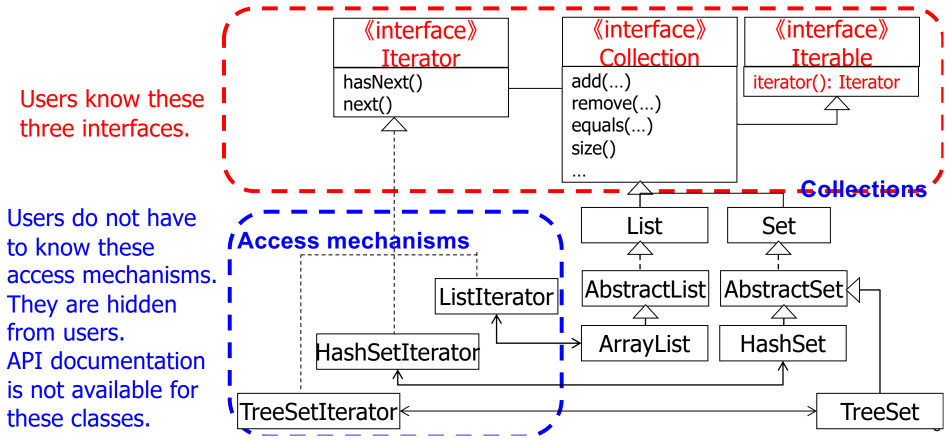

Stream API
Java 8
Streams
Parallel Streams
Collection API
Java 8
LE-compliant Methods
(Default and static interface methods for collection processing)
Java 5
Ease-of-Use Enhancement
(Generics, for-each statement, auto-boxing, etc.)
Java 2
New/Better Collections
(ArrayList, LinkedList, HashMap, Iterator, Collections, Arrays, Comparator, etc.)
Java 1
Early-stage Collections
(Vector, Hashtable, etc.)
No longer encouraged to use
CS681
CS680

Make Sure to Understand…

- Major collection types
  - List, Queue, Dequeue, Set, Map, etc.
- Differences between `ArrayList` and `LinkedList`

Lambda Expressions for Iteration

- **`java.lang.Iterable<T>`**
  - Has default methods since Java 8.
  - c.f. *Iterator*



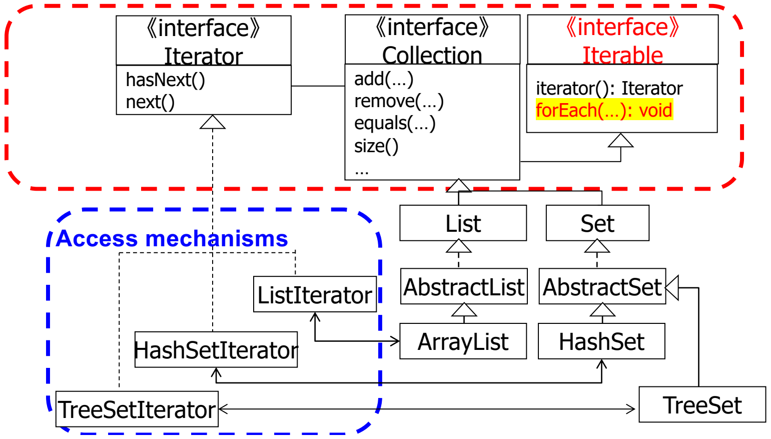
- Before Java 8…

```
– ArrayList<String> strList = new ArrayList<>();
  strList.add("a"); strList.add("b");

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

```
• for(String str: strList){
      System.out.println(str) }
```

  – Note: "for-each" is a syntactic sugar for iterator-based code.

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



- **`Iterable<T>`**
  - `default void forEach(Consumer<T> action)`

- **`Consumer<T>`**: Functional interface
  - Represents a function (LE) that accepts a parameter (T) and returns no result.
    - The LE receives a collection element as a parameter (T) and specifies an action to be applied to that element in its code block.

```
ArrayList<String> strList = new ArrayList<>();
strList.add("a"); strList.add("b");
strList.forEach( (String s)->System.out.println(s) );
```

- Without a lambda expression

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

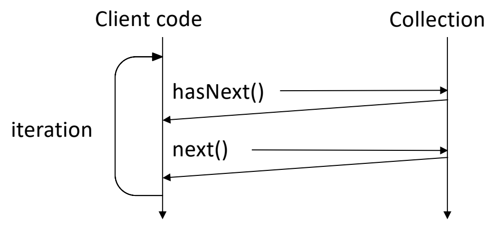
  -
    ```
    for(String str: strList){
        System.out.println(str) }
    ```
- With a lambda expression

  -
    ```
    strList.forEach( (Integer i)->System.out.println(i) )
    ```
  - Alternatively, with a method reference:
    - `strList.forEach( System.out::println )`

Traditional Way of Collection Processing

- ***External*** iteration:
  - Iterates over a collection outside of the collection and
  - Performs an operation on each element in turn outside of the collection.

  -
    ```
    Iterator<ArrayList> iterator = strList.iterator();
    while( iterator.hasNext() ) {
     System.out.print( iterator.next() ); }
    ```
  - Need to write a boilerplate code whenever you need to iterate over a collection.



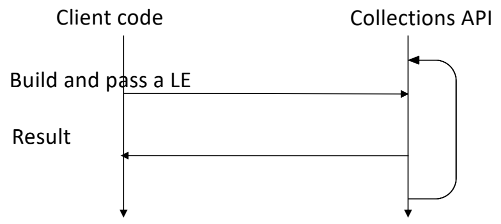
- The loop mixes up *what you want to do on a collection* and *how you do it*.
  - "How" is often emphasized than "what." (Or, "what" is often obscured by "how.")

  -
    ```
    Iterator<ArrayList> iterator = strList.iterator();
    while( iterator.hasNext() ) {
     System.out.print(iterator.next()); }
    ```
- Inherently serial
  - Hard to make it concurrent/parallel.

New Way of Collection Processing

- ***Internal*** iteration:
  - `forEach()`: Plays a similar role to the call of `iterator()`
    - Does not return an `Iterator`, which externally controls an iteration
    - Creates an equivalent object, which exists inside of the collection.
    - Uses the iterator-like object to perform iteration

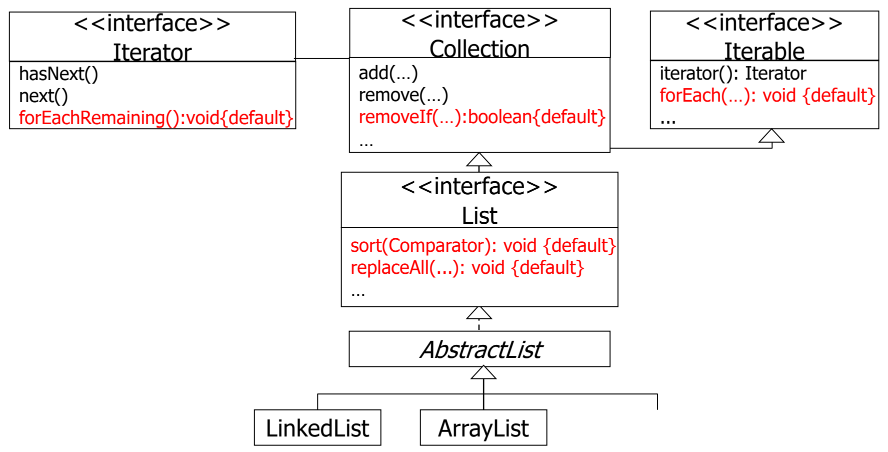
  -
    ```
    strList.forEach( (String i)->System.out.println(i) )
    ```



- 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

New Default Methods for Lists

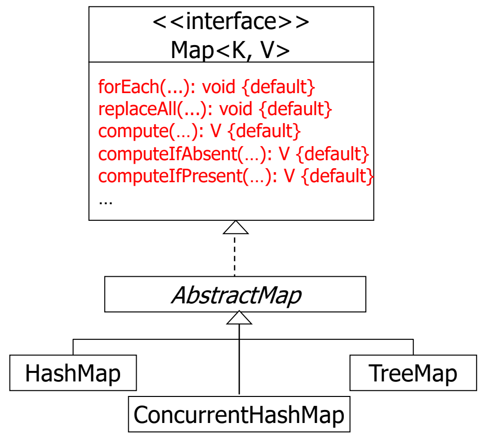
- Default methods have been added to various interfaces for lists.
  - Accept lambda expressions



```
ArrayList<String> list = Arrays.asList(
                                   “Yahoo”,“Yahooo”,”Yahoooo”));
// Print out each element
list.forEach( (String s)-> System.out.println(s) );
list.forEach(System.out::println);

// Sort elements based on the length of each element (in descending order)
// Result: Yahoooo, Yahooo, Yahoo
list.sort( (String s1, String s2)-> s2.length()-s1.length() );
list.sort( Comparator.comparing( (String s)-> s.length(),
                                 Comparator.reverseOrder() );
list.sort( Comparator.comparing(String::length).reversed());

// Replace each element with the one returned by a given lambda expression
// Result: YAHOOOO, YAHOOO, YAHOO
list.replaceAll( (String s)-> s.toUpperCase() );
list.replaceAll( String::toUpperCase );

// Remove every element that matches a criterion defined in a given lambda
expression
// Result: YAHOOO, YAHOO
list.removeIf( (String s)-> s.endsWith(“OOOO”) );
```

New Default Methods for Maps

- Default methods have been added to `java.util.Map<K, V>`
  - Accept lambda expressions



<<interface>> Map<K, V>
forEach(…): void {default}
replaceAll(…): void {default}
compute(…): V {default}
computeIfAbsent(…): V {default}
computeIfPresent(…): V {default}
…
AbstractMap
HashMap
TreeMap
ConcurrentHashMap

- **`forEach(LE)`**
  - Perform an action, which is defined as a given lambda expression, on each element (each key-value pair).

  -
    ```
    HashMap<String,Integer> map = new HashMap<>();
    map.put("A",1); map.put("B",2); map.put("C",3);

    // Print out each element
    // Result: A-1, B-2, C-3
    map.forEach((String key,Integer val)->
                    System.out.println(key + "-" + val));
    ```
- **`replaceAll(LE)`**
  - Replace each element with the one returned by a given lambda expression

  -
    ```
    // Result: A-10, B-20, C-30
    map.replaceAll((String key,Integer val)-> val*10);
    ```
- **`compute(key, LE)`**
  - Pair a key with a value that a given lambda expression returns and add the key-value pair.

  -
    ```
    // Result: A-9, B-20, C-30
    map.compute("A", (String key,Integer val)->{
                         if(val<0){returns 0;}
                         else{returns --val;} });
    ```

– Just in case, note that:

-
  ```
  int i, x, y = 0;
  i--;             // i==-1
  int x = i--;     // i==-1, x==0
  int y = --i;     // i==-1, y==-1
  ```

- **`computeIfAbsent(key, LE)`**
  - Pair a key with a value that a given lambda expression returns, ONLY IF the key does not exist, and add the key-value pair.

  -
    ```
    // Result: A-9, B-20, C-30, D-4
    map.computeIfAbsent("D", (String key)-> 4);
    ```
- **`computeIfPresent(key, LE)`**
  - Pair a key with a value that a given lambda expression returns, ONLY IF the key does exist, and replace an existing key-value pair with the new pair.

  -
    ```
    // Result: A-900, B-20, C-30, D-4
    map.computeIfPresent("A", (String key,Integer val)->
                                                  val*100);
    ```