Iterator Types

fail-fast

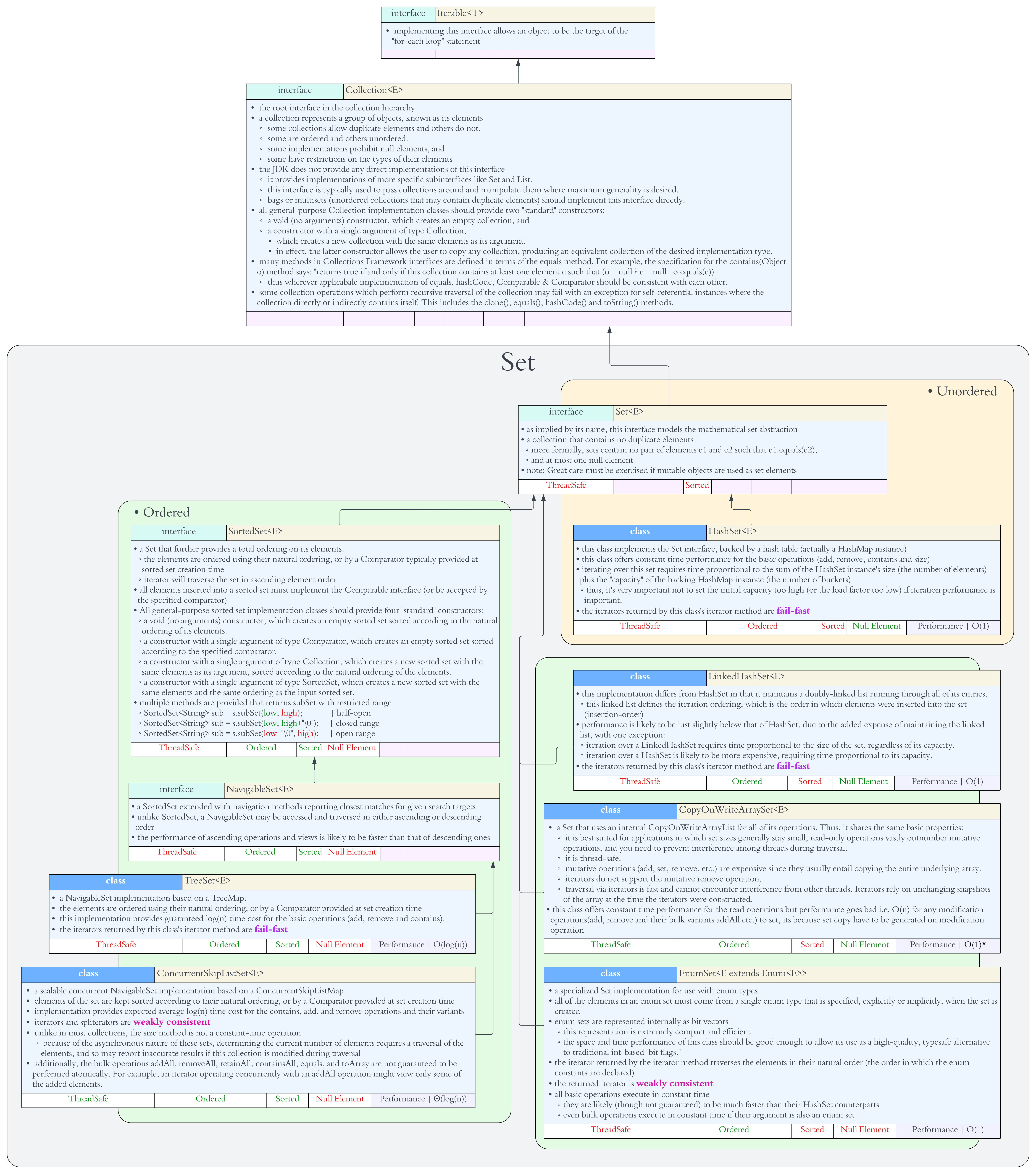
- collections maintain an internal counter called modCount.
  - each time an item is added or removed from the Collection, counter gets incremented.
  - when iterating, on each next() call, the current value of modCount gets compared with the initial value.
  - if there's a mismatch, it throws ConcurrentModificationException which aborts the entire operation.
  - however, this check is done without synchronization, so there is a risk of seeing a stale value of the modification count and therefore that the iterator does not realize a modification has been made.
  - this was a deliberate design tradeoff to reduce the performance impact of the concurrent modification detection code
- ConcurrentModificationException can arise in single-threaded code as well; this happens when objects are removed from the collection directly.
- if during iteration over a Collection, an item is removed using Iterator's remove() method, that's entirely safe and doesn't throw an exception.
- example: Default iterators on Collections from java.util package such as ArrayList, HashMap, etc.

weakly consistent

- reflects some but not necessarily all of the changes that have been made to their backing collection since they were created.
  - e.g. , if elements in the collection have been modified or removed before the iterator reaches them, it definitely will reflect these changes, but no such guarantee is made for insertions.
- the default iterator for the ConcurrentHashMap is weakly consistent.
  - this means that this Iterator can tolerate concurrent modification, traverses elements as they existed when Iterator was constructed and may (but isn't guaranteed to) reflect modifications to the Collection after the construction of the Iterator.
- example: Default iterators on Collections from java.util.concurrent package such as ConcurrentHashMap, ConcurrentSkipListSet etc.

fail-safe

- these iterators create a clone of the actual Collection and iterate over it. If any modification happens after the iterator is created, the copy still remains untouched. Hence, these Iterators continue looping over the Collection even if it's modified.
- disadvantage is the overhead of creating a copy of the Collection, both regarding time and memory.
- example: CopyOnWriteArrayList, CopyOnWriteArraySet



interface Iterable<T>

- implementing this interface allows an object to be the target of the "for-each loop" statement

interface Collection<E>

- the root interface in the collection hierarchy
- a collection represents a group of objects, known as its elements
  - some collections allow duplicate elements and others do not.
  - some are ordered and others unordered.
  - some implementations prohibit null elements, and
  - some have restrictions on the types of their elements
- the JDK does not provide any direct implementations of this interface
  - it provides implementations of more specific subinterfaces like Set and List.
  - this interface is typically used to pass collections around and manipulate them where maximum generality is desired.
  - bags or multisets (unordered collections that may contain duplicate elements) should implement this interface directly.
- all general-purpose Collection implementation classes should provide two "standard" constructors:
  - a void (no arguments) constructor, which creates an empty collection, and
  - a constructor with a single argument of type Collection,
    - which creates a new collection with the same elements as its argument.
    - in effect, the latter constructor allows the user to copy any collection, producing an equivalent collection of the desired implementation type.
- many methods in Collections Framework interfaces are defined in terms of the equals method. For example, the specification for the contains(Object o) method says: "returns true if and only if this collection contains at least one element e such that (o==null ? e==null : o.equals(e))
  - thus wherever applicabale impleimentation of equals, hashCode, Comparable & Comparator should be consistent with each other.
- some collection operations which perform recursive traversal of the collection may fail with an exception for self-referential instances where the collection directly or indirectly contains itself. This includes the clone(), equals(), hashCode() and toString() methods.

Set

• Unordered

interface Set<E>

- as implied by its name, this interface models the mathematical set abstraction
- a collection that contains no duplicate elements
  - more formally, sets contain no pair of elements e1 and e2 such that e1.equals(e2),
  - and at most one null element
- note: Great care must be exercised if mutable objects are used as set elements

ThreadSafe		Sorted			

class HashSet<E>

- this class implements the Set interface, backed by a hash table (actually a HashMap instance)
- this class offers constant time performance for the basic operations (add, remove, contains and size)
- iterating over this set requires time proportional to the sum of the HashSet instance's size (the number of elements) plus the "capacity" of the backing HashMap instance (the number of buckets).
  - thus, it's very important not to set the initial capacity too high (or the load factor too low) if iteration performance is important.
- the iterators returned by this class's iterator method are **fail-fast**

ThreadSafe	Ordered	Sorted	Null Element	Performance | O(1)

class LinkedHashSet<E>

- this implementation differs from HashSet in that it maintains a doubly-linked list running through all of its entries.
  - this linked list defines the iteration ordering, which is the order in which elements were inserted into the set (insertion-order)
- performance is likely to be just slightly below that of HashSet, due to the added expense of maintaining the linked list, with one exception:
  - iteration over a LinkedHashSet requires time proportional to the size of the set, regardless of its capacity.
  - iteration over a HashSet is likely to be more expensive, requiring time proportional to its capacity.
- the iterators returned by this class's iterator method are **fail-fast**

ThreadSafe	Ordered	Sorted	Null Element	Performance | O(1)

class CopyOnWriteArraySet<E>

- a Set that uses an internal CopyOnWriteArrayList for all of its operations. Thus, it shares the same basic properties:
  - it is best suited for applications in which set sizes generally stay small, read-only operations vastly outnumber mutative operations, and you need to prevent interference among threads during traversal.
  - it is thread-safe.
  - mutative operations (add, set, remove, etc.) are expensive since they usually entail copying the entire underlying array.
  - iterators do not support the mutative remove operation.
  - traversal via iterators is fast and cannot encounter interference from other threads. Iterators rely on unchanging snapshots of the array at the time the iterators were constructed.
- this class offers constant time performance for the read operations but performance goes bad i.e. O(n) for any modification operations(add, remove and their bulk variants addAll etc.) to set, its because set copy have to be generated on modification operation

ThreadSafe	Ordered	Sorted	Null Element	Performance | O(1)*

class EnumSet<E extends Enum<E>>

- a specialized Set implementation for use with enum types
- all of the elements in an enum set must come from a single enum type that is specified, explicitly or implicitly, when the set is created
- enum sets are represented internally as bit vectors
  - this representation is extremely compact and efficient
  - the space and time performance of this class should be good enough to allow its use as a high-quality, typesafe alternative to traditional int-based "bit flags."
- the iterator returned by the iterator method traverses the elements in their natural order (the order in which the enum constants are declared)
- the returned iterator is **weakly consistent**
- all basic operations execute in constant time
  - they are likely (though not guaranteed) to be much faster than their HashSet counterparts
  - even bulk operations execute in constant time if their argument is also an enum set

ThreadSafe	Ordered	Sorted	Null Element	Performance | O(1)

• Ordered

interface SortedSet<E>

- a Set that further provides a total ordering on its elements.
  - the elements are ordered using their natural ordering, or by a Comparator typically provided at sorted set creation time
  - iterator will traverse the set in ascending element order
- all elements inserted into a sorted set must implement the Comparable interface (or be accepted by the specified comparator)
- All general-purpose sorted set implementation classes should provide four "standard" constructors:
  - a void (no arguments) constructor, which creates an empty sorted set sorted according to the natural ordering of its elements.
  - a constructor with a single argument of type Comparator, which creates an empty sorted set sorted according to the specified comparator.
  - a constructor with a single argument of type Collection, which creates a new sorted set with the same elements as its argument, sorted according to the natural ordering of the elements.
  - a constructor with a single argument of type SortedSet, which creates a new sorted set with the same elements and the same ordering as the input sorted set.
- multiple methods are provided that returns subSet with restricted range
  - SortedSet<String> sub = s.subSet(low, high); | half-open
  - SortedSet<String> sub = s.subSet(low, high+"\0"); | closed range
  - SortedSet<String> sub = s.subSet(low+"\0", high); | open range

ThreadSafe	Ordered	Sorted	Null Element		

interface NavigableSet<E>

- a SortedSet extended with navigation methods reporting closest matches for given search targets
- unlike SortedSet, a NavigableSet may be accessed and traversed in either ascending or descending order
- the performance of ascending operations and views is likely to be faster than that of descending ones

ThreadSafe	Ordered	Sorted	Null Element		

class TreeSet<E>

- a NavigableSet implementation based on a TreeMap.
- the elements are ordered using their natural ordering, or by a Comparator provided at set creation time
- this implementation provides guaranteed log(n) time cost for the basic operations (add, remove and contains).
- the iterators returned by this class's iterator method are **fail-fast**

ThreadSafe	Ordered	Sorted	Null Element	Performance | O(log(n))

class ConcurrentSkipListSet<E>

- a scalable concurrent NavigableSet implementation based on a ConcurrentSkipListMap
- elements of the set are kept sorted according to their natural ordering, or by a Comparator provided at set creation time
- implementation provides expected average log(n) time cost for the contains, add, and remove operations and their variants
- iterators and spliterators are **weakly consistent**
- unlike in most collections, the size method is not a constant-time operation
  - because of the asynchronous nature of these sets, determining the current number of elements requires a traversal of the elements, and so may report inaccurate results if this collection is modified during traversal
- additionally, the bulk operations addAll, removeAll, retainAll, containsAll, equals, and toArray are not guaranteed to be performed atomically. For example, an iterator operating concurrently with an addAll operation might view only some of the added elements.

ThreadSafe	Ordered	Sorted	Null Element	Performance | Θ(log(n))

Collections with Uniqueness: Sets

Sets are collections of distinct elements. There are no duplicates

Outline

Set Features

The Why and How of Sets

Hashcode and Equals

Understanding the contract behind HashSet

Set Implementations

Performance tradeoffs and features

Hashcode and Equals

```
object.equals(other)

➔

object.hashCode() == other.hashCode()
```

Hashcode / Equals Contract

Equality

It can be reference based or value based. Reference based just needs to inherit equals from Object. Value based requires a custom equals method.

```
result = 31 * result + obj.hashCode();


// Arrays

Arrays.hashCode()


// Primitives (Java 8+)

Long.hashCode(longValue)


// Old Primitives

(int) (l ^ (l >>> 32))
Float.floatToIntBits(f);
```

- Combine hashcode information from each field
- IDE Can auto-generate
- Objects.hash() (Java 7+)
- ALWAYS use the same fields as equals()

Set Implementations

HashSet

Based upon HashMap

Uses hashcode() and looks up location

Good General Purpose Implementation

Use by default

TreeSet

Based Upon TreeMap

Red/Black binary tree with defined sort order

Provides Extra Features

Implements SortedSet and NavigableSet

Summary

Sets are a commonly used collection

Different implementations for different purposes

Remember to get the hashcode/equals contract correct