Javadoc snippets

Collections (documentation snippets from some interfaces and classes)

public interface Iterable<T>

default void	forEach(Consumer<? super T> action)	Performs the given action for each element of the Iterable until all elements have been processed or the action throws an exception.

public interface Collection<E> extends Iterable<E>

boolean	contains(Object o)	Returns true if this collection contains the specified element.
default boolean	removeIf(Predicate<? super E> filter)	Removes all of the elements of this collection that satisfy the given predicate.

public interface List<E> extends Collection<E>

default void	sort(Comparator<? super E> c)	Sorts this list according to the order induced by the specified Comparator.

public class HashSet<E> extends AbstractSet<E> implements Set<E>, Cloneable, Serializable

Constructor Summary

Constructors

Constructor	Description
HashSet()	Constructs a new, empty set; the backing HashMap instance has default initial capacity (16) and load factor (0.75).
HashSet(int initialCapacity)	Constructs a new, empty set; the backing HashMap instance has the specified initial capacity and default load factor (0.75).
HashSet(int initialCapacity, float loadFactor)	Constructs a new, empty set; the backing HashMap instance has the specified initial capacity and the specified load factor.
HashSet(Collection<? extends E> c)	Constructs a new set containing the elements in the specified collection.

public class TreeSet<E> extends AbstractSet<E> implements NavigableSet<E>, Cloneable, Serializable

Constructor Summary

Constructors

Constructor	Description
TreeSet()	Constructs a new, empty tree set, sorted according to the natural ordering of its elements.
TreeSet(Collection<? extends E> c)	Constructs a new tree set containing the elements in the specified collection, sorted according to the *natural ordering* of its elements.
TreeSet(Comparator<? super E> comparator)	Constructs a new, empty tree set, sorted according to the specified comparator.
TreeSet(SortedSet<E> s)	Constructs a new tree set containing the same elements and using the same ordering as the specified sorted set.

public class LinkedHashSet<E> extends HashSet<E> implements Set<E>, Cloneable, Serializable

Constructor Summary

Constructors

Constructor	Description
LinkedHashSet()	Constructs a new, empty linked hash set with the default initial capacity (16) and load factor (0.75).
LinkedHashSet(int initialCapacity)	Constructs a new, empty linked hash set with the specified initial capacity and the default load factor (0.75).
LinkedHashSet(int initialCapacity, float loadFactor)	Constructs a new, empty linked hash set with the specified initial capacity and load factor.
LinkedHashSet(Collection<? extends E> c)	Constructs a new linked hash set with the same elements as the specified collection.

```
public class HashMap<K,V> extends AbstractMap<K,V>
implements Map<K,V>, Cloneable, Serializable
```

Constructor Summary

Constructors

Constructor	Description
`HashMap()`	Constructs an empty HashMap with the default initial capacity (16) and the default load factor (0.75).
`HashMap(int initialCapacity)`	Constructs an empty HashMap with the specified initial capacity and the default load factor (0.75).
`HashMap(int initialCapacity, float loadFactor)`	Constructs an empty HashMap with the specified initial capacity and load factor.
`HashMap(Map<? extends K,? extends V> m)`	Constructs a new HashMap with the same mappings as the specified Map.

```
public class TreeMap<K,V> extends AbstractMap<K,V>
implements NavigableMap<K,V>, Cloneable, Serializable
```

Constructor Summary

Constructors

Constructor	Description
`TreeMap()`	Constructs a new, empty tree map, using the natural ordering of its keys.
`TreeMap(Comparator<? super K> comparator)`	Constructs a new, empty tree map, ordered according to the given comparator.
`TreeMap(Map<? extends K,? extends V> m)`	Constructs a new tree map containing the same mappings as the given map, ordered according to the *natural ordering* of its keys.
`TreeMap(SortedMap<K,? extends V> m)`	Constructs a new tree map containing the same mappings and using the same ordering as the specified sorted map.

```
public class LinkedHashMap<K,V> extends HashMap<K,V>
implements Map<K,V>
```

Constructor Summary

Constructors

Constructor	Description
`LinkedHashMap()`	Constructs an empty insertion-ordered LinkedHashMap instance with the default initial capacity (16) and load factor (0.75).
`LinkedHashMap(int initialCapacity)`	Constructs an empty insertion-ordered LinkedHashMap instance with the specified initial capacity and a default load factor (0.75).
`LinkedHashMap(int initialCapacity, float loadFactor)`	Constructs an empty insertion-ordered LinkedHashMap instance with the specified initial capacity and load factor.
`LinkedHashMap(int initialCapacity, float loadFactor, boolean accessOrder)`	Constructs an empty LinkedHashMap instance with the specified initial capacity, load factor and ordering mode.
`LinkedHashMap(Map<? extends K,? extends V> m)`	Constructs an insertion-ordered LinkedHashMap instance with the same mappings as the specified map.

```
public class ArrayList<E> extends AbstractList<E>
implements List<E>, RandomAccess, Cloneable, Serializable
```

Constructor Summary

Constructors

Constructor	Description
`ArrayList()`	Constructs an empty list with an initial capacity of ten.
`ArrayList(int initialCapacity)`	Constructs an empty list with the specified initial capacity.
`ArrayList(Collection<? extends E> c)`	Constructs a list containing the elements of the specified collection, in the order they are returned by the collection's iterator.

```
public class LinkedList<E> extends AbstractSequentialList<E>
implements List<E>, Deque<E>, Cloneable, Serializable
```

Constructor Summary

Constructors

Constructor	Description
`LinkedList()`	Constructs an empty list.
`LinkedList(Collection<? extends E> c)`	Constructs a list containing the elements of the specified collection, in the order they are returned by the collection's iterator.

Some (functional) interfaces (with the most important or the only one abstract method)

```
public interface Comparable<T>
```

int	`compareTo(T o)`	Compares this object with the specified object for order.

```
@FunctionalInterface
public interface Comparator<T>
```

int	`compare(T o1, T o2)`	Compares its two arguments for order.

```
@FunctionalInterface
public interface Consumer<T>
```

void	`accept(T t)`	Performs this operation on the given argument.

```
@FunctionalInterface
public interface Function<T,R>
```

R	`apply(T t)`	Applies this function to the given argument.

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

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

```
@FunctionalInterface
public interface Supplier<T>
```

T	`get()`	Gets a result.

Optional<T> and specialized Optional

```
public final class Optional<T> extends Object
```

T	`get()`	If a value is present, returns the value, otherwise throws `NoSuchElementException`.
int	`hashCode()`	Returns the hash code of the value, if present, otherwise `0` (zero) if no value is present.
void	`ifPresent(Consumer<? super T> action)`	If a value is present, performs the given action with the value, otherwise does nothing.
void	`ifPresentOrElse(Consumer<? super T> action, Runnable emptyAction)`	If a value is present, performs the given action with the value, otherwise performs the given empty-based action.
boolean	`isEmpty()`	If a value is not present, returns `true`, otherwise `false`.
boolean	`isPresent()`	If a value is present, returns `true`, otherwise `false`.

```
public final class OptionalDouble extends Object
```

double	**getAsDouble**()	If a value is present, returns the value, otherwise throws NoSuchElementException.
int	**hashCode**()	Returns the hash code of the value, if present, otherwise 0 (zero) if no value is present.
void	**ifPresent**(**DoubleConsumer** action)	If a value is present, performs the given action with the value, otherwise does nothing.
void	**ifPresentOrElse**(**DoubleConsumer** action, **Runnable** emptyAction)	If a value is present, performs the given action with the value, otherwise performs the given empty-based action.
boolean	**isEmpty**()	If a value is not present, returns true, otherwise false.
boolean	**isPresent**()	If a value is present, returns true, otherwise false.

Most important stream methods (with parameters and return types)

- Terminal operations
  - boolean allMatch(*Predicate*)
  - boolean anyMatch(*Predicate*)
  - long count()
  - Optional<T> findAny()
  - Optional<T> findFirst()
  - void forEach(*Consumer*)
  - Optional<T> max(*Comparator*)
  - Optional<T> min(*Comparator*)
  - boolean noneMatch(*Predicate*)
  - ...
- Intermediate operations
  - Stream<T> distinct()
  - Stream<T> filter(*Predicate*)
  - flatMap(*Function*)
  - ...Stream flatMapTo...(*Function*)[*]
  - Stream<T> limit(*long*)
  - <R> Stream<R>map(*Function*)
  - ...Stream mapTo...(*Function*)[*]
  - Stream<T> skip(*long*)
  - Stream<T> sorted()
  - ...

[*]... = Int, Long, Double

Most important additional specialized stream methods (example shows IntStream; same for DoubleStream or LongStream)

- Terminal operations
  - OptionalDouble average()
  - OptionalInt max()
  - OptionalInt min()
  - int sum()
  - IntSummaryStatistics summaryStatistics()
  - toArray()
  - ...
- Intermediate operations
  - DoubleStream asDoubleStream()
  - LongStream asLongStream()
  - Stream<Integer> boxed()
  - DoubleStream mapToDouble(*IntToDoubleFunction*)
  - LongStream mapToLong(*IntToLongFunction*)
  - <U> Stream<U> mapToObj(*IntFunction*)
  - IntStream range()
  - IntStream rangeClosed()
  - ...

Some Collectors functions

toMap

```
public static <T,K,U> Collector<T,?,Map<K,U>> toMap(Function<? super T,? extends K> keyMapper,
Function<? super T,? extends U> valueMapper)
```

Returns a `Collector` that accumulates elements into a `Map` whose keys and values are the result of applying the provided mapping functions to the input elements.

If the mapped keys contain duplicates (according to `Object.equals(Object)`), an `IllegalStateException` is thrown when the collection operation is performed. If the mapped keys might have duplicates, use `toMap(Function, Function, BinaryOperator)` instead.

toMap

```
public static <T,K,U> Collector<T,?,Map<K,U>> toMap(Function<? super T,? extends K> keyMapper,
Function<? super T,? extends U> valueMapper, BinaryOperator<U> mergeFunction)
```

Returns a `Collector` that accumulates elements into a `Map` whose keys and values are the result of applying the provided mapping functions to the input elements.

If the mapped keys contain duplicates (according to `Object.equals(Object)`), the value mapping function is applied to each equal element, and the results are merged using the provided merging function.

groupingBy

```
public static <T,K> Collector<T,?,Map<K,List<T>>> groupingBy(Function<? super T,? extends
K> classifier)
```

Returns a `Collector` implementing a "group by" operation on input elements of type `T`, grouping elements according to a classification function, and returning the results in a `Map`.

The classification function maps elements to some key type `K`. The collector produces a `Map<K, List<T>>` whose keys are the values resulting from applying the classification function to the input elements, and whose corresponding values are `Lists` containing the input elements which map to the associated key under the classification function.

groupingBy

```
public static <T,K,A,D> Collector<T,?,Map<K,D>> groupingBy(Function<? super T,? extends
K> classifier, Collector<? super T,A,D> downstream)
```

Returns a `Collector` implementing a cascaded "group by" operation on input elements of type `T`, grouping elements according to a classification function, and then performing a reduction operation on the values associated with a given key using the specified downstream `Collector`.

The classification function maps elements to some key type `K`. The downstream collector operates on elements of type `T` and produces a result of type `D`. The resulting collector produces a `Map<K, D>`.

partitioningBy

```
public static <T> Collector<T,?,Map<Boolean,List<T>>> partitioningBy(Predicate<? super
T> predicate)
```

Returns a `Collector` which partitions the input elements according to a `Predicate`, and organizes them into a `Map<Boolean, List<T>>`. The returned `Map` always contains mappings for both `false` and `true` keys. There are no guarantees on the type, mutability, serializability, or thread-safety of the `Map` or `List` returned.

partitioningBy

```
public static <T,D,A> Collector<T,?,Map<Boolean,D>> partitioningBy(Predicate<? super
T> predicate, Collector<? super T,A,D> downstream)
```

Returns a `Collector` which partitions the input elements according to a `Predicate`, reduces the values in each partition according to another `Collector`, and organizes them into a `Map<Boolean, D>` whose values are the result of the downstream reduction.

The returned `Map` always contains mappings for both `false` and `true` keys. There are no guarantees on the type, mutability, serializability, or thread-safety of the `Map` returned.

joining

```
public static Collector<CharSequence,?,String> joining(CharSequence delimiter)
```

Returns a `Collector` that concatenates the input elements, separated by the specified delimiter, in encounter order.

Parameters:

`delimiter` - the delimiter to be used between each element

Returns:

A `Collector` which concatenates CharSequence elements, separated by the specified delimiter, in encounter order

counting

```
public static <T> Collector<T,?,Long> counting()
```

Returns a `Collector` accepting elements of type `T` that counts the number of input elements. If no elements are present, the result is `0`.

mapping

```
public static <T,U,A,R> Collector<T,?,R> mapping(Function<? super T,? extends U> mapper,
Collector<? super U,A,R> downstream)
```

Adapts a `Collector` accepting elements of type `U` to one accepting elements of type `T` by applying a mapping function to each input element before accumulation.