Collections

The Collections Framework

The Collections Framework

- A collection is a container to group objects together
- Collections allow you to store, retrieve, and manipulate objects
- The Collections Framework is a library to support collections
  - Interfaces
  - Implementations
  - Algorithms
- https://docs.oracle.com/javase/9/docs/api/java/util/package-summary.html#CollectionsFramework

Interface vs. Implementation

- Interface classes describe how a collection works
    - How you can access elements, how elements can be modified, etc.
        - Allow duplicates?
        - Ordered or unordered?
        - Sorted or unsorted?
        - Allow direct access?
    - Example:
        - Queue- only add to back and remove from front
        - Stack- only add and remove from top
        - Set- unordered, no duplicates
- Concrete classes implement that design
    - Using an array, linked nodes, hash table, etc.
    - Example:
        - Queue could be implemented with linked nodes
        - Stack could be implemented with an array
        - ArrayList and LinkedList are both lists with different implementation designs

The Collection Interface

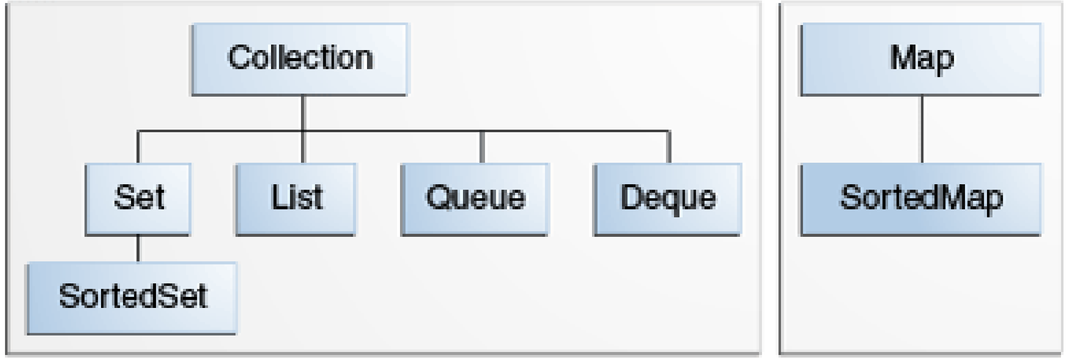
- The fundamental interface for the collection classes
  - https://docs.oracle.com/javase/9/docs/api/java/util/Collection.html
- Key methods:
  - `boolean add (E element)`
  - `Iterator<E> iterator()`
  - `boolean remove (Object o)`
  - `int size()`

The Collection Interface

- Other methods:
  - `addAll(Collection<? extends E> c)`
  - `clear()`
  - `contains(Object o)`
  - `containsAll(Collection<?> c)`
  - `removeAll(Collection<?> c)`
  - `Object[] toArray()`
  - `<T> T[] toArray(T[] a)`
- Many Collection methods are implemented in AbstractCollection so that the implementation can be inherited in all collections.

The Collection Interface

- There are three interfaces that extend Collection:
  - Set
  - List
  - Queue
- Concrete classes then implement these interfaces.



The Iterator Interface

- Allows you to access elements in a collection
- Methods:
  - `E next()`
  - `boolean hasNext()`
  - `void remove()`
- The order of iteration is dependent on the type of collection
  - Example: ArrayList iterates in order starting at 0 but HashSet returns the elements in random order

The Iterator Interface

- General syntax:

```
Iterator<String> iterator = collection.iterator();
while(iterator.hasNext()) {
   String element = iterator.next();
   // do something with element
}
```

For-Each

- You can also use the for-each loop for any collections object.
    - This is because the Collection interface extends the Iterable interface
- The compiler translates for-each loops into iterator-loops.

```
for(String element : collection) {
   // do something with element
}
```

The Iterator Interface- remove Method

- The remove method removes the element returned by the last call to next.
- The iterator's remove method is the best way to remove an element from a collection.
- Avoid removing inside of a loop- this is error prone.
    - Example:

```
for (int i = 0; i < list.size(); i++) {
        list.remove(i);
}
```

The Iterator Interface- remove Method

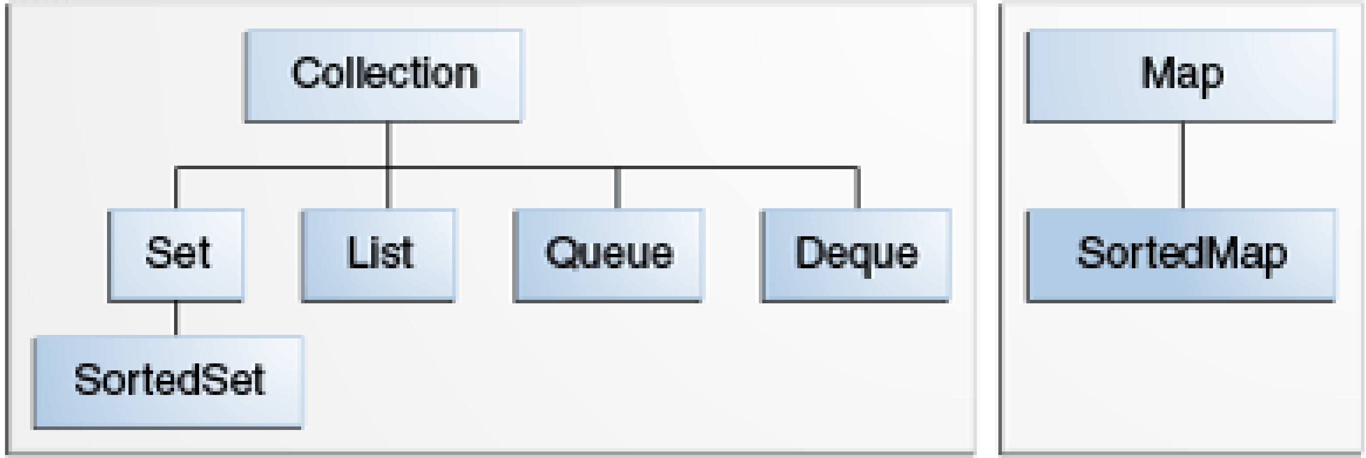
- You **must** advance to an element with next() before removing it.

```
iterator.next();
iterator.remove();
iterator.remove(); // not allowed!
```

Lists

Lists

- The List interface describes a collection that:
  - is ordered (and indexed)
  - typically allows duplicates
- List indices start at position 0.
- Key methods:
  - E get(int index)



ListIterator

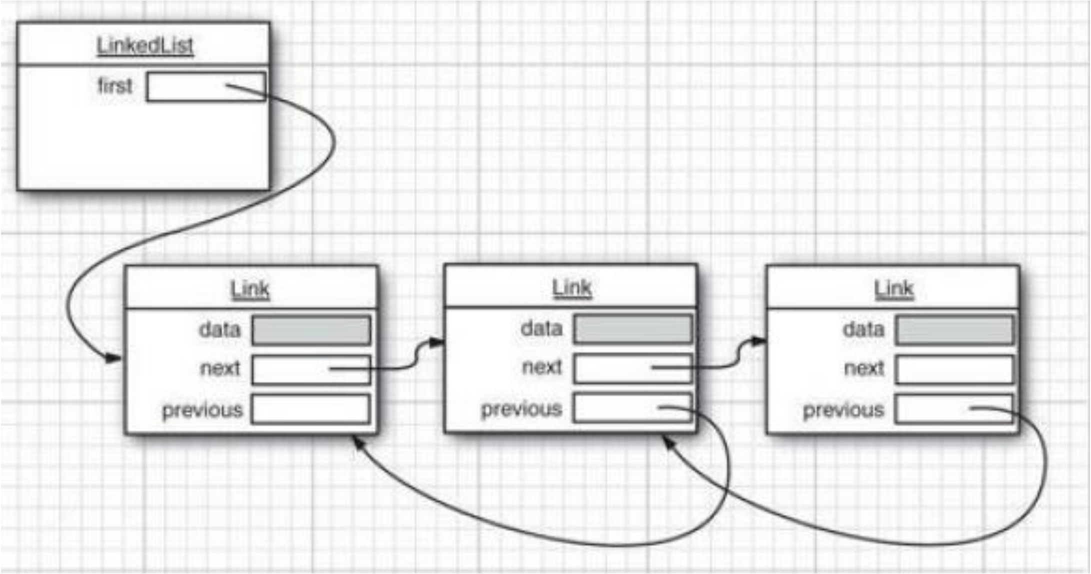
- You can use an Iterator on ArrayList or LinkedList.
- You can also use a ListIterator (which extends Iterator), which includes these additional methods:
  - add(E element) // adds new element *before* the iterator position
  - E previous()
  - boolean hasPrevious()
- These methods allow you to traverse a list backwards or forwards or add at a certain position while iterating.
- Obtain the ListIterator with .listIterator() method.

ArrayList

- Ordered collection of elements
- Implemented using an array
- Direct access to any element (good!)
- Inserting or removing from the middle requires shifting (bad!)

LinkedList

- Ordered collection of elements
- Implemented using doubly linked nodes
- Inserting or removing from the middle with an iterator is inexpensive (good!)
- No direct access to any element (bad!)
  - Although you *can* access any direct element with the get method, you shouldn't! If you need to do this very often, you probably want to use an ArrayList instead.



Practice

- Look over the business data.
  - Downloaded and simplified from: https://data.sfgov.org/Economy-and-Community/Registered-Business-Locations-San-Francisco/g8m3-pdis
- Run the ListTester program to read in the data, find, and remove some businesses.
  - Use an ArrayList.
  - Use a LinkedList.

Lists- When to Use?

- Good for:
    - Keeping track of elements *in order*
    - Applying an action to all elements in the collection
    - Applying an action only to elements that meet some criteria
        - e.g., all CA businesses
- Not as good for:
    - Finding a specific object based on a characteristic
        - e.g., find a business with a specific ID
        - This will require us to traverse the list to find the object

ArrayList vs LinkedList

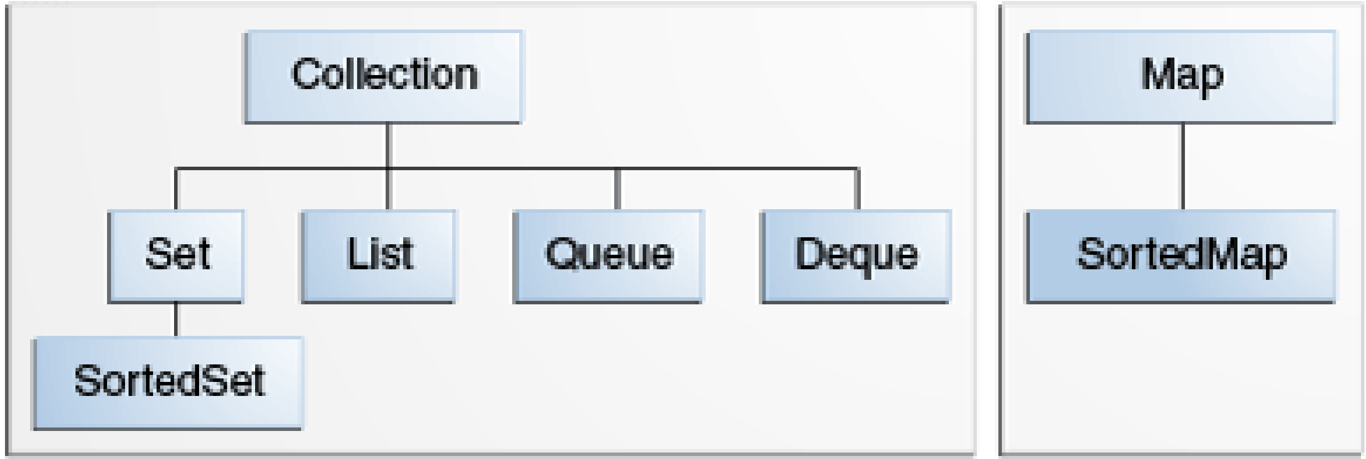
- Most often, you’ll want to use ArrayList.
  - ArrayList is “just plain fast”
- *Consider* LinkedList if:
  - frequently adding to the beginning of a list
  - frequently deleting elements from the interior of the list

Action	ArrayList	LinkedList
Direct access- get(i)	O(1)	O(n)
Adding/removing beginning	O(n)	O(1)
Adding/removing middle	O(n)	O(n) O(1) with iterator
Adding/removing end	O(1)	O(1)

Sets

Sets

- The Set interface describes a collection that:
  - does not allow duplicates
  - may or may not be ordered



HashSet

- Implements Set with a hash table
  - Uses an object’s hashCode method to determine if the object is already in the set (and thus will not add a duplicate)
- Does not guarantee any order (even of the iterator)

TreeSet

- Implements a sorted set
- The iterator returns the values in sorted order (using the *natural* ordering- compareTo)
  - You can also send in a Comparator object to the TreeSet constructor if you want to use a different method of comparison.
- Be careful- duplicates are determined by the compareTo or compare method!

Practice

- Review the set examples.

Sets- When to Use?

- Good for:
    - Not allowing duplicates
    - Applying an action to all elements in the collection
    - Applying an action only to elements that meet some criteria
    - Determining if an element is in the set
- Not as good for:
    - Finding a specific object based on a characteristic
        - e.g., find a business with a specific ID
        - This will require us to traverse the set to find the object
- Which set to use?
    - HashSet is quicker than TreeSet, but does not provide ordering.

Caution!

- You cannot add a duplicate element to a set.
- HOWEVER! You could *modify* an element in the set to make it a duplicate of an element already in the set.
- Example:
  - Set<Student> contains s1 (“Jane Doe”) and s2(“Mike Smith”)
  - If you invoke s1.setName(“Mike Smith”), your set now has duplicates!

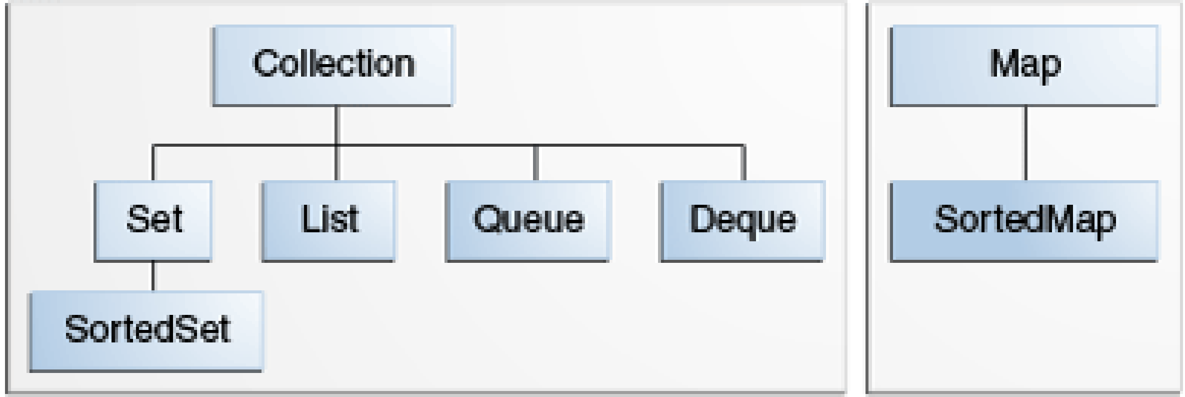
Queues and Deques

Queue

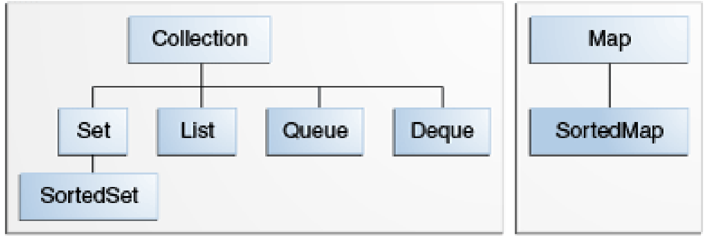
- The Queue interface describes a collection that:
  - is ordered
  - typically allows duplicates
  - only allows adds at the back (tail) and removes from the front (head)
    - no middle access is supported!
- Implementing Classes: ArrayDeque and LinkedList
- Key methods:
  - boolean add(E) or offer(E)
  - E remove() or E poll()
  - E element() or E peek()

throw exception

return a special value



Deque



- The Deque (double-ended queue) interface describes a collection that:
  - is ordered
  - typically allows duplicates
  - allows adds and removes at the front and back
    - no middle access is supported!
- Implementing Classes: ArrayDeque and LinkedList
- Key methods:
  - addFirst(E item), addLast(E item),   offerFirst(E item), offerLast(E item)
  - E removeFirst(), E removeLast(),   E pollFirst(), E pollLast()
  - E getFirst(), getLast(),   E peekFirst(), E peekLast()

throw
exception

return a
special value

Using a Queue

```
Queue<Business> businessQueue = new LinkedList<Business>();
```

- Even though instantiated as a LinkedList, it is declared as a Queue, which restricts the methods to the methods in the Queue class.
- Why is a LinkedList a good implementation of a Queue?

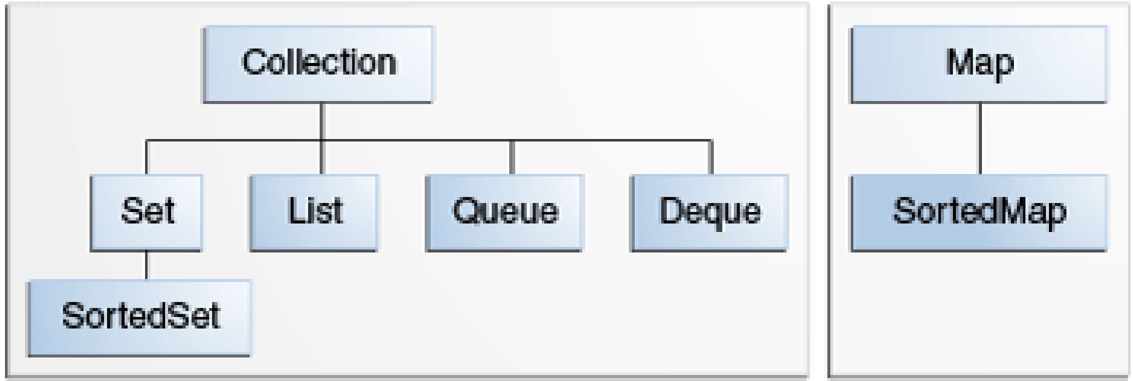
Stacks

- A stack allows you only to add and remove from the top
  - No middle (or bottom) access is supported
- Stack implements Deque interface and adds these methods:
  - E push(E)
  - E pop()
  - E peek()

Maps

Maps (also known as *Hash Tables*)

- Collections of data stored with a *key* and *value*
- Each key is unique
- Each key maps to exactly one value
- Whether nulls are allowed or what happens when a key does not exist depends on the implementation
- Methods:
  - V put(K key, V value)
  - V get(K key)
  - boolean containsKey(Object key)
  - V replace(K key, V value)



Views of the Map

- Set<K> keySet()
- Collection<V> values()
- These are **backed by the map**.
  - Changes to the map affect the keyset and values
  - You cannot add things to these views
    - You’ll get an UnsupportedOperationException
  - Removing from these views removes from the map
  - Removing or adding to or from the map changes the views

HashMap

- An unordered map
- Implement behind the scenes using hash tables
- Only one value per key is supported

TreeMap

- An ordered map (ordering when iterating over the keyset or values)
- The **keys** maintain sorted order
  - Natural ordering of the keys
  - You can also specify a Comparator
- Only one value per key is supported

Practice

- Review the maps example that maps by ID
- What if we want to have a map where the key is the owner’s name? There are multiple values for each key!
    - What can we use for the value?

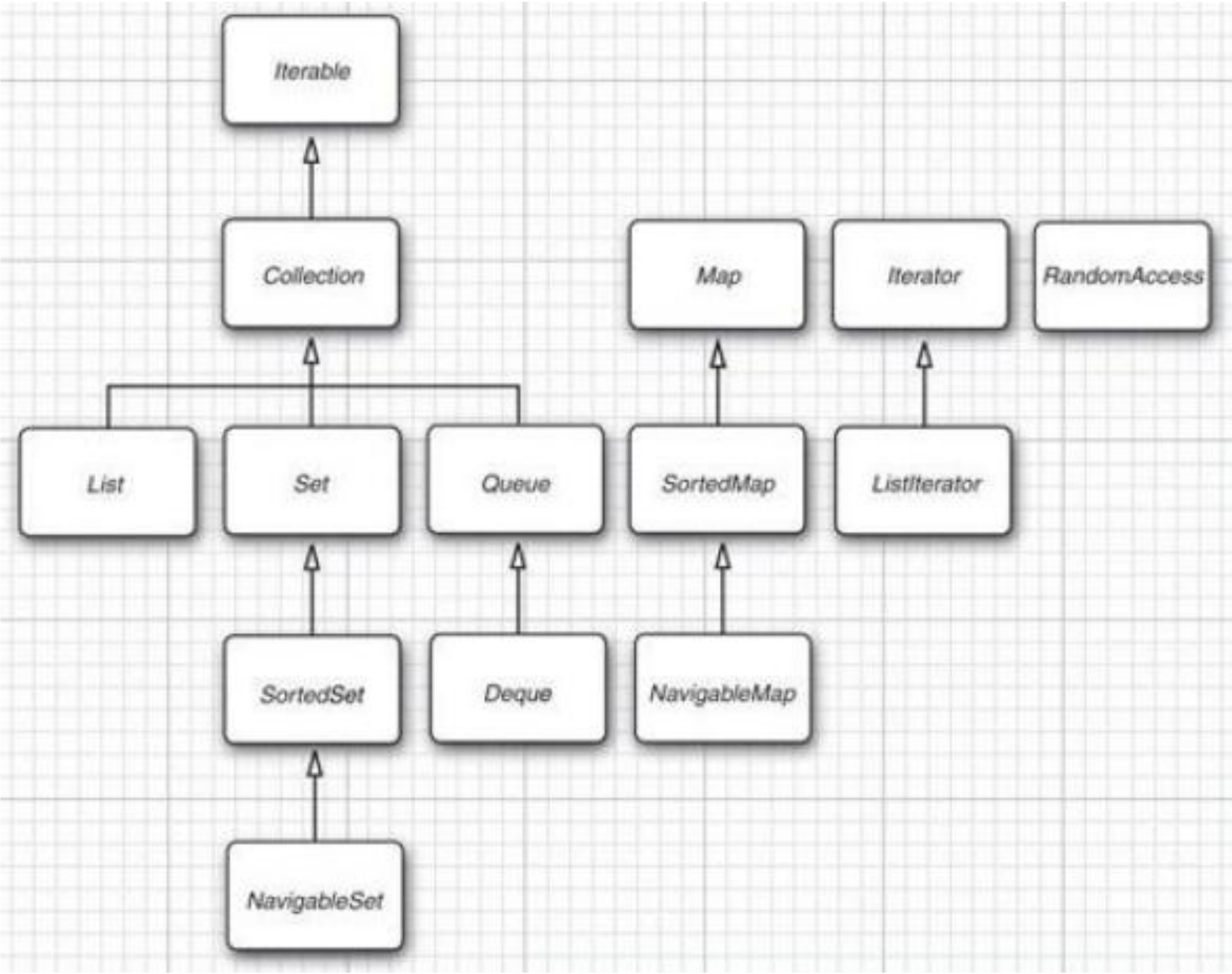
Maps- When to Use?

- Good for:
  - Finding an object based on a characteristic
  - Keeping track of multiple collections
- Which map to use?
  - HashMap is unordered, TreeMap is ordered

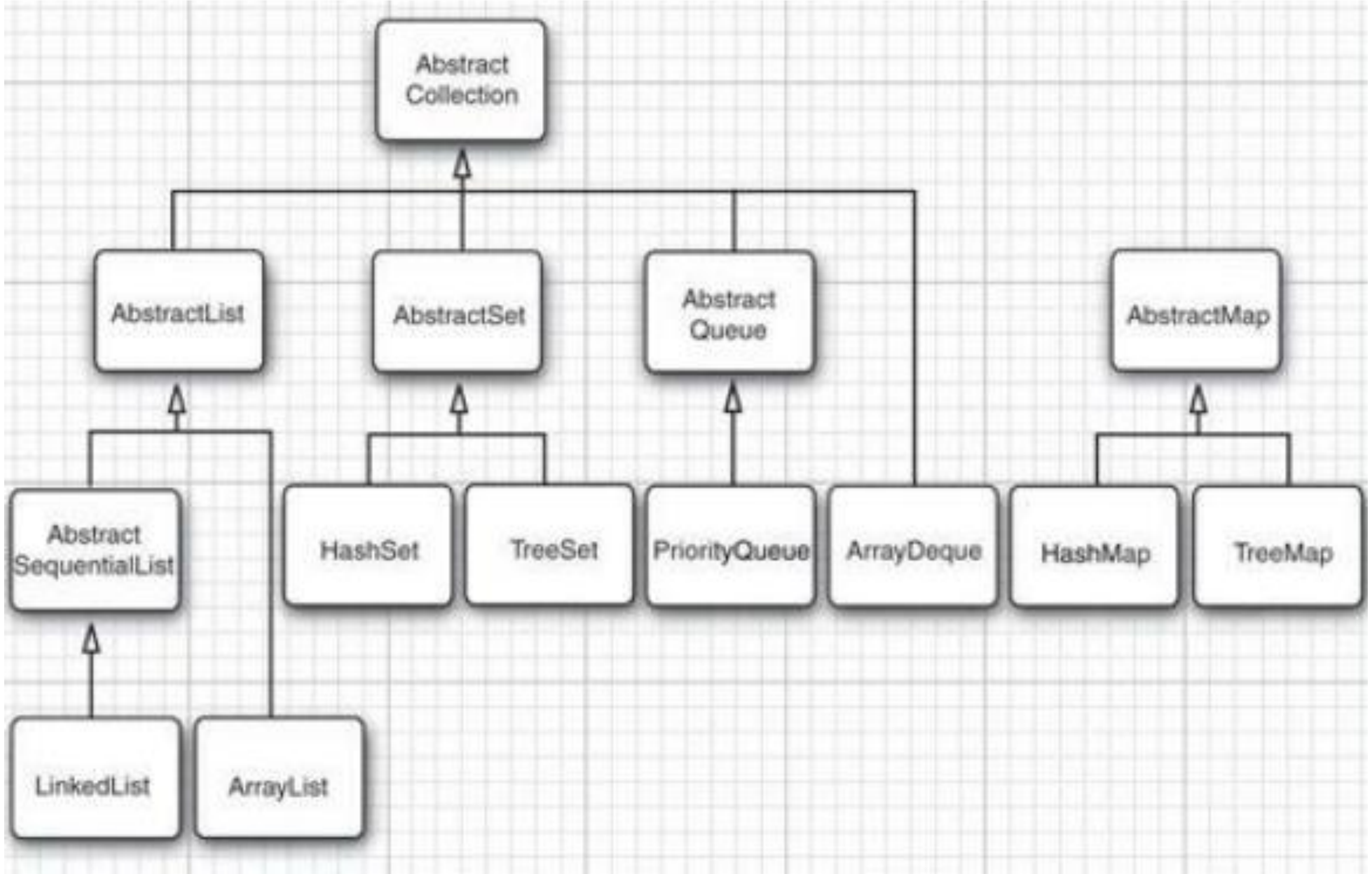
Hashtable

- Essentially the same as HashMap, only synchronized
- If you don’t need synchronization, use HashMap
- Enumeration
  - Similar to iterator
  - hasMoreElements(), nextElement()
  - myHashtable.keys() and myHashtable.elements() return Enumerations

Interfaces


Iterable
Collection
Map
Iterator
RandomAccess
List
Set
Queue
SortedMap
ListIterator
SortedSet
Deque
NavigableMap
NavigableSet

Classes



Abstract Collection
AbstractList
AbstractSet
Abstract Queue
AbstractMap
Abstract SequentialList
HashSet
TreeSet
PriorityQueue
ArrayDeque
HashMap
TreeMap
LinkedList
ArrayList

VIEWS AND WRAPPERS

Views

- A *view* is a object that implements one of the Collection or Map interfaces but is **linked/connected** to another collection object.
  - It could be immutable.
  - If it can be changed, changes to the view can affect the object
- Example: Set<String> keys = hashMap.keySet();
  - Keys is a Set, but you cannot add to it.
  - Removing from the set affects the map.
  - It’s not that we have created a whole new set of Strings that we can go off and use.
  - We’ve instead created a *view* of the keys from the hash map.

Wrappers

- Wrapper methods return a collection that has additional functionality.
- Three main purposes:
  - to create a synchronized collection
  - to create an unmodifiable collection
  - to create a type-checked collection
- An example of the *decorator* pattern
- Methods are in the Collections class

Unmodifiable Collections Methods

- You cannot add or remove to or from these collections.
- Static methods in the Collections class:
  - List<T> unmodifiableList(List<? extends T> list)
  - Map<K,V> unmodifiableMap(Map<? extends K,? extends V> m)
  - Set<T> unmodifiableSet(Set<? extends T> s)

Generating **Immutable** Views from Collections

- Collections.emptyList()   emptySet()   emptyMap()
- Collections.singletonList(item)   singleton(item)   singletonMap(key, val)
- Collections.nCopies(int, item) // returns a list

The Arrays.asList Wrapper

- Creates a *view* of an array as a List.
- You can use get or set methods on the list.
  - These will also change the array!!
- You cannot use add or remove.
  - Gives you a runtime UnsupportedOperationException
  - Not a compile time error (Unfortunately!)

Subrange Views

- You can generate a subrange **view** for some collections
  - Changes to the view affect the original collection!
- subList(int, int)
  - first is inclusive, second exclusive (just like substring)
- subMap(K from, K to)
  - views of the map with all entries where the keys are in the specific ranges

Bulk Operations

- retainAll(collection)
- removeAll(collection)
- addAll(collection)

Obtaining Arrays

- toArray() returns an Object[]
  - You cannot cast this!
- Instead, send in an array of the type you want.
  - Size 0 means the method will create a new appropriate sized array.
  - Bigger size means the method will use that array.
  - Example: words.toArray(new String[0]);
  - Example: words.toArray(new String(words.size()));

ALGORITHMS

Collections Methods

- Collections.max(Collection)
  - Collections.max(Collection, Comparator)
- Collections.min(Collection)
  - Collections.min(Collection, Comparator)

- Collections.shuffle(Collection)

Collections Methods

- Collections.sort(Collection)
  - Collections.sort(Collection, Comparator)
  - Collections.sort(Collection, Collections.reverseOrder())
  - Collections.sort(Collection, Collections.reverseOrder(Comparator))

- Collections.binarySearch(Collection, element)
  - Collections.binarySearch(Collection, element, Comparator)
  - A negative value represents where the item would have been: at position -i-1
  - Reverts to linear search if given a linked list

Collections Methods

- Collections.copy(toList, fromList)
- Collections.fill(toList, element)
- Collections.swap(list, positionA, position)
- Collections.reverse(list)
- Collections.rotate(list, rotateFactor)
- Collections.frequency(collection, element)