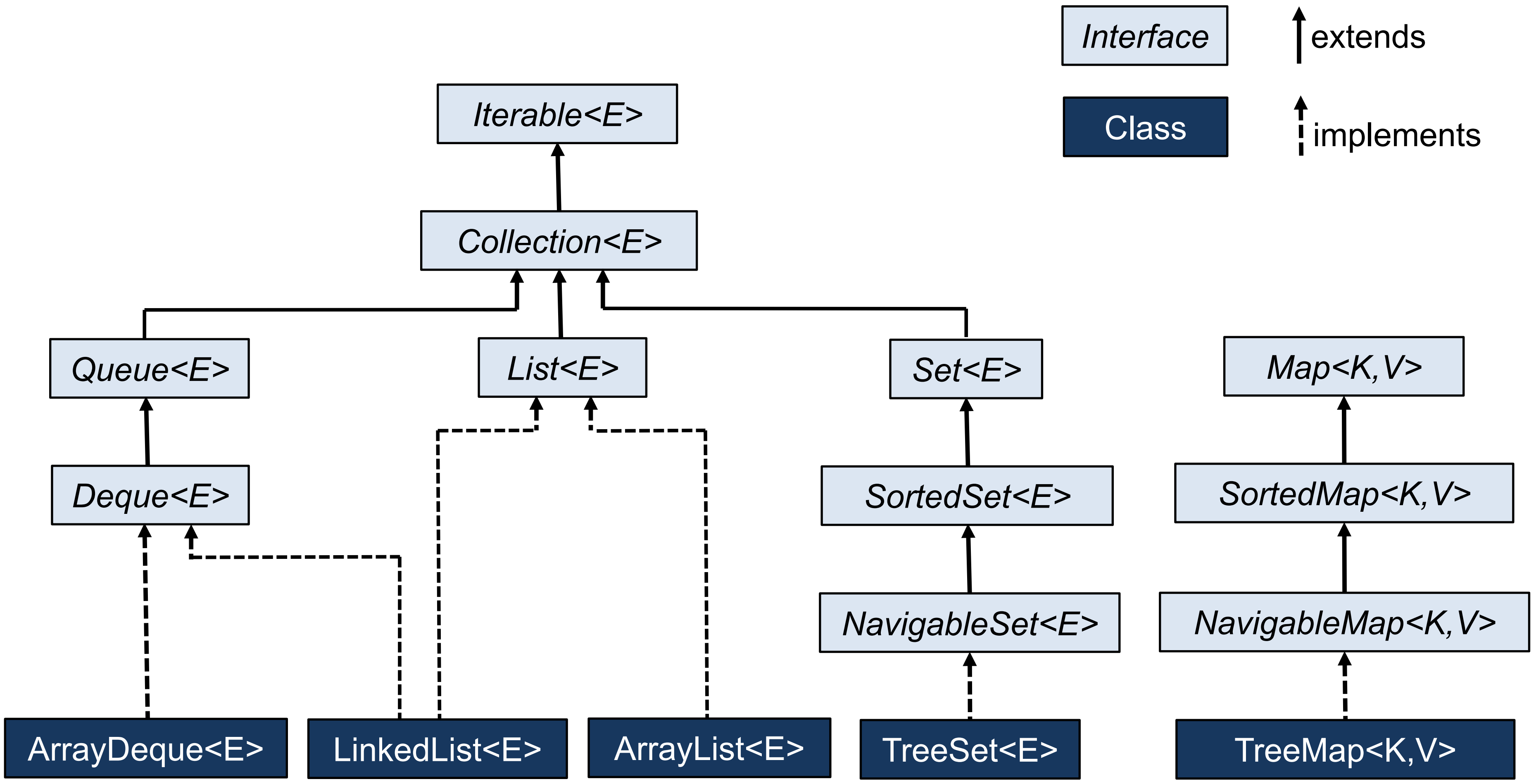
Java API Extract: Collection, Map, Functional Interfaces and Streams

(JDK 1.2 or later; presentation is not complete!)

- Overview of Collection and Map Types
- Iterable, Iterator and ListIterator
- Collection
- List
- Queue
- Deque
- Set, SortedSet and NavigableSet
- Map, Map.Entry, SortedMap and NavigableMap
- TreeSet and TreeMap
- Comparable and Comparator
- Functional Interfaces in package java.util.function
  - Predicate and BiPredicate,
  - Function, BiFunction, UnaryOperator and BinaryOperator
  - Consumer, BiConsumer and Supplier
- Streams
  - Creating Streams
  - Intermediate Stream Operations
  - Terminal Stream Operations

Overview of Collection and Map Types



Iterable<E>, Iterator<E> and ListIterator<E>

```
public interface Iterable<E> {
        Iterator<E> iterator();                              // returns an iterator over elements of type E
        default void forEach(Consumer<? super T> action)     // Performs the given action for each element of the Iterable
}
```

```
public interface Iterator<E> {
        boolean hasNext();        // returns true if the iteration has more elements
        E next();                 // returns the next element in the iteration
        default void remove();    // removes the last element returned by the iteration
                                  // The default implementation throws an instance of UnsupportedOperationException.
}
```

```
public interface ListIterator<E> extends Iterator<E> {
        boolean hasPrevious();    // returns true if this list has further elements in the reverse direction
        E previous();             // returns the previous element in the list
        void add(E e);            // inserts e into the list immediately before the next element
        int nextIndex();          // returns the index of the next element in the list
        int previousIndex();      // returns the index of previous element in the list
        void set(E e);            // Replaces the last element returned by next() or previous() with the specified element
}
```

Collection<E>

```
public interface Collection<E> extends Iterable<E> {

   boolean add(E e);                               // adds the element e 1)
   boolean addAll(Collection<? extends E> c);      // adds all of the elements in c to this collection 1)
   boolean remove(Object o);                       // removes the element o 1)
   boolean removeAll(Collection<?> c)              // removes all elements of this collection that are contained in c 1)
   boolean retainAll(Collection<?> c);             // removes all elements of this collection that are not contained in c 1)
   void clear();                                   // removes all elements

   boolean contains(Object o);                     // returns true if o is present
   boolean containsAll(Collection<?> c);           // returns true if all elements of c are present
   boolean isEmpty();                              // returns true if no element is present
   int size();                                     // returns the number of elements

   Iterator<E> iterator();                         // returns an Iterator over the elements
   Object[ ] toArray();                            // copy contents to an Object[ ]
   <T> T[ ] toArray(T[ ] t);                       // copy contents to a T[ ] for any T

   default boolean removeIf(Predicate<? super E> filter)     // Removes all of the elements of this collection
                                                             // that satisfy the given predicate. 1)
   default Stream<E> stream()                      // Returns a sequential Stream with this collection as its source.
}
```

[1)] The methods add, addAll, remove, removeAll, retainAll and removeIf return true, if the collection changes as a result of the call.

List<E>

```
public interface List<E> extends Collection<E> {

    boolean add(E e);                                          // adds the element e at the end of this list
    void add(int idx, E e);                                    // adds element e at index idx
    boolean addAll(Collection<? extends E> c);                 // adds the elements of c at the end of this list
    boolean addAll(int idx, Collection<? extends E> c);        // adds contents of c at index idx;
                                                               // returns true if this list changed as a result of the call.

    E set(int idx, E x);                                       // replaces element e at index idx by x; returns old value.
    E get(int idx);                                            // returns element at index idx
    E remove(int idx);                                         // removes and returns element at index idx

    int indexOf(Object o);                                     // returns index of first occurrence of o
    int lastIndexOf(Object o);                                 // returns index of fierst occurrence of o
    List<E> subList(int fromIdx, int toIdx);                   // returns a view of a portion of the list from fromIdx inclusive to
                                                               // toIdx exclusive

    ListIterator<E> listIterator();                            // returns a ListIterator over the elements
    ListIterator<E> listIterator(int index);                   // returns a ListIterator over the elements initially positioned at index idx

    default void replaceAll(UnaryOperator<E> operator)         // Replaces each element of this list with the result of
                                                               // applying the operator to that element.
    default void sort(Comparator<? super E> c)                 // Sorts this list according to the order induced by
                                                               // the specified Comparator.
}
```

Queue<E>

```
public interface Queue<E> extends Collection<E> {

        // Methods with boolean or null as return value:
        boolean offer(E e);       // adds the element e at the end of this queue
                                  // or returns false if this operation is not possible.
        E peek();                 // Retrieves, but does not remove, the head of this queue
                                  // or returns null if this queue is empty.
        E poll();                 // Retrieves and removes the head of this queue,
                                  // or returns null if this queue is empty.

        // Methods that throw an exception if operation is not possible:
        boolean add(E e);         // adds the element e at the end of this queue.
        E element();              // Retrieves, but does not remove, the head of this queue.
        E remove()                // Retrieves and removes the head of this queue.
}
```

Deque<E>

```
public interface Deque<E> extends Queue<E> {
        // Deque operations with boolean or null as return value:
        boolean offerFirst(E e);
        E peekFirst();
        E pollFirst();
        boolean offerLast(E e);
        E peekLast();
        E pollLast();

        // Deque operations that throw an exception if operation is not possible:
        void addFirst(E e);
        E getFirst();
        E removeFirst();
        void addLast(E e);
        E getLast();
        E removeLast();

        // Stack operations:
        void push(E e);         // equivalent to addFirst(e)
        E pop();                // equivalent to removeFirst()
        E peek();               // equivalent to peekFirst(e)

        // Others:
        boolean removeFirstOccurrence(Object o) ;
        boolean removeLastOccurrence(Object o) ;
}
```

Set<E>, SortedSet<E> and NavigableSet<E>

```
public interface Set<E> extends Collection<E> {
}
```

```
public interface SortedSet<E> extends Set<E> {
        Comparator<? super E> comparator();
        SortedSet<E> subSet(E fromElementInclusive, E toElementExclusive);      // returns a range view.
        SortedSet<E> headSet(E toElementExclusive );                            // returns a range view.
        SortedSet<E> tailSet(E fromElementInclusive);                           // returns a range view.
        E first();
        E last();
}
```

```
public interface NavigableSet<E> extends SortedSet<E> {
        E lower(E e);         // greatest element less than e, or null if there is no such element
        E higher(E e);        // least element greater than e, or null if there is no such element
        E floor(E e);         // greatest element less than or equal to e, or null if there is no such element
        E ceiling(E e);       // least element greater than or equal to e, or null if there is no such element
        E pollFirst();
        E pollLast();
        NavigableSet<E> descendingSet();          // returns a reverse-order view.
        Iterator<E> descendingIterator();         // returns a reverse-order iterator.
        NavigableSet<E> subSet(E fromElement, boolean fromInclusive, E toElement, boolean toInclusive);
        NavigableSet<E> headSet(E toElement, boolean inclusive);
        NavigableSet<E> tailSet(E fromElement, boolean inclusive);
}
```

Map<K,V>

```
public interface Map<K, V> {

    V put(K key, V value);                    // adds or replaces a key-value-pair.
                                              // returns the old value if the key was present; otherwise null

    void putAll(Map<? extends K, ? extends V> m);          // puts all key-value-pairs of m in this map.

    void clear();                             // removes all key-value-pairs
    V remove(Object key) ;                    // removes key-value-pair. Returns the value with which key was associated, or null

    V get(Object key);                        // returns the value corresponding to key, or null if key is not present
    boolean containsKey(Object key);          // returns true if key is present in the map
    boolean containsValue(Object value);      // returns true if value is present in the map
    boolean isEmpty();                        // true if no key-value-pair is present
    int size();                                // number of key-value-pairs

    Set<Map.Entry<K, V>> entrySet();          // returns a Set view of the key-value-pairs
    Set<K> keySet();                          // returns a Set view of the keys
    Collection<V> values();                   // returns a Collection view of the values

    default void forEach(BiConsumer<? super K, ? super V> action)
                                              // Performs the given action for each entry in this map
                                              // until all entries have been processed or the action throws an exception.

    default void replaceAll(BiFunction<? super K, ? super V, ? extends V> function)
                                              // Replaces each entry's value with the result of invoking the given function on that
                                              // entry until all entries have been processed or the function throws an exception.
}
```

Map.Entry<K,V> and SortedMap<K,V>

```
public interface Map.Entry<K, V> {

        K getKey()                      // Returns the key corresponding to this entry.
        V getValue()                    // Returns the value corresponding to this entry.
        int hashCode()                  // Returns the hash code value for this map entry.
        V setValue(V value)             // Replaces the value corresponding to this entry with the specified value.

}
```

```
public interface SortedMap<K, V> extends Map<K, V> {
        Comparator<? super K> comparator();
        SortedMap<K, V> subMap(K fromKeyInclusive, K toKeyExclusive);     // returns a range view.
        SortedMap<K, V> headMap(K toKeyExclusive);                        // returns a range view.
        SortedMap<K, V> tailMap(K fromKeyInclusive);                      // returns a range view.
        K firstKey();
        K lastKey();
}
```

NavigableMap<K,V>

```
public interface NavigableMap<K, V> extends SortedMap<K, V> {


        Map.Entry<K,V> pollFirstEntry();
        Map.Entry<K,V> pollLastEntry();
        Map.Entry<K,V> firstEntry();
        Map.Entry<K,V> lastEntry();

        Map.Entry<K,V> lowerEntry(K k);          // greatest entry less than k (or null)
        Map.Entry<K,V> higherEntry(K k);         // least entry greater than k (or null)
        Map.Entry<K,V> floorEntry(K k);          // greatest entry less than or equal to k (or null)
        Map.Entry<K,V> ceilingEntry(K k);  /     // least entry greater than or equal to k (or null)
        K lowerKey(K key);                       // greatest key less than k (or null)
        K higherKey(K key);                      // least key greater than k (or null)
        K floorKey (K key);                      // greatest key less than or equal to k (or null)
        K ceilingKey (K key);                    // least key greater than or equal to k (or null)

        NavigableMap<K, V> descendingMap();      // returns a reverse-order view of the map.
        NavigableSet<K> descendingKeySet();      // returns a reverse-order navigable key set view.
        NavigableSet<K> navigableKeySet();       // returns a forward-order navigable key set view.

        NavigableMap<K, V> subMap(K fromKey, boolean fromInclusive, K toKey, boolean toInclusive);
        NavigableMap<K, V> headMap(K toKey, boolean inclusive);
        NavigableMap<K, V> tailMap(K fromKey, boolean inclusive);
}
```

TreeSet<E> and TreeMap<K,V>

```
public class TreeSet<E> implements NavigableSet<E> {

        public TreeSet() {...}
        public TreeSet(Comparator<? super E> comparator) {...}
        public TreeSet(Collection<? extends E> c)
        public TreeSet(SortedSet<E> s) {...}
}
```

```
public class TreeMap<K, V> implements NavigableMap<K, V> {

        public TreeMap() {...}
        public TreeMap(Comparator<? super K> comparator) {...}
        public TreeMap(Map<? extends K, ? extends V> m) {...}
        public TreeMap(SortedMap<K, ? extends V> m) {...}
}
```

Comparable<E> and Comparator<E>

```
public interface Comparable<E> {
    int compareTo(E x);                  // returns a negative integer, zero, or a positive integer as this object is
                                         // less than, equal to, or greater than object x.
}
```

```
@FunctionalInterface
public interface Comparator<E> {
    int compare(E x, E y);               // returns a negative integer , zero, or a positive integer as object x is
                                         // less than, equal to, or greater than object y.

    default Comparator<E> reversed()     // returns a comparator that imposes the reverse ordering of this comparator.

    static <E,U extends Comparable<? super U>> Comparator<T> comparing(Function<? super E, ? extends U> keyExtractor)
                                         // Accepts a function that extracts a Comparable sort key from a type E,
                                         // and returns a Comparator<E> that compares by that sort key.
}
```

Predicate<T> and BiPredicate<T>

```
@FunctionalInterface
public interface Predicate<T> {

   boolean test(T t)                                   // Evaluates this predicate on the given argument.

   default Predicate<T> and(Predicate<? super T> other)  // Returns a composed predicate that represents a short-
                                                       // circuiting logical AND of this predicate and another.
   default Predicate<T> negate()                       // Returns a predicate that represents the logical negation
                                                       // of this predicate.
   default Predicate<T> or(Predicate<? super T> other)   // Returns a composed predicate that represents a short-
                                                       // circuiting logical OR of this predicate and another.
}
```

```
@FunctionalInterface
public interface BiPredicate<T, U> {

   boolean test(T t, U u)                                           // Evaluates this predicate on the given arguments.

   default BiPredicate<T,U> and(BiPredicate<? super T,? super U> other)  // Returns a composed predicate that represents a short-
                                                                    // circuiting logical AND of this predicate and another.
   default BiPredicate<T,U> negate()                                // Returns a predicate that represents the logical negation
                                                                    // of this predicate.
   default Predicate<T> or(BiPredicate<? super T,? super U> other)    // Returns a composed predicate that represents a short-
                                                                    // circuiting logical OR of this predicate and another.
}
```

Function<T,R>, BiFunction<T,U,R>, UnaryOperator<T>, BinaryOperator<T>

```
@FunctionalInterface
public interface Function<T,R> {
    R apply(T t)                    // Applies this function to the given argument.
    default <V> Function<T,V> andThen(Function<? super R,? extends V> after)
                                    // Returns a composed function that first applies this function to its input,
                                    // and then applies the after function to the result.
    default <V> Function<V,R> compose(Function<? super V,? extends T> before)
                                    // Returns a composed function that first applies the before function to its input,
                                    // and then applies this function to the result.
}
```

```
@FunctionalInterface
public interface BiFunction<T, U,R> {
    R apply(T t, U u)               // Applies this function to the given arguments.
    default <V> BiFunction<T,U,V> andThen(Function<? super R,? extends V> after)
                                    // Returns a composed function that first applies this function to its input,
                                    // and then applies the after function to the result.
}
```

```
@FunctionalInterface
public interface UnaryOperator<T> extends Function<T,T> { }
```

```
@FunctionalInterface
public interface BinaryOperator<T> extends Function<T,T,T> { }
```

Consumer<T>, BiConsumer<T,U> and Supplier<T>

```
@FunctionalInterface
public interface Consumer<T> {
    void accept(T t)           // Performs this operation on the given argument.
    default Consumer<T> andThen(Consumer<? super T> after)
                               // Returns a composed Consumer that performs, in sequence,
                               // this operation followed by the after operation.
}
```

```
@FunctionalInterface
public interface BiConsumer<T, U> {
    void accept(T t, U u)      //  Performs this operation on the given arguments.
    default BiConsumer<T,U> andThen(BiConsumer<? super T,? super U> after)
                               // Returns a composed BiConsumer that performs,
                               // in sequence, this operation followed by the after operation.
}
```

```
@FunctionalInterface
public interface Supplier<T> {
    T get()                    // Gets a result.
}
```

Creating Streams (1)

- Streams can be obtained from collections, arrays and files, e.g.:

default Stream<T> stream() **default** Stream<T> parallelStream()	Default methods from Collection<T>. Returns a sequential and parallel stream, respectively, with this collection as its source.
static <T> Stream<T> stream(T[ ] a)	Static method from Arrays. Returns a sequential Stream with the specified array a as its source.
Stream<String> lines()	Method from BufferedReader. Returns a Stream, the elements of which are lines read from this BufferedReader.

- Random streams can be obtained from class Random, e.g.:

DoubleStream doubles()	Returns an effectively unlimited stream of pseudorandom double values, each between zero (inclusive) and one (exclusive).
IntStream ints()	Returns an effectively unlimited stream of pseudorandom int values.

Creating Streams (2)

- Different static factory methods from the stream classes Stream<T>, IntStream, DoubleStream, LongStream etc. in package java.util.stream, e.g.:

static <T> Stream<T> empty()	Returns an empty sequential stream.
static <T> Stream<T> of(T ... values)	Returns a sequential ordered stream whose elements are the specified values.
static <T> Stream<T> generate(Supplier<T> s)	Returns an infinite sequential unordered stream where each element is generated by the provided Supplier s: s(), s(), s(), ...
static <T> Stream<T> iterate(T seed, UnaryOperator<T> f)	Returns an infinite sequential ordered Stream produced by iterative application of a function f to an initial element seed, producing a Stream consisting of seed, f(seed), f(f(seed)), ...
static IntStream range(**int** startInclusive, **int** endExclusive)	Returns a sequential ordered IntStream from startInclusive (inclusive) to endExclusive (exclusive) by an incremental step of 1.

Intermediate Stream Operations

- Intermediate stream operations are defined in package java.util.stream.
- Some intermediate methods from Stream<T>, e.g.:

Stream<T> filter(Predicate<? super T> predicate)	Returns a stream consisting of the elements of this stream that match the given predicate.
<R> Stream<R> map(Function<? super T,? extends R> m)	Returns a stream consisting of the results of applying the given function m to the elements of this stream.
<R> Stream<R> flatMap(Function<? super T, ? extends Stream<? extends R>> mapper)	Returns an stream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided function mapper to each element.
Stream<T> peek(Consumer<? super T> action)	Returns a stream consisting of the elements of this stream, additionally performing the provided action on each element.
Stream<T> sorted() Stream<T> sorted(Comparator<? super T> comparator)	Returns a stream consisting of the elements of this stream, sorted according to natural order or according to the provided Comparator.
Stream<T> distinct()	Returns a stream consisting of the distinct elements of this stream.
Stream<T> skip(**long** n)	Returns a stream consisting of the remaining elements of this stream after discarding the first n elements of the stream.
Stream<T> limit(**long** n)	Returns a stream consisting of the elements of this stream, truncated to be no longer than n in length.

Terminal Stream Operations (1)

- Terminal operations are defined in package java.util.stream.
- Logical operations from Stream<T> (with short-circuit evaluation), e.g.:

boolean anyMatch(Predicate<? super T> predicate)	Returns whether any elements of this stream match the provided predicate. Returns false if this stream is empty.
boolean allMatch(Predicate<? super T> predicate)	Returns whether all elements of this stream match the provided predicate. Returns true if this stream is empty.
boolean noneMatch(Predicate<? super T> predicate)	Returns whether no elements of this stream match the provided predicate. Returns true if this stream is empty.

- Reduction operations from Stream<T>, e.g.:

T reduce(T id, BinaryOperator<T> op)	Performs a reduction on the elements of this stream, using the provided identity value id and an associative accumulation function op, and returns the reduced value. A stream x_0, x_1, x_2, ... is reduced to (...(((id op x_0) op x_1) op x_2) op ...
long count()	Returns the count of elements in this stream.
Optional<T> min(Comparator<? super T> comparator) Optional<T> max(Comparator<? super T> comparator)	Returns the minimum or maximum element of this stream according to the provided Comparator. A Optional<T> is a container object which may or may not contain a non-null value. If a value is present, isPresent() will return true and get() will return the value.

Terminal Stream Operations (2)

- Some reduction operations from IntStream, e.g.:

int count() **int** sum() OptionalInt min(), OptionalInt max(), OptionalDouble average()	Returns the number, the sum, the minimum, the maximum and the average of this stream, respectively. An OptionalInt is a container object which may or may not contain a int value. If a value is present, isPresent() will return true and getAsInt() will return the value. OptionalDouble is defined analogously.

- Collect operation from Stream<T>:

collect(collector)	Accumulates stream values in mutable containers like the classes of the Java Collections Framework. For example, to collect all values of a String stream into a List, it can be written: List<String> aList = stringStream.collect(Collectors.toList()); Analogously, a String stream can be collected into a set: Set<String> aSet = stringStream.collect(Collectors.toSet());

- forEach operation from Stream<T>:

void forEach(Consumer<? super T> action)	Performs an action for each element of this stream.