Collection framework

Condensed Notes

May 4, 2025

1 Core Interfaces

- there are seven core interfaces in the java collection framework

Interface	Description
Collection	Enables you to work with groups of objects; it is at the top of the collections hierarchy.
Deque	Extends **Queue** to handle a double-ended queue.
List	Extends **Collection** to handle sequences (lists of objects).
NavigableSet	Extends **SortedSet** to handle retrieval of elements based on closest-match searches.
Queue	Extends **Collection** to handle special types of lists in which elements are removed only from the head.
Set	Extends **Collection** to handle sets, which must contain unique elements.
SortedSet	Extends **Set** to handle sorted sets.

2 Collection Interface

- core interface upon which collection framework is built
- implemented by any class that defines collections
- extends Iterable interface which means that it can be cycled through for-each loop

2.1 Methods defined

Method	Description
boolean add(E *obj*)	Adds *obj* to the invoking collection. Returns **true** if *obj* was added to the collection. Returns **false** if *obj* is already a member of the collection and the collection does not allow duplicates.
boolean addAll(Collection<? extends E> *c*)	Adds all the elements of *c* to the invoking collection. Returns **true** if the collection changed (i.e., the elements were added). Otherwise, returns **false**.
void clear( )	Removes all elements from the invoking collection.
boolean contains(Object *obj*)	Returns **true** if *obj* is an element of the invoking collection. Otherwise, returns **false**.
boolean containsAll(Collection<?> *c*)	Returns **true** if the invoking collection contains all elements of *c*. Otherwise, returns **false**.
boolean equals(Object *obj*)	Returns **true** if the invoking collection and *obj* are equal. Otherwise, returns **false**.
int hashCode( )	Returns the hash code for the invoking collection.
boolean isEmpty( )	Returns **true** if the invoking collection is empty. Otherwise, returns **false**.
Iterator<E> iterator( )	Returns an iterator for the invoking collection.
default Stream<E> parallelStream( )	Returns a stream that uses the invoking collection as its source for elements. If possible, the stream supports parallel operations.
boolean remove(Object *obj*)	Removes one instance of *obj* from the invoking collection. Returns **true** if the element was removed. Otherwise, returns **false**.
boolean removeAll(Collection<?> *c*)	Removes all elements of *c* from the invoking collection. Returns **true** if the collection changed (i.e., elements were removed). Otherwise, returns **false**.
default boolean removeIf(Predicate<? super E> *predicate*)	Removes from the invoking collection those elements that satisfy the condition specified by *predicate*.

boolean retainAll(Collection<?> *c*)	Removes all elements from the invoking collection except those in *c*. Returns **true** if the collection changed (i.e., elements were removed). Otherwise, returns **false**.
int size( )	Returns the number of elements held in the invoking collection.
default Spliterator<E> spliterator( )	Returns a spliterator to the invoking collections.
default Stream<E> stream( )	Returns a stream that uses the invoking collection as its source for elements. The stream is sequential.
default <T> T[ ] toArray(IntFunction<T[ ]> *arrayGen*)	Returns an array of the elements from the invoking collection. The returned array is created by the function specified by *arrayGen*. An **ArrayStoreException** is thrown if any collection element has a type that is not compatible with the array type. (Added by JDK 11.)
Object[ ] toArray( )	Returns an array of the elements from the invoking collection.
<T> T[ ] toArray(T *array*[ ])	Returns an array of the elements from the invoking collection. If the size of *array* equals the number of elements, these are returned in *array*. If the size of *array* is less than the number of elements, a new array of the necessary size is allocated and returned. If the size of *array* is greater than the number of elements, the array element following the last collection element is set to **null**. An **ArrayStoreException** is thrown if any collection element has a type that is not compatible with the array type.

3 List Interface

- extends Collection
- declares the behavior of a collection that stores a sequence of elements
- Elements can be inserted or accessed by their position in the list, using a zero-based index
- A list may contain duplicate elements

3.1 Methods defined

Method	Description
void add(int *index*, E *obj*)	Inserts *obj* into the invoking list at the index passed in *index*. Any preexisting elements at or beyond the point of insertion are shifted up. Thus, no elements are overwritten.
boolean addAll(int *index*, Collection<? extends E> *c*)	Inserts all elements of *c* into the invoking list at the index passed in *index*. Any preexisting elements at or beyond the point of insertion are shifted up. Thus, no elements are overwritten. Returns **true** if the invoking list changes and returns **false** otherwise.
static <E> List<E> copyOf(Collection<? extends E> *from*)	Returns a list that contains the same elements as that specified by *from*. The returned list is unmodifiable. Null values are not allowed.
E get(int *index*)	Returns the object stored at the specified index within the invoking collection.
int indexOf(Object *obj*)	Returns the index of the first instance of *obj* in the invoking list. If *obj* is not an element of the list, –1 is returned.
int lastIndexOf(Object *obj*)	Returns the index of the last instance of *obj* in the invoking list. If *obj* is not an element of the list, –1 is returned.
ListIterator<E> listIterator( )	Returns an iterator to the start of the invoking list.
ListIterator<E> listIterator(int *index*)	Returns an iterator to the invoking list that begins at the specified *index*.
static <E> List<E> of(*parameter-list*)	Creates an unmodifiable list containing the elements specified in *parameter-list*. Null elements are not allowed. Many overloaded versions are provided. See the discussion in the text for details.
E remove(int *index*)	Removes the element at position *index* from the invoking list and returns the deleted element. The resulting list is compacted. That is, the indexes of subsequent elements are decremented by one.
default void replaceAll(UnaryOperator<E> *opToApply*)	Updates each element in the list with the value obtained from the *opToApply* function.
E set(int *index*, E *obj*)	Assigns *obj* to the location specified by *index* within the invoking list. Returns the old value.
default void sort(Comparator<? super E> *comp*)	Sorts the list using the comparator specified by *comp*.
List<E> subList(int *start*, int *end*)	Returns a list that includes elements from *start* to *end*–1 in the invoking list. Elements in the returned list are also referenced by the invoking object.

4 Set Interface

- extends Collection
- specifies the behavior of a collection that does not allow duplicate elements
- it does not specify any additional methods of its own

only two additional methods are defined **of**: this is a factory method with a number of overloads which takes non-null objects as input and returns an un-modifiable set.

copyOf: this method takes a collection as input and makes an un-modifiable set out of it

5 SortedSet Interface

- extends Set
- eclares the behavior of a set sorted in ascending order

5.1 Methods defined

Method	Description
Comparator<? super E> comparator( )	Returns the invoking sorted set's comparator. If the natural ordering is used for this set, **null** is returned.
E first( )	Returns the first element in the invoking sorted set.
SortedSet<E> headSet(E *end*)	Returns a **SortedSet** containing those elements less than *end* that are contained in the invoking sorted set. Elements in the returned sorted set are also referenced by the invoking sorted set.
E last( )	Returns the last element in the invoking sorted set.
SortedSet<E> subSet(E *start*, E *end*)	Returns a **SortedSet** that includes those elements between *start* and *end*–1. Elements in the returned collection are also referenced by the invoking object.
SortedSet<E> tailSet(E *start*)	Returns a **SortedSet** that contains those elements greater than or equal to *start* that are contained in the sorted set. Elements in the returned set are also referenced by the invoking object.

6 NavigableSet Interface

- extends SortedSet
- declares the behavior of a collection that supports the retrieval of elements based on the closest match to a given value or values

6.1 Methods defined

Method	Description
E ceiling(E *obj*)	Searches the set for the smallest element *e* such that *e* >= *obj*. If such an element is found, it is returned. Otherwise, **null** is returned.
Iterator<E> descendingIterator( )	Returns an iterator that moves from the greatest to least. In other words, it returns a reverse iterator.
NavigableSet<E> descendingSet( )	Returns a **NavigableSet** that is the reverse of the invoking set. The resulting set is backed by the invoking set.
E floor(E *obj*)	Searches the set for the largest element *e* such that *e* <= *obj*. If such an element is found, it is returned. Otherwise, **null** is returned.
NavigableSet<E> headSet(E *upperBound*, boolean *incl*)	Returns a **NavigableSet** that includes all elements from the invoking set that are less than *upperBound*. If *incl* is **true**, then an element equal to *upperBound* is included. The resulting set is backed by the invoking set.
E higher(E *obj*)	Searches the set for the smallest element *e* such that *e* > *obj*. If such an element is found, it is returned. Otherwise, **null** is returned.
E lower(E *obj*)	Searches the set for the largest element *e* such that *e* < *obj*. If such an element is found, it is returned. Otherwise, **null** is returned.
E pollFirst( )	Returns the first element, removing the element in the process. Because the set is sorted, this is the element with the least value. **null** is returned if the set is empty.
E pollLast( )	Returns the last element, removing the element in the process. Because the set is sorted, this is the element with the greatest value. **null** is returned if the set is empty.
NavigableSet<E> subSet(E *lowerBound*, boolean *lowIncl*, E *upperBound*, boolean *highIncl*)	Returns a **NavigableSet** that includes all elements from the invoking set that are greater than *lowerBound* and less than *upperBound*. If *lowIncl* is **true**, then an element equal to *lowerBound* is included. If *highIncl* is **true**, then an element equal to *upperBound* is included. The resulting set is backed by the invoking set.
NavigableSet<E> tailSet(E *lowerBound*, boolean *incl*)	Returns a **NavigableSet** that includes all elements from the invoking set that are greater than *lowerBound*. If *incl* is **true**, then an element equal to *lowerBound* is included. The resulting set is backed by the invoking set.

7 Queue Interface

- extends Collection
- declares the behavior of a queue, which is often a first-in, first-out list

7.1 Methods defined

Method	Description
E element( )	Returns the element at the head of the queue. The element is not removed. It throws **NoSuchElementException** if the queue is empty.
boolean offer(E *obj*)	Attempts to add *obj* to the queue. Returns **true** if *obj* was added and **false** otherwise.
E peek( )	Returns the element at the head of the queue. It returns **null** if the queue is empty. The element is not removed.
E poll( )	Returns the element at the head of the queue, removing the element in the process. It returns **null** if the queue is empty.
E remove( )	Removes the element at the head of the queue, returning the element in the process. It throws **NoSuchElementException** if the queue is empty.

8 DeQueue Interface

- extends Queue
- declares the behavior of a double- ended queue.
- can function as standard, first-in, first-out queues or as last-in, first-out stacks.

8.1 Methods defined

Method	Description
void addFirst(E *obj*)	Adds *obj* to the head of the deque. Throws an **IllegalStateException** if a capacity-restricted deque is out of space.
void addLast(E *obj*)	Adds *obj* to the tail of the deque. Throws an **IllegalStateException** if a capacity-restricted deque is out of space.
Iterator<E> descendingIterator( )	Returns an iterator that moves from the tail to the head of the deque. In other words, it returns a reverse iterator.
E getFirst( )	Returns the first element in the deque. The object is not removed from the deque. It throws **NoSuchElementException** if the deque is empty.
E getLast( )	Returns the last element in the deque. The object is not removed from the deque. It throws **NoSuchElementException** if the deque is empty.

boolean offerFirst(E *obj*)	Attempts to add *obj* to the head of the deque. Returns **true** if *obj* was added and **false** otherwise. Therefore, this method returns **false** when an attempt is made to add *obj* to a full, capacity-restricted deque.
boolean offerLast(E *obj*)	Attempts to add *obj* to the tail of the deque. Returns **true** if *obj* was added and **false** otherwise.
E peekFirst( )	Returns the element at the head of the deque. It returns **null** if the deque is empty. The object is not removed.
E peekLast( )	Returns the element at the tail of the deque. It returns **null** if the deque is empty. The object is not removed.
E pollFirst( )	Returns the element at the head of the deque, removing the element in the process. It returns **null** if the deque is empty.
E pollLast( )	Returns the element at the tail of the deque, removing the element in the process. It returns **null** if the deque is empty.
E pop( )	Returns the element at the head of the deque, removing it in the process. It throws **NoSuchElementException** if the deque is empty.
void push(E *obj*)	Adds *obj* to the head of the deque. Throws an **IllegalStateException** if a capacity-restricted deque is out of space.
E removeFirst( )	Returns the element at the head of the deque, removing the element in the process. It throws **NoSuchElementException** if the deque is empty.
boolean removeFirstOccurrence(Object *obj*)	Removes the first occurrence of *obj* from the deque. Returns **true** if successful and **false** if the deque did not contain *obj*.
E removeLast( )	Returns the element at the tail of the deque, removing the element in the process. It throws **NoSuchElementException** if the deque is empty.
boolean removeLastOccurrence(Object *obj*)	Removes the last occurrence of *obj* from the deque. Returns **true** if successful and **false** if the deque did not contain *obj*.

9 Collection classes

The abstract and concrtete clases of the collection frame work are detailed below

Class	Description
AbstractCollection	Implements most of the **Collection** interface.
AbstractList	Extends **AbstractCollection** and implements most of the **List** interface.
AbstractQueue	Extends **AbstractCollection** and implements parts of the **Queue** interface.
AbstractSequentialList	Extends **AbstractList** for use by a collection that uses sequential rather than random access of its elements.
LinkedList	Implements a linked list by extending **AbstractSequentialList**.
ArrayList	Implements a dynamic array by extending **AbstractList**.
ArrayDeque	Implements a dynamic double-ended queue by extending **AbstractCollection** and implementing the **Deque** interface.
AbstractSet	Extends **AbstractCollection** and implements most of the **Set** interface.
EnumSet	Extends **AbstractSet** for use with **enum** elements.
HashSet	Extends **AbstractSet** for use with a hash table.
LinkedHashSet	Extends **HashSet** to allow insertion-order iterations.
PriorityQueue	Extends **AbstractQueue** to support a priority-based queue.
TreeSet	Implements a set stored in a tree. Extends **AbstractSet**.

10 ArrayList

- extends AbstractList and implements the List interface
- ArrayList supports dynamic arrays that can grow as needed
- Constructors:
  - ArrayList( ) : builds an empty arraylist
  - ArrayList(Collection$\langle ?extendsE\rangle$ c): builds a list from give collection
  - ArrayList(int capacity) : builds a list with specific initial capacity
- can pe passed to print functions because the underlying toString formats the collection nicely
- ensureCapacity(int cap) : increases the capacity
- trimToSize() : decreases capacity
- toArray function gives the array from arraylist
- signature of toArray
  - object[ ] toArray( )
  - $\langle T\rangle T[]toArray(Tarray[])$
  - $\langle T\rangle T[]toArray(IntFunction\langle T[]\rangle arrayGen)$

11 LinkedList class

- extends AbstractSequentialList and implements the List, Deque, and Queue interfaces
- Constructors
  - LinkedList( )
  - $LinkedList(Collection\langle ?extendsE\rangle c)$

12 HashSet class

- extends AbstractSet and implements the Set interface
- creates a collection that uses a hash table for storage.
- Constructors:
  - HashSet( )
  - HashSet(Collection$\langle ?extendsE\rangle$ c)
  - HashSet(int capacity)
  - HashSet(int capacity, float fillRatio)

13 LinkedHashSet class

- extends HashSet and adds no members of its own
- LinkedHashSet maintains a linked list of the entries in the set, in the order in which they were inserted. This allows insertion-order iteration over the set.

14 TreeSet class

- extends AbstractSet and implements the NavigableSet interface
- It creates a collection that uses a tree for storage
- Constructors:
  - TreeSet( )
  - TreeSet(Collection$\langle ?extendsE\rangle$ c)
  - TreeSet(Comparator$\langle ?superE\rangle$ comp)
  - TreeSet(SortedSet$\langle E\rangle$ ss)

15 PriorityQueue class

- extends AbstractQueue and implements the Queue interface
- It creates a queue that is prioritized based on the queue's comparator.
- Constructors:
  - PriorityQueue( )
  - PriorityQueue(int capacity)
  - PriorityQueue(Comparator$\langle ?superE\rangle$ comp)
  - PriorityQueue(int capacity,Comparator$\langle ?superE\rangle$ comp)
  - PriorityQueue(Collection$\langle ?extendsE\rangle$ c)
  - PriorityQueue(PriorityQueue$\langle ?extendsE\rangle$ c)
  - PriorityQueue(SortedSet$\langle ?extendsE\rangle$ c)
- Although you can iterate through a PriorityQueue using an iterator, the order of that iteration is undefined

16 ArrayDeque class

- extends AbstractCollection and implements the Deque interface
- Constructors
  - ArrayDeque( )
  - ArrayDeque(int size)
  - ArrayDeque(Collection$\langle ?extendsE\rangle$ c)

17 EnumSet class

- extends AbstractSet and implements Set
- it defines no constructors but has a bunch of factory methods given below

Method	Description
static <E extends Enum<E>> EnumSet<E> allOf(Class<E> *t*)	Creates an **EnumSet** that contains the elements in the enumeration specified by *t*.
static <E extends Enum<E>> EnumSet<E> complementOf(EnumSet<E> *e*)	Creates an **EnumSet** that is comprised of those elements not stored in *e*.
static <E extends Enum<E>> EnumSet<E> copyOf(EnumSet<E> *c*)	Creates an **EnumSet** from the elements stored in *c*.
static <E extends Enum<E>> EnumSet<E> copyOf(Collection<E> *c*)	Creates an **EnumSet** from the elements stored in *c*.
static <E extends Enum<E>> EnumSet<E> noneOf(Class<E> *t*)	Creates an **EnumSet** that contains the elements that are not in the enumeration specified by *t*, which is an empty set by definition.
static <E extends Enum<E>> EnumSet<E> of(E *v*, E ... *varargs*)	Creates an **EnumSet** that contains *v* and zero or more additional enumeration values.
static <E extends Enum<E>> EnumSet<E> of(E *v*)	Creates an **EnumSet** that contains *v*.
static <E extends Enum<E>> EnumSet<E> of(E *v1*, E *v2*)	Creates an **EnumSet** that contains *v1* and *v2*.
static <E extends Enum<E>> EnumSet<E> of(E *v1*, E *v2*, E *v3*)	Creates an **EnumSet** that contains *v1* through *v3*.
static <E extends Enum<E>> EnumSet<E> of(E *v1*, E *v2*, E *v3*, E *v4*)	Creates an **EnumSet** that contains *v1* through *v4*.
static <E extends Enum<E>> EnumSet<E> of(E *v1*, E *v2*, E *v3*, E *v4*, E *v5*)	Creates an **EnumSet** that contains *v1* through *v5*.
static <E extends Enum<E>> EnumSet<E> range(E *start*, E *end*)	Creates an **EnumSet** that contains the elements in the range specified by *start* and *end*.

18 Iterator

- Iterator and ListIterator interfaces are responsible for iteration
- Iterator enables cycling through a collection obtaining or removing elements
- ListIterator extends Iterator and enables bidirectional movement

18.1 methods by iterator

Method	Description
default void forEachRemaining(Consumer<? super E> *action*)	The action specified by *action* is executed on each unprocessed element in the collection.
boolean hasNext( )	Returns **true** if there are more elements. Otherwise, returns **false**.
E next( )	Returns the next element. Throws **NoSuchElementException** if there is not a next element.
default void remove( )	Removes the current element. Throws **IllegalStateException** if an attempt is made to call **remove( )** that is not preceded by a call to **next( )**. The default version throws an **UnsupportedOperationException**.

18.2 methods by ListItertors

Method	Description
void add(E *obj*)	Inserts *obj* into the list in front of the element that will be returned by the next call to **next( )**.
default void forEachRemaining(Consumer<? super E> *action*)	The action specified by *action* is executed on each unprocessed element in the collection.
boolean hasNext( )	Returns **true** if there is a next element. Otherwise, returns **false**.
boolean hasPrevious( )	Returns **true** if there is a previous element. Otherwise, returns **false**.
E next( )	Returns the next element. A **NoSuchElementException** is thrown if there is not a next element.
int nextIndex( )	Returns the index of the next element. If there is not a next element, returns the size of the list.
E previous( )	Returns the previous element. A **NoSuchElementException** is thrown if there is not a previous element.
int previousIndex( )	Returns the index of the previous element. If there is not a previous element, returns –1.
void remove( )	Removes the current element from the list. An **IllegalStateException** is thrown if **remove( )** is called before **next( )** or **previous( )** is invoked.
void set(E *obj*)	Assigns *obj* to the current element. This is the element last returned by a call to either **next( )** or **previous( )**.

18.3 Using an iterator

- call collections iterator method to obtain an iterator
- set up a loop which calls the iterators hasnext method for condition
- call next method of iterator to get the next element

19 Spliterators

- these are another type of iterators defined by Spliteretors interface
- A spliterator cycles through a sequence of elements, and in this regard, it is similar to the iterators
- it offers substantially more functionality than does either Iterator or ListIterator
- Spliterator supports parallel programming

Method	Description
int characteristics( )	Returns the characteristics of the invoking spliterator, encoded into an integer.
long estimateSize( )	Estimates the number of elements left to iterate and returns the result. Returns **Long.MAX_VALUE** if the count cannot be obtained for any reason.
default void forEachRemaining(Consumer<? super T> *action*)	Applies *action* to each unprocessed element in the data source.
default Comparator<? super T> getComparator( )	Returns the comparator used by the invoking spliterator or **null** if natural ordering is used. If the sequence is unordered, **IllegalStateException** is thrown.
default long getExactSizeIfKnown( )	If the invoking spliterator is sized, returns the number of elements left to iterate. Returns –1 otherwise.
default boolean hasCharacteristics(int *val*)	Returns **true** if the invoking spliterator has the characteristics passed in *val*. Returns **false** otherwise.
boolean tryAdvance(Consumer<? super T> *action*)	Executes *action* on the next element in the iteration. Returns **true** if there is a next element. Returns **false** if no elements remain.
Spliterator<T> trySplit( )	If possible, splits the invoking spliterator, returning a reference to a new spliterator for the partition. Otherwise, returns **null**. Thus, if successful, the original spliterator iterates over one portion of the sequence and the returned spliterator iterates over the other portion.

20 RandomAccess Interface

- The RandomAccess interface contains no members
- by implementing this interface, a collection signals that it supports efficient random access to its elements
- RandomAccess is implemented by ArrayList and by the legacy Vector class, among others

21 Map Interfaces

the following interfaces supports map operation

Interface	Description
Map	Maps unique keys to values.
Map.Entry	Describes an element (a key/value pair) in a map. This is an inner class of **Map**.
NavigableMap	Extends **SortedMap** to handle the retrieval of entries based on closest-match searches.
SortedMap	Extends **Map** so that the keys are maintained in ascending order.

22 Map Interface

- does not support null key or value

23 Map Methods

Method	Description
void clear( )	Removes all key/value pairs from the invoking map.
default V compute(K *k*, BiFunction<? super K, ? super V, ? extends V> *func*)	Calls *func* to construct a new value. If func returns non-**null**, the new key/value pair is added to the map, any preexisting pairing is removed, and the new value is returned. If *func* returns **null**, any preexisting pairing is removed, and **null** is returned.
default V computeIfAbsent(K *k*, Function<? super K, ? extends V> *func*)	Returns the value associated with the key *k*. Otherwise, the value is constructed through a call to *func* and the pairing is entered into the map and the constructed value is returned. If no value can be constructed, **null** is returned.
default V computeIfPresent(K *k*, BiFunction<? super K, ? super V, ? extends V> *func*)	If *k* is in the map, a new value is constructed through a call to *func* and the new value replaces the old value in the map. In this case, the new value is returned. If the value returned by *func* is **null**, the existing key and value are removed from the map and **null** is returned.
boolean containsKey(Object *k*)	Returns **true** if the invoking map contains *k* as a key. Otherwise, returns **false**.
boolean containsValue(Object *v*)	Returns **true** if the map contains *v* as a value. Otherwise, returns **false**.
static <K, V> Map<K, V> copyOf(Map<? extends K, ? extends V> *from*)	Returns a map that contains the same key/value pairs as that specified by *from*. The returned map is unmodifiable. Null keys or values are not allowed.
static <K, V> Map.Entry<K, V> entry(K *k*, V *v*)	Returns an unmodifiable map entry comprised of the specified key and value. A **null** key or value is not allowed.
Set<Map.Entry<K, V>> entrySet( )	Returns a **Set** that contains the entries in the map. The set contains objects of type **Map.Entry**. Thus, this method provides a set-view of the invoking map.
boolean equals(Object *obj*)	Returns **true** if *obj* is a **Map** and contains the same entries. Otherwise, returns **false**.
default void forEach(BiConsumer< ? super K, ? super V> *action*)	Executes *action* on each element in the invoking map. A **ConcurrentModificationException** will be thrown if an element is removed during the process.
V get(Object *k*)	Returns the value associated with the key *k*. Returns **null** if the key is not found.
default V getOrDefault(Object *k*, V *defVal*)	Returns the value associated with *k* if it is in the map. Otherwise, *defVal* is returned.
int hashCode( )	Returns the hash code for the invoking map.
boolean isEmpty( )	Returns **true** if the invoking map is empty. Otherwise, returns **false**.

Set<K> keySet( )	Returns a **Set** that contains the keys in the invoking map. This method provides a set-view of the keys in the invoking map.
default V merge(K *k*, V *v*, BiFunction<? super V, ? super V, ? extends V> *func*)	If *k* is not in the map, the pairing *k*,*v* is added to the map. In this case, *v* is returned. Otherwise, *func* returns a new value based on the old value, the key is updated to use this value, and **merge( )** returns this value. If the value returned by *func* is **null**, the existing key and value are removed from the map and **null** is returned.
static <K, V> Map<K, V> of(*parameter-list*)	Creates an unmodifiable map containing the entries specified in *parameter-list*. Null keys or values are not allowed. Many overloaded versions are provided. See the discussion in the text for details.
static <K, V> Map<K, V> ofEntries(Map.Entry<? extends K, ? extends V> ... *entries*)	Returns an unmodifiable map that contains the key/value mappings described by the entries passed in *entries*. Null keys or values are not allowed.
V put(K *k*, V *v*)	Puts an entry in the invoking map, overwriting any previous value associated with the key. The key and value are *k* and *v*, respectively. Returns **null** if the key did not already exist. Otherwise, the previous value linked to the key is returned.
void putAll(Map<? extends K, ? extends V> *m*)	Puts all the entries from *m* into this map.
default V putIfAbsent(K *k*, V *v*)	Inserts the key/value pair into the invoking map if this pairing is not already present or if the existing value is **null**. Returns the old value. The **null** value is returned when no previous mapping exists, or the value is **null**.
V remove(Object *k*)	Removes the entry whose key equals *k*.
default boolean remove(Object *k*, Object *v*)	If the key/value pair specified by *k* and *v* is in the invoking map, it is removed and **true** is returned. Otherwise, **false** is returned.
default boolean replace(K *k*, V *oldV*, V *newV*)	If the key/value pair specified by *k* and *oldV* is in the invoking map, the value is replaced by *newV* and **true** is returned. Otherwise **false** is returned.
default V replace(K *k*, V *v*)	If the key specified by *k* is in the invoking map, its value is set to *v* and the previous value is returned. Otherwise, **null** is returned.
default void replaceAll(BiFunction< ? super K, ? super V, ? extends V> *func*)	Executes *func* on each element of the invoking map, replacing the element with the result returned by *func*. A **ConcurrentModificationException** will be thrown if an element is removed during the process.

int size( )	Returns the number of key/value pairs in the map.
Collection<V> values( )	Returns a collection containing the values in the map. This method provides a collection-view of the values in the map.

24 SortedMap Interface

- SortedMap interface extends Map
- t ensures that the entries are maintained in ascending order based on the keys

25 methods

Method	Description
Comparator<? super K> comparator( )	Returns the invoking sorted map's comparator. If natural ordering is used for the invoking map, **null** is returned.
K firstKey( )	Returns the first key in the invoking map.
SortedMap<K, V> headMap(K *end*)	Returns a sorted map for those map entries with keys that are less than *end*.
K lastKey( )	Returns the last key in the invoking map.
SortedMap<K, V> subMap(K *start*, K *end*)	Returns a map containing those entries with keys that are greater than or equal to *start* and less than *end*.
SortedMap<K, V> tailMap(K *start*)	Returns a map containing those entries with keys that are greater than or equal to *start*.

26 NavigableMap Interface

- extends SortedMap
- declares the behavior of a map that supports the retrieval of entries based on the closest match to a given key or keys

27 Methods

Method	Description
Map.Entry<K,V> ceilingEntry(K *obj*)	Searches the map for the smallest key *k* such that *k* >= *obj*. If such a key is found, its entry is returned. Otherwise, **null** is returned.
K ceilingKey(K *obj*)	Searches the map for the smallest key *k* such that *k* >= *obj*. If such a key is found, it is returned. Otherwise, **null** is returned.
NavigableSet<K> descendingKeySet( )	Returns a **NavigableSet** that contains the keys in the invoking map in reverse order. Thus, it returns a reverse set-view of the keys. The resulting set is backed by the map.
NavigableMap<K,V> descendingMap( )	Returns a **NavigableMap** that is the reverse of the invoking map. The resulting map is backed by the invoking map.
Map.Entry<K,V> firstEntry( )	Returns the first entry in the map. This is the entry with the least key.
Map.Entry<K,V> floorEntry(K *obj*)	Searches the map for the largest key *k* such that *k* <= *obj*. If such a key is found, its entry is returned. Otherwise, **null** is returned.
K floorKey(K *obj*)	Searches the map for the largest key *k* such that *k* <= *obj*. If such a key is found, it is returned. Otherwise, **null** is returned.
NavigableMap<K,V> headMap(K *upperBound*, boolean *incl*)	Returns a **NavigableMap** that includes all entries from the invoking map that have keys that are less than *upperBound*. If *incl* is **true**, then an element equal to *upperBound* is included. The resulting map is backed by the invoking map.
Map.Entry<K,V> higherEntry(K *obj*)	Searches the set for the largest key *k* such that *k* > *obj*. If such a key is found, its entry is returned. Otherwise, **null** is returned.
K higherKey(K *obj*)	Searches the set for the largest key *k* such that *k* > *obj*. If such a key is found, it is returned. Otherwise, **null** is returned.
Map.Entry<K,V> lastEntry( )	Returns the last entry in the map. This is the entry with the largest key.
Map.Entry<K,V> lowerEntry(K obj)	Searches the set for the largest key *k* such that *k* < *obj*. If such a key is found, its entry is returned. Otherwise, **null** is returned.
K lowerKey(K *obj*)	Searches the set for the largest key *k* such that *k* < *obj*. If such a key is found, it is returned. Otherwise, **null** is returned.

NavigableSet<K> navigableKeySet( )	Returns a **NavigableSet** that contains the keys in the invoking map. The resulting set is backed by the invoking map.
Map.Entry<K,V> pollFirstEntry( )	Returns the first entry, removing the entry in the process. Because the map is sorted, this is the entry with the least key value. **null** is returned if the map is empty.
Map.Entry<K,V> pollLastEntry( )	Returns the last entry, removing the entry in the process. Because the map is sorted, this is the entry with the greatest key value. **null** is returned if the map is empty.
NavigableMap<K,V> subMap(K *lowerBound*, boolean *lowIncl*, K *upperBound* boolean *highIncl*)	Returns a **NavigableMap** that includes all entries from the invoking map that have keys that are greater than *lowerBound* and less than *upperBound*. If *lowIncl* is **true**, then an element equal to *lowerBound* is included. If *highIncl* is **true**, then an element equal to *highIncl* is included. The resulting map is backed by the invoking map.
NavigableMap<K,V> tailMap(K *lowerBound*, boolean *incl*)	Returns a **NavigableMap** that includes all entries from the invoking map that have keys that are greater than *lowerBound*. If *incl* is **true**, then an element equal to *lowerBound* is included. The resulting map is backed by the invoking map.

28 Map.Entry Interface

- Map.Entry interface enables you to work with a map entry

29 methods

Method	Description
boolean equals(Object *obj*)	Returns **true** if *obj* is a **Map.Entry** whose key and value are equal to that of the invoking object.
K getKey( )	Returns the key for this map entry.
V getValue( )	Returns the value for this map entry.
int hashCode( )	Returns the hash code for this map entry.
V setValue(V *v*)	Sets the value for this map entry to *v*. A **ClassCastException** is thrown if *v* is not the correct type for the map. An **IllegalArgumentException** is thrown if there is a problem with *v*. A **NullPointerException** is thrown if *v* is **null** and the map does not permit **null** keys. An **UnsupportedOperationException** is thrown if the map cannot be changed.

30 Map classes

these are the main map classes

Class	Description
AbstractMap	Implements most of the **Map** interface.
EnumMap	Extends **AbstractMap** for use with **enum** keys.
HashMap	Extends **AbstractMap** to use a hash table.
TreeMap	Extends **AbstractMap** to use a tree.
WeakHashMap	Extends **AbstractMap** to use a hash table with weak keys.
LinkedHashMap	Extends **HashMap** to allow insertion-order iterations.
IdentityHashMap	Extends **AbstractMap** and uses reference equality when comparing documents.

31 HashMap class

- extends AbstractMap and implements the Map interface
- uses a hash table to store the map. This allows the execution time of get( ) and put( ) to remain constant even for large sets
- Constructors:
  - HashMap( )
  - HashMap(Map$\langle ?extendsK, ?extendsV\rangle$ m)
  - HashMap(int capacity)
  - HashMap(int capacity, float fillRatio)

32 TreeMap class

- extends AbstractMap and implements the NavigableMap interface
- creates maps stored in a tree structure. A TreeMap provides an efficient means of storing key/value pairs in sorted order and allows rapid retrieval
- unlike a hash map, a tree map guarantees that its elements will be sorted in ascending key order
- Constructors:
  - TreeMap( )
  - TreeMap(Comparator$\langle ?superK\rangle$ comp)
  - TreeMap(Map$\langle ?extendsK, ?extendsV\rangle$ m)
  - TreeMap(SortedMap$\langle K, ?extendsV\rangle$ sm)

33 LinkedHashMap class

- extends HashMap
- maintains a linked list of the entries in the map, in the order in which they were inserted. This allows insertion-order iteration over the map
- Constructors:
  - LinkedHashMap( )
  - LinkedHashMap(Map$\langle ?extendsK, ?extendsV\rangle$ m)
  - LinkedHashMap(int capacity)
  - LinkedHashMap(int capacity, float fillRatio)
  - LinkedHashMap(int capacity, float fillRatio, boolean Order) : if order is true then acces order other wise insertion order

34 methods

- LinkedHashMap adds only one method to those defined by HashMap. This method is removeEldestEntry( )
- protected boolean removeEldestEntry(Map.Entry$\langle K, V\rangle$ e)
- This method is called by put( ) and putAll( ). The oldest entry is passed in e. By default, this method returns false and does nothing
- override this method, then you can have the LinkedHashMap remove the oldest entry in the map. To do this, have your override return true. To keep the oldest entry, return false

35 IdentityHashMap class

- extends AbstractMap and implements the Map interface
- similar to HashMap except that it uses reference equality when comparing elements : need to overview this point

36 EnumMap class

- extends AbstractMap and implements Map
- Constructors:
  - EnumMap(Class $\langle K\rangle$ kType)
  - EnumMap(Map$\langle K, ?extends V\rangle$ m)
  - EnumMap(EnumMap$\langle K, ?extends V\rangle$ em)

37 Comparator Interface

- can be implemented by a coustom class to define a compaerator
- passed in various sorting algos to get the benifit of algos even with our own comparators

38 Methods

the following methods may be defined by the implementing class

methods	remarks	reqired to be overriden
public int compare(T obj1, t obj 2)	returns 0 if equal posetive if obj1 ⟩obj2 negetive otherwise	yes
public boolean equals(T obj1)	returns true if the current comparator is equal to passed argument false otherwise	no
default Comparator$\langle T\rangle$ reversed()	returns a reversed comparator as current	no
static $\langle TextendsComparable\langle ?superT\rangle Comparator\langle T\rangle$ reverseOrder()	Returns a comparator that imposes the reverse of the natural ordering.	no
static $\langle TextendsComparable\langle ?superT\rangle Comparator\langle T\rangle$ naturalOrder()	Returns a comparator that compares Comparable objects in natural order.	no
static $\langle T\rangle Comparator\langle T\rangle$ nullsFirst(Comparator$\langle ?superT\rangle$ comparator)	Returns a null-friendly comparator that considers null to be less than non-null. When both are null, they are considered equal. If both are non-null, the specified Comparator is used to determine the order. If the specified comparator is null, then the returned comparator considers all non-null values to be equal.	no
static $\langle T\rangle Comparator\langle T\rangle$ nullsLast(Comparator$\langle ?superT\rangle$ comparator)	Returns a null-friendly comparator that considers null to be greater than non-null. When both are null, they are considered equal. If both are non-null, the specified Comparator is used to determine the order. If the specified comparator is null, then the returned comparator considers all non-null values to be equal.	no

Table 1: Your caption here

Table 1 shows my first longtable.

39 Collection Algorithms

The Collections Framework defines several algorithms that can be applied to collections and maps. These algorithms are defined as static methods within the Collections class.

Method	Description
static <T> boolean addAll(Collection <? super T> *c*, T... *elements*)	Inserts the elements specified by *elements* into the collection specified by *c*. Returns **true** if the elements were added and **false** otherwise.
static <T> Queue<T> asLifoQueue(Deque<T> *c*)	Returns a last-in, first-out view of *c*.
static <T> int binarySearch(List<? extends T> *list*, T *value*, Comparator<? super T> *c*)	Searches for *value* in *list* ordered according to *c*. Returns the position of value in *list*, or a negative value if *value* is not found.
static <T> int binarySearch(List<? extends Comparable<? super T>> *list*, T *value*)	Searches for *value* in *list*. The list must be sorted. Returns the position of *value* in *list*, or a negative value if *value* is not found.
static <E> Collection<E> checkedCollection(Collection<E> *c*, Class<E> *t*)	Returns a run-time type-safe view of a collection. An attempt to insert an incompatible element will cause a **ClassCastException**.
static <E> List<E> checkedList(List<E> *c*, Class<E> *t*)	Returns a run-time type-safe view of a **List**. An attempt to insert an incompatible element will cause a **ClassCastException**.
static <K, V> Map<K, V> checkedMap(Map<K, V> *c*, Class<K> *keyT*, Class<V> *valueT*)	Returns a run-time type-safe view of a **Map**. An attempt to insert an incompatible element will cause a **ClassCastException**.
static <K, V> NavigableMap<K, V> checkedNavigableMap(NavigableMap<K, V> *nm*, Class<E> *keyT*, Class<V> *valueT*)	Returns a run-time type-safe view of a **NavigableMap**. An attempt to insert an incompatible element will cause a **ClassCastException**.
static <E> NavigableSet<E> checkedNavigableSet(NavigableSet<E> *ns*, Class<E> *t*)	Returns a run-time type-safe view of a **NavigableSet**. An attempt to insert an incompatible element will cause a **ClassCastException**.
static <E> Queue<E> checkedQueue(Queue<E> *q*, Class<E> *t*)	Returns a run-time type-safe view of a **Queue**. An attempt to insert an incompatible element will cause a **ClassCastException**.

static <E> List<E> checkedSet(Set<E> *c*, Class<E> *t*)	Returns a run-time type-safe view of a **Set**. An attempt to insert an incompatible element will cause a **ClassCastException**.
static <K, V> SortedMap<K, V> checkedSortedMap(SortedMap<K, V> *c*, Class<K> *keyT*, Class<V> *valueT*)	Returns a run-time type-safe view of a **SortedMap**. An attempt to insert an incompatible element will cause a **ClassCastException**.
static <E> SortedSet<E> checkedSortedSet(SortedSet<E> *c*, Class<E> *t*)	Returns a run-time type-safe view of a **SortedSet**. An attempt to insert an incompatible element will cause a **ClassCastException**.
static <T> void copy(List<? super T> *list1*, List<? extends T> *list2*)	Copies the elements of *list2* to *list1*.
static boolean disjoint(Collection<?> *a*, Collection<?> *b*)	Compares the elements in *a* to elements in *b*. Returns **true** if the two collections contain no common elements (i.e., the collections contain disjoint sets of elements). Otherwise, returns **false**.
static <T> Enumeration<T> emptyEnumeration( )	Returns an empty enumeration, which is an enumeration with no elements.
static <T> Iterator<T> emptyIterator( )	Returns an empty iterator, which is an iterator with no elements.
static <T> List<T> emptyList( )	Returns an immutable, empty **List** object of the inferred type.
static <T> ListIterator<T> emptyListIterator( )	Returns an empty list iterator, which is a list iterator that has no elements.
static <K, V> Map<K, V> emptyMap( )	Returns an immutable, empty **Map** object of the inferred type.
static <K, V> NavigableMap<K, V> emptyNavigableMap( )	Returns an immutable, empty **NavigableMap** object of the inferred type.
static <E> NavigableSet<E> emptyNavigableSet( )	Returns an immutable, empty **NavigableSet** object of the inferred type.
static <T> Set<T> emptySet( )	Returns an immutable, empty **Set** object of the inferred type.
static <K, V> SortedMap<K, V> emptySortedMap( )	Returns an immutable, empty **SortedMap** object of the inferred type.
static <E> SortedSet<E> emptySortedSet( )	Returns an immutable, empty **SortedSet** object of the inferred type.
static <T> Enumeration<T> enumeration(Collection<T> *c*)	Returns an enumeration over *c*. (See "The Enumeration Interface," later in this chapter.)
static <T> void fill(List<? super T> *list*, T *obj*)	Assigns *obj* to each element of *list*.

static int frequency(Collection<?> *c*, object *obj*)	Counts the number of occurrences of *obj* in *c* and returns the result.
static int indexOfSubList(List<?> *list*, List<?> *subList*)	Searches *list* for the first occurrence of *subList*. Returns the index of the first match, or –1 if no match is found.
static int lastIndexOfSubList(List<?> *list*, List<?> *subList*)	Searches *list* for the last occurrence of *subList*. Returns the index of the last match, or –1 if no match is found.
static <T> ArrayList<T> list(Enumeration<T> *enum*)	Returns an **ArrayList** that contains the elements of *enum*.
static <T> T max(Collection<? extends T> *c*, Comparator<? super T> *comp*)	Returns the maximum element in *c* as determined by *comp*.
static <T extends Object & Comparable<? super T>> T max(Collection<? extends T> *c*)	Returns the maximum element in *c* as determined by natural ordering. The collection need not be sorted.
static <T> T min(Collection<? extends T> *c*, Comparator<? super T> *comp*)	Returns the minimum element in *c* as determined by *comp*. The collection need not be sorted.
static <T extends Object & Comparable<? super T>> T min(Collection<? extends T> *c*)	Returns the minimum element in *c* as determined by natural ordering.
static <T> List<T> nCopies(int *num*, T *obj*)	Returns *num* copies of *obj* contained in an immutable list. *num* must be greater than or equal to zero.
static <E> Set<E> newSetFromMap(Map<E, Boolean> *m*)	Creates and returns a set backed by the map specified by *m*, which must be empty at the time this method is called.
static <T> boolean replaceAll(List<T> *list*, T *old*, T *new*)	Replaces all occurrences of *old* with *new* in *list*. Returns **true** if at least one replacement occurred. Returns **false** otherwise.
static void reverse(List<T> *list*)	Reverses the sequence in *list*.
static <T> Comparator<T> reverseOrder(Comparator<T> *comp*)	Returns a reverse comparator based on the one passed in *comp*. That is, the returned comparator reverses the outcome of a comparison that uses *comp*.
static <T> Comparator<T> reverseOrder( )	Returns a reverse comparator, which is a comparator that reverses the outcome of a comparison between two elements.
static void rotate(List<T> *list*, int *n*)	Rotates *list* by *n* places to the right. To rotate left, use a negative value for *n*.
static void shuffle(List<T> *list*, Random *r*)	Shuffles (i.e., randomizes) the elements in *list* by using *r* as a source of random numbers.

static void shuffle(List<T> *list*)	Shuffles (i.e., randomizes) the elements in *list*.
static <T> Set<T> singleton(T *obj*)	Returns *obj* as an immutable set. This is an easy way to convert a single object into a set.
static <T> List<T> singletonList(T *obj*)	Returns *obj* as an immutable list. This is an easy way to convert a single object into a list.
static <K, V> Map<K, V> singletonMap(K *k*, V *v*)	Returns the key/value pair *k/v* as an immutable map. This is an easy way to convert a single key/value pair into a map.
static <T> void sort(List<T> *list*, Comparator<? super T> *comp*)	Sorts the elements of *list* as determined by *comp*.
static <T extends Comparable<? super T>> void sort(List<T> *list*)	Sorts the elements of *list* as determined by their natural ordering.
static void swap(List<?> *list*, int *idx1*, int *idx2*)	Exchanges the elements in *list* at the indices specified by *idx1* and *idx2*.
static <T> Collection<T> synchronizedCollection(Collection<T> *c*)	Returns a thread-safe collection backed by *c*.
static <T> List<T> synchronizedList(List<T> *list*)	Returns a thread-safe list backed by *list*.
static <K, V> Map<K, V> synchronizedMap(Map<K, V> *m*)	Returns a thread-safe map backed by *m*.
static <K, V> NavigableMap<K, V> synchronizedNavigableMap(NavigableMap<K, V> *nm*)	Returns a synchronized navigable map backed by *nm*.
static <T> NavigableSet<T> synchronizedNavigableSet(NavigableSet<T> *ns*)	Returns a synchronized navigable set backed by *ns*.
static <T> Set<T> synchronizedSet(Set<T> *s*)	Returns a thread-safe set backed by *s*.
static <K, V> SortedMap<K, V> synchronizedSortedMap(SortedMap<K, V> *sm*)	Returns a thread-safe sorted map backed by *sm*.
static <T> SortedSet<T> synchronizedSortedSet(SortedSet<T> *ss*)	Returns a thread-safe sorted set backed by *ss*.
static <T> Collection<T> unmodifiableCollection(Collection<? extends T> *c*)	Returns an unmodifiable collection backed by *c*.
static <T> List<T> unmodifiableList(List<? extends T> *list*)	Returns an unmodifiable list backed by *list*.
static <K, V> Map<K, V> unmodifiableMap(Map<? extends K, ? extends V> *m*)	Returns an unmodifiable map backed by *m*.

static <K, V> NavigableMap<K, V> unmodifiableNavigableMap(NavigableMap<K, ? extends V> *nm*)	Returns an unmodifiable navigable map backed by *nm*.
static <T> NavigableSet<T> unmodifiableNavigableSet(NavigableSet<T> *ns*)	Returns an unmodifiable navigable set backed by *ns*.
static <T> Set<T> unmodifiableSet(Set<? extends T> *s*)	Returns an unmodifiable set backed by *s*.
static <K, V> SortedMap<K, V> unmodifiableSortedMap(SortedMap<K, ? extends V> *sm*)	Returns an unmodifiable sorted map backed by *sm*.
static <T> SortedSet<T> unmodifiableSortedSet(SortedSet<T> *ss*)	Returns an unmodifiable sorted set backed by *ss*.

41 Arrays class

The Arrays class provides various methods that are useful when working with arrays. These methods help bridge the gap between collections and arrays.

42 method

each method is overloded for byte, char, double, float, int, long, short, object collectively reffered as 'Types' here in notes note that T impliese a generic while Types implies overlods

- —**binarySearch()**
- static int binarySearch(Type array[ ], Type value)
- static $\langle T \rangle$ int binarySearch(T[ ] array, T value, Comparator$\langle ? super T \rangle$ c)
- —**copyOf()**
- static Type[ ] copyOf(Type[ ] source, int len) : returns a copy of an array
- static $\langle T \rangle$ T[ ] copyOf(T[ ] source, int len)
- static $\langle T, U \rangle$ T[ ] copyOf(U[ ] source, int len, Class$\langle ? extends T[] \rangle$ resultT)
- The original array is specified by source, and the length of the copy is specified by len. If the copy is longer than source, then the copy is padded with zeros (for numeric arrays), nulls (for object arrays), or false (for boolean arrays). If the copy is shorter than source, then the copy is truncated. In the last form, the type of resultT becomes the type of the array returned
- —**copyOfRange()**
- The copyOfRange( ) method returns a copy of a range within an array
- static Type[ ] copyOfRange(Type[ ] source, int start,int end)
- static $\langle T \rangle$ T[ ] copyOfRange(T[ ] source,int start,int end)
- static $\langle T, U \rangle$ T[ ] copyOfRange(U[ ] source, int start,int end, Class$\langle ? extends T[] \rangle$ resultT)
- The original array is specified by source. The range to copy is specified by the indices passed via start and end. The range runs from start to end – 1. If the range is longer than source, then the copy is padded with zeros (for numeric arrays), nulls (for object arrays), or false (for boolean arrays). In the last form, the type of resultT becomes the type of the array returned
- —**equals()**
- The equals( ) method returns true if two arrays are equivalent. Otherwise, it returns false
- static boolean equals(Type array1[ ], Type array2 [ ])
- Several more versions are available that let you specify a range and/or a comparator
- —**deepEquals()**
- The deepEquals( ) method can be used to determine if two arrays, whichmight contain nested arrays, are equal.

- static boolean deepEquals(Object[ ] a, Object[ ] b)
- —**fill()**
- The fill( ) method assigns a value to all elements in an array. In other words, it fills an array with a specified value. The fill( ) method has two versions. The first version, which has the following forms, fills an entire array
- static void fill(Type array[ ], Type value)
- The second version of the fill() method assigns a value to a subset of an array
- —**sort()**
- The sort( ) method sorts an array so that it is arranged in ascending order. The sort( ) method has two versions. The first version, shown here, sorts the entire array
- static void sort(Type array[ ])
- static $\langle T \rangle$ void sort(T array[ ], Comparator$\langle ?superT \rangle$ c)
- The second version of sort( ) enables you to specify a range within an array that you want to sort.
- —**parallelSort()**
- parallelSort( ) sorts, into ascending order, portions of an array in parallel and then merges the results. This approach can greatly speed up sorting times. Like sort( ), there are two basic types of parallelSort( ), each with several overloads. The first type sorts the entire array. It is shown here
- static void parallelSort(Type array[ ])
- static $\langle T \rangle extendsComparable\langle ?superT \rangle$ void parallelSort(T array[ ])
- static $\langle T \rangle voidparallelSort(Tarray[], Comparator\langle ?superT \rangle$ c)
- The second version of parallelSort( ) enables you to specify a range within the array that you want to sort
- —**spliterator()**
- Arrays supports spliterators by including the spliterator( ) method. It has two basic forms. The first type returns a spliterator to an entire array
- static Spliterator.OfDouble spliterator(double array[ ])
- static Spliterator.OfInt spliterator(int array[ ])
- static Spliterator.OfLong spliterator(long array[ ])
- static $\langle T \rangle$ Spliterator spliterator(T array[ ])
- The second version of spliterator( ) enables you to specify a range to iterate within the array
- —**stream()**
- Arrays supports the Stream interface by including the stream( ) method. It has two forms. The first is shown here
- static DoubleStream stream(double array[ ])
- static IntStream stream(int array[ ])
- static LongStream stream(long array[ ])
- static $\langle T \rangle$ Stream stream(T array[ ])
- The second version of stream( ) enables you to specify a range within the array
- —**setAll( ) and parallelSetAll( )**
- setAll( ) and parallelSetAll( ). Both assign values to all of the elements, but parallelSetAll( ) works in parallel.
- static void setAll(Type array[],Type1toType2Function$\langle ?extendsT \rangle$ genVal)
- —**parallelPrefix()**
- parallelPrefix( ), and it modifies an array so that each element contains the cumulative result of an operation applied to all previous elements. For example, if the operation is multiplication, then on return, the array elements will contain the values associated with the running product of the original values
- static void parallelPrefix(Type array[ ], TypeBinaryOperator func)
- —**comparison methods**
- JDK 9 added three comparison methods to Arrays. They are compare( ), compareUnsigned( ), and mismatch( ). Each has several overloads and each has versions that let you define a range to compare. Here is a brief description of each
- The compare( ) method compares two arrays. It returns zero if they are the same, a positive value if the first array is greater than the second, and negative if the first array is less than the second
- To perform an unsigned comparison of two arrays that hold integer values, use compareUnsigned( ).
- To find the location of the first mismatch between two arrays, use mismatch( ). It returns the index of the mismatch, or -1 if the arrays are equivalent.
- Arrays also provides toString( ) and hashCode( ) for the various types of arrays. In addition, deepToString( ) and deepHashCode( ) are provided, which operate effectively on arrays that contain nested arrays.

43 End notes

- to convert a list to array: int[] arr = list.stream().mapToInt(i −⟩ i).toArray();