Infinitetrance/
Collections-Training-...

Infinitetrance/Collections-Training-Material
GitHub

Map

• Unordered

interface Map<K,V>

- an object that maps keys to values
- a map cannot contain duplicate keys
- map interface provides three collection views
  - set of keys — Set<K> — keySet()
  - collection of values — Collection<V> — values()
  - set of key-value mappings — Set<Map.Entry<K,V>> — entrySet()
- keys are recommended to be immutable objects

		Sorted			

interface ConcurrentMap<K,V>

- because of the asynchronous nature of these maps, the size method is not accurate & of not constant-time operation.

ThreadSafe					

class ConcurrentHashMap<K,V>

- a hash table supporting full concurrency of retrievals and high *expected* concurrency for updates
- as too many hash collisions can cause internal table to dynamically expand
  - and because table resizing is relatively slow operation
  - initialCapacity should be wisely selected
- constructor parameter *concurrencyLevel* provides concurrency tuning
  - ConcurrentHashMap(int initialCapacity, float loadFactor, int concurrencyLevel)
  - *concurrencyLevel is* the estimated number of concurrently updating threads

ThreadSafe	Ordered	Sorted	Null Key	Null Value	Performance ~ O(1)*

class HashMap<K,V>

- hash table based implementation of the Map interface
- constant-time performance for the basic operations (get and put)
- capacity is the number of buckets in the hash table
- initial capacity is the capacity at the time the hash table is created
- load factor is a measure of how full the hash table is allowed to get before its capacity is automatically increased
- size is the number of entries stored in
- iteration over HashMap requires time proportional to its capacity not its size.

ThreadSafe	Ordered	Sorted	Null Key	Null Value	Performance ~ O(1)

class Hashtable<K,V>

- its legacy implementation from Java 2 retrofitted to the collection framework
- HashMap or ConcurrentHashMap should be preferred over it

ThreadSafe	Ordered	Sorted	Null Key	Null Value	Performance ~ O(1)

class IdentityHashMap<K,V>

- internal hash table is implemented using reference-equality in place of object-equality when comparing keys (and values)
- a typical use of this class is
  - topology-preserving object graph transformations, such as serialization or deep-copying. To perform such a transformation, a program must maintain a "node table" that keeps track of all the object references that have already been processed. The node table must not equate distinct objects even if they happen to be equal.
  - another typical use of this class is to maintain proxy objects. For example, a debugging facility might wish to maintain a proxy object for each object in the program being debugged.

ThreadSafe	Ordered	Sorted	Null Key	Null Value	Performance ~ O(1)

• Ordered

interface SortedMap<K,V>

- entries are ordered according to the natural ordering of its keys, or as per the provided Comparator
- all keys must implement Comparable or be acceptable by the provided Comparator
- Each implementation generally provide 4 constructors
  - A void (no arguments) constructor, makes map sorted according to the natural ordering of its keys.
  - A constructor with a single argument of type Comparator, makes map sorted according to the specified comparator
  - A constructor with a single argument of type Map, which creates a new map with the same key-value mappings as its argument, sorted according to the keys' natural ordering.
  - A constructor with a single argument of type SortedMap, which creates a new sorted map with the same key-value mappings and the same ordering as the input sorted map.
- multiple methods are provided that return submaps with restricted key ranges.
  - a half-open range includes either their low endpoint or high endpoint
    - color code | inclusive | exclusive
    - SortedMap<String, V> sub = m.subMap(low, high);
  - a closed range includes both endpoints
    - SortedMap<String, V> sub = m.subMap(low, high+"\0");
  - open range includes neither endpoints
    - SortedMap<String, V> sub = m.subMap(low+"\0", high);

	Ordered	Sorted			

interface ConcurrentNavigableMap<K,V>

ThreadSafe	Ordered	Sorted			

interface NavigableMap<K,V>

- provides multiple variants of methods additional to SortedMap for locating/searching elements, not traversing
- a NavigableMap may be accessed and traversed in either ascending or descending key order
- performance of ascending operations and views is likely to be faster than that of descending ones

	Ordered	Sorted			

class ConcurrentSkipListMap<K,V>

- this class implements a concurrent variant of SkipLists
- implementation provides average log(n) time cost for the containsKey, get, put and remove operations

ThreadSafe	Ordered	Sorted	Null Key	Null Value	Performance ~ Θ(log(n))

class TreeMap<K,V>

- its a Red-Black tree based NavigableMap implementation
- implementation provides guaranteed log(n) time cost for the containsKey, get, put and remove operations

ThreadSafe	Ordered	Sorted	Null Key	Null Value	Performance ~ O(log(n))

class LinkedHashMap<K,V>

- hash table and linked list implementation of the Map interface
- it maintains a doubly-linked list running through all of its entries
- constant-time performance for the all basic operations (get and put), slightly more expensive than HashMap due to overhead of LinkedList
- iteration requires time proportional to its size regardless of its capacity & thus faster than HashMap
- A special constructor is provided to create a linked hash map whose order of iteration is the order in which its entries were last accessed
- This kind of map is well-suited to building LRU caches

ThreadSafe	Ordered	Sorted	Null Key	Null Value	Performance ~ O(1)

class EnumMap<K extends Enum<K>,V>

- a specialized Map implementation for use with only enum type keys
- all keys should belong to single enum type
- it uses array based hash table implementation
- constant-time performance for the all basic operations, generally faster than HashMap
- it maintains the natural order of keys (the order in which the enum constants are declared)

ThreadSafe	Ordered	Sorted	Null Key	Null Value	Performance ~ O(1)

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