Lập trình hướng đối tượng

Biên soạn: Nguyễn Thị Tuyết Hải

Email: tuyethai@ptithcm.edu.vn

Chapter 9: Collections

9.1 The Java Collections Framework

9.1.1 Separating Collection Interfaces and Implementation

E.g.,

- Interface tells nothing about how the queue is implemented.

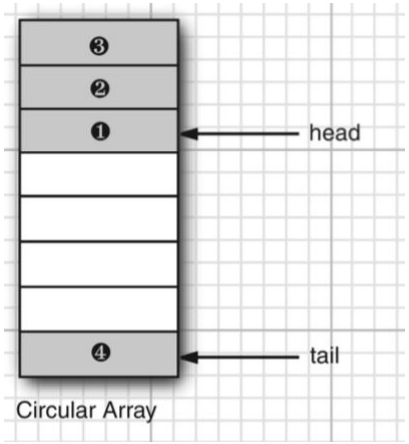
```
public interface Queue<E>{
        void add(E element);
        E remove();
        int size();
}
```

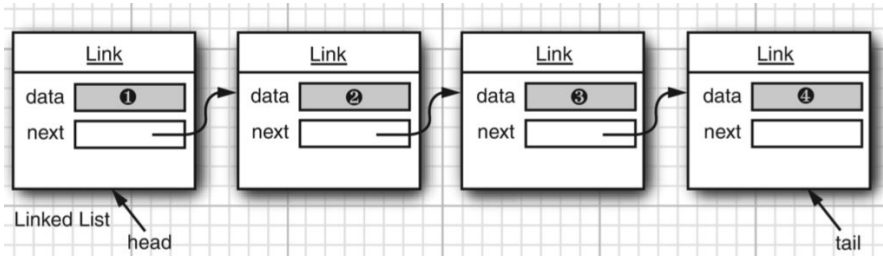
- Implementation:

```
public class CircularArrayQueue<E> implements Queue<E> …
public class LinkedListQueue<E> implements Queue<E> …
```

- When use a queue, don't need to know which implementation is actually used once the collection has been constructed.

```
Queue<Customer> expressLane = new CircularArrayQueue<>(100);
expressLane = new LinkedListQueue<>();
```





9.1 The Java Collections Framework

9.1.2 The Collection Interface

Two fundamental methods:

```
public interface Collection<E> {
    boolean add(E element);
    Iterator<E> iterator();
    ...
}
```

9.1.3 Iterators

```
public interface Iterator<E> {
    E next();
    boolean hasNext();
    void remove();
    default void forEachRemaining(Consumer<? super E> action);
}
```

9.1 The Java Collections Framework

9.1.3 Iterators

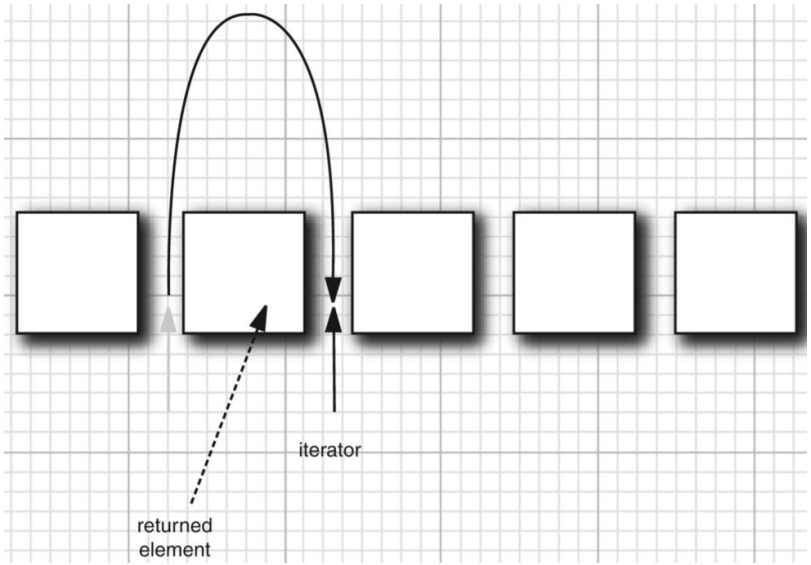
- inspect all elements in a collection:
  - request an iterator
  - keep calling the next method while hasNext returns true
- “for each” loop works with any object that implements the Iterable interface
  - collection interface extends the Iterable interface

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

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

9.1 The Java Collections Framework

9.1.3 Iterators

- Java iterators as being between elements.
- next(), the iterator
  - jumps over the next element,
  - returns a reference to the element that it just passed



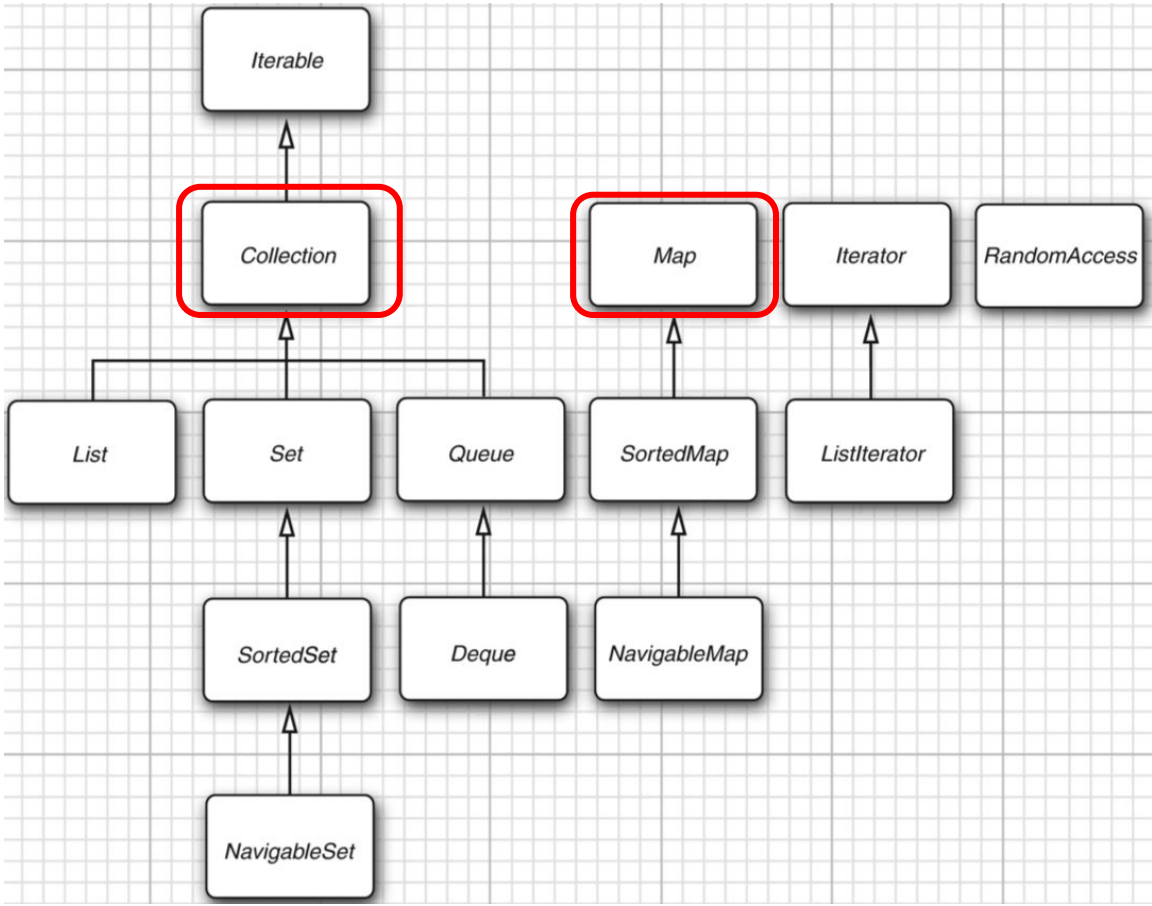

```
// remove 1st element
Iterator<String> it = c.iterator();
// skip over the first element
it.next();
// now remove it
it.remove();

//remove two adjacent elements
it.remove();
it.next();
it.remove();
```

9.1.4 Generic Utility Methods

- Collection and Iterator interfaces are generic
- Collection interface declares quite a few useful methods that all implementing classes must supply.
- Class AbstractCollection leaves the fundamental methods size and iterator abstract

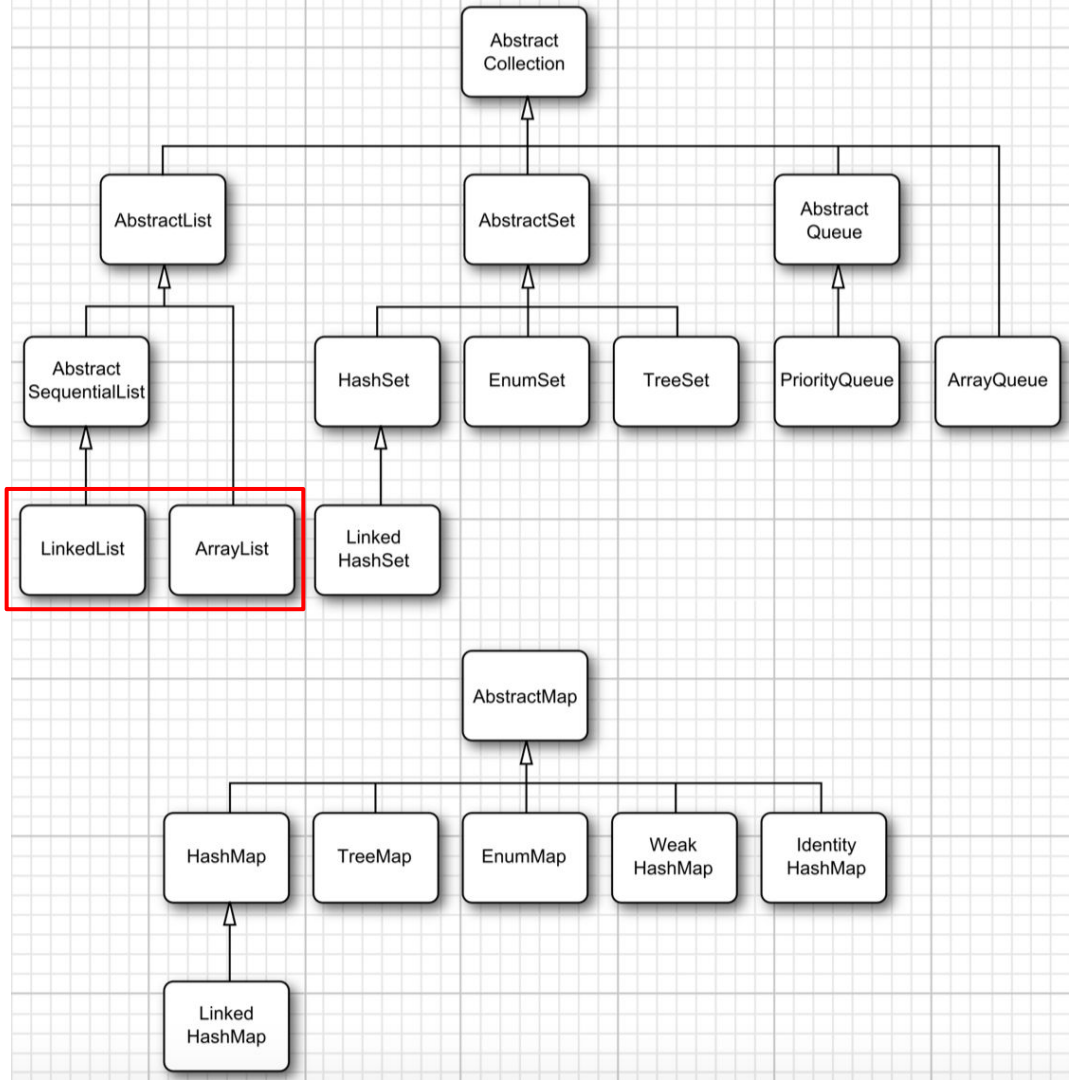
9.1.5 Interfaces in the Collections Framework

- Two fundamental interfaces for collections: Collection and Map.



9.2 Concrete Collections

- Classes in the collections framework: extend abstract classes.

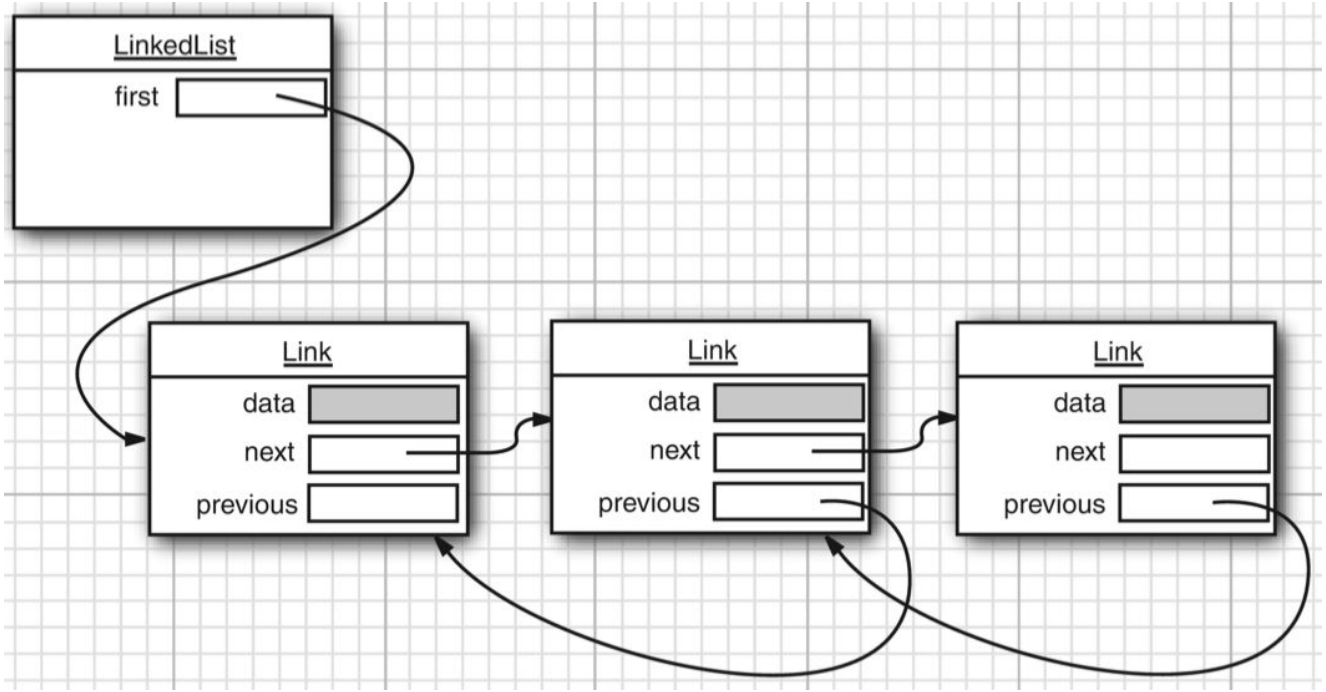


9.2.2 Array Lists

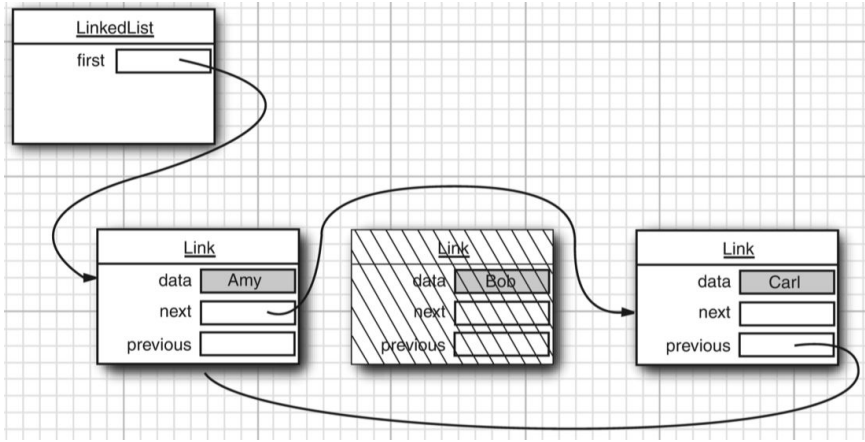
- an ordered collection
- visiting the elements:
  - an iterator
  - random access with methods get and set

9.2.1 Linked Lists

- Removing/inserting an element from/in the middle of an array: so expensive
- In Java, all linked lists are doubly linked: link to next, previous element.



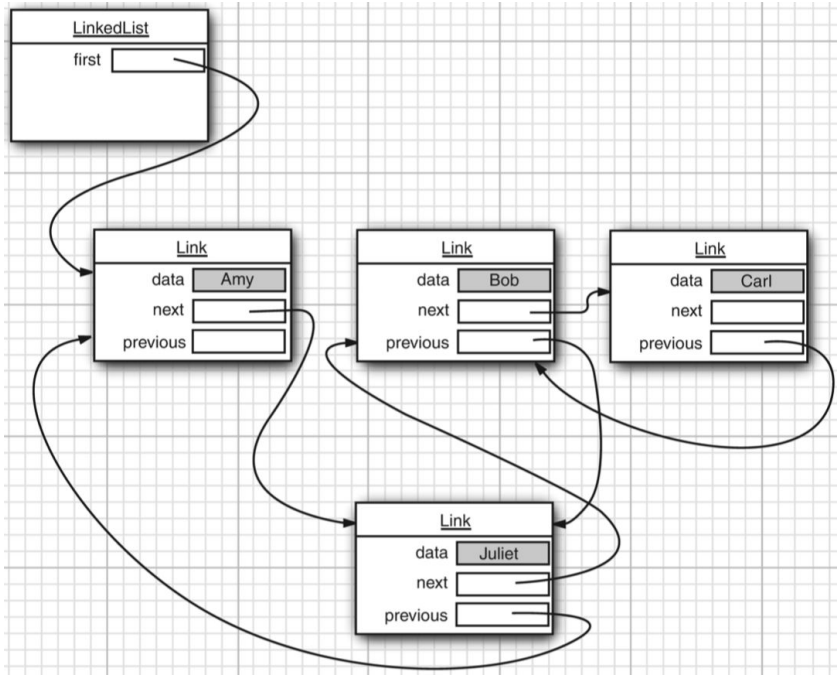
9.2.1 Linked Lists

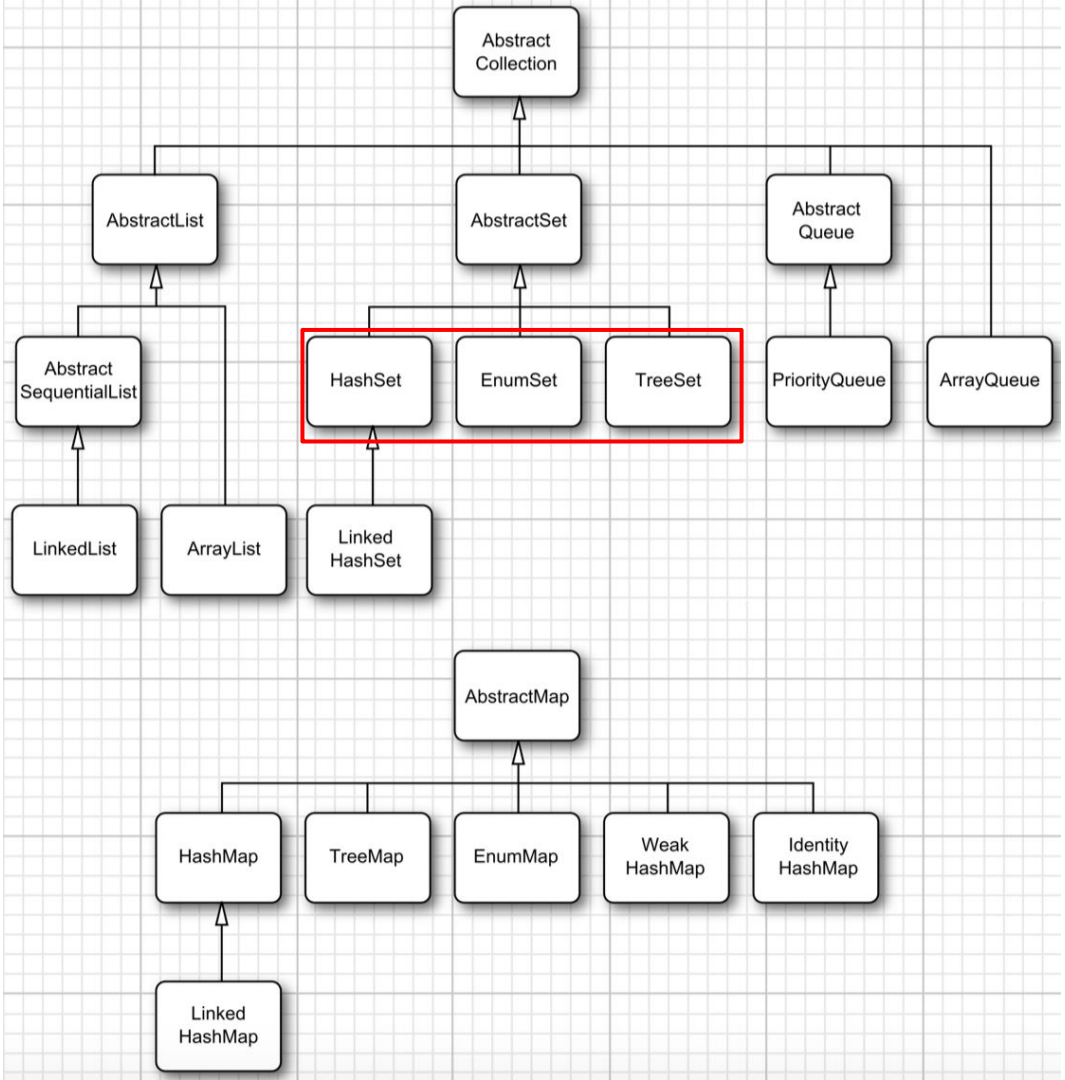



```
List<String> staff = new LinkedList<>();
// LinkedList implements List
staff.add("Amy");
staff.add("Bob");
staff.add("Carl");

Iterator iter = staff.iterator();
String first = iter.next(); // visit first element String
second = iter.next(); // visit second element
iter.remove(); // remove last visited element
```

9.2.1 Linked Lists

```
List<String> staff = new LinkedList<>();
staff.add("Amy");
staff.add("Bob");
staff.add("Carl");

ListIterator<String> iter =
staff.listIterator();
iter.next(); // skip past first element
iter.add("Juliet");
```



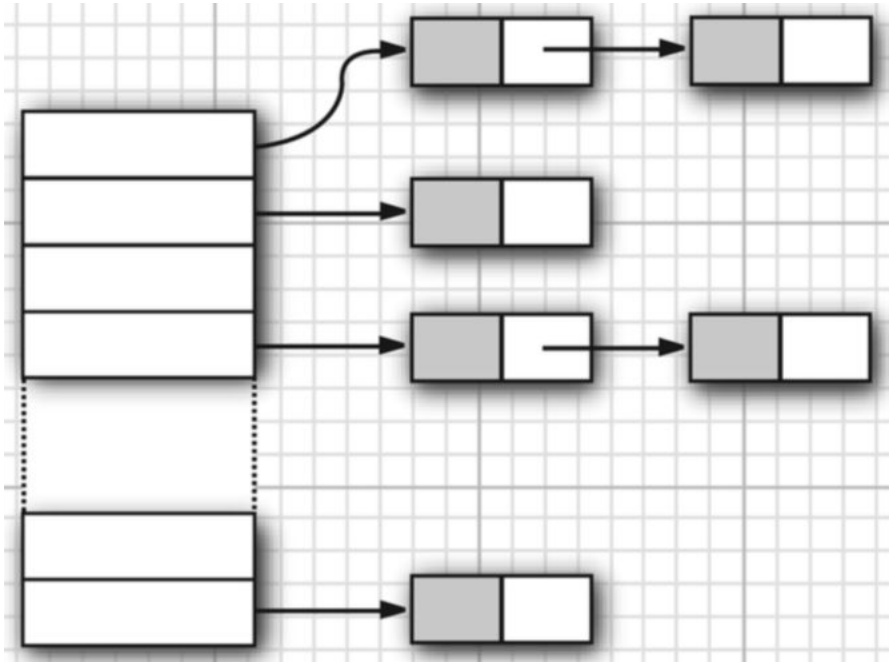

Abstract Collection
AbstractList
AbstractSet
Abstract Queue
Abstract SequentialList
HashSet
EnumSet
TreeSet
PriorityQueue
ArrayQueue
LinkedList
ArrayList
Linked HashSet
AbstractMap
HashMap
TreeMap
EnumMap
Weak HashMap
Identity HashMap
Linked HashMap

9.2.3 Hash Sets

Hash table helps to find objects quickly

In Java, hash tables are implemented as arrays of linked lists.

- each list is called a *bucket*.
- find the place of an object in the table:
  - compute its hash code
  - reduce it modulo the total number of buckets.



9.2.3 Hash Sets

HashSet class implements a set based on a hash table.

- **add():** add elements.
- contains(): make a fast lookup to see if an element is already present in the set.
- iterator visits all buckets in turn in a random order.

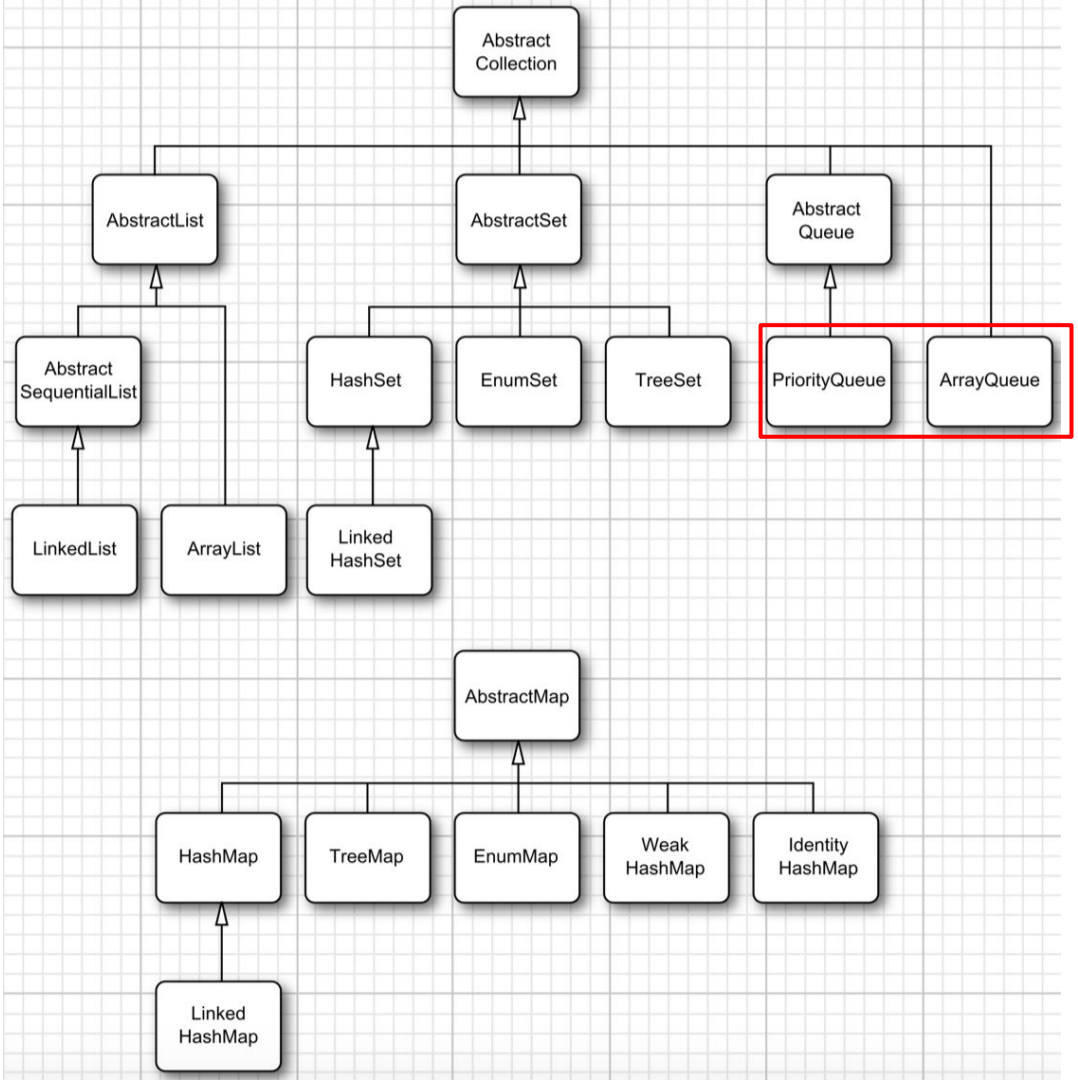
Full code: set/SetTest.java

9.2.4 Tree Sets

- TreeSet class: hashset + a sorted collection.
- Every time an element is added to a tree, it is placed into its proper sorting position.
  - The elements must implement the Comparable interface for comparing them.
- Adding an element to a tree is slower than adding it to a hash table.

Table 9.3 Adding Elements into Hash and Tree Sets

Document	Total Number of Words	Number of Distinct Words	HashSet	TreeSet
Alice in Wonderland	28195	5909	5 sec	7 sec
The Count of Monte Cristo	466300	37545	75 sec	98 sec

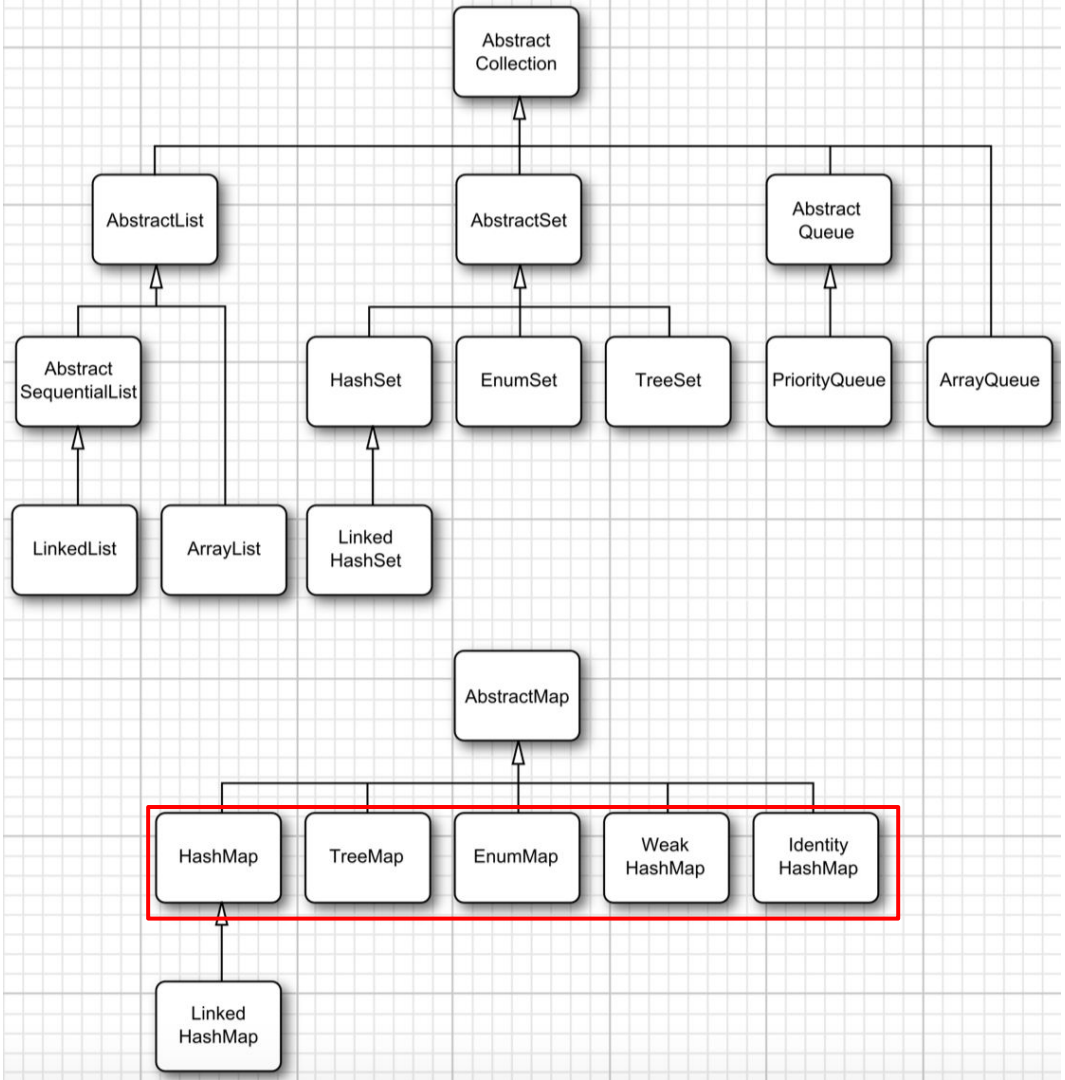

Abstract Collection
AbstractList
AbstractSet
Abstract Queue
Abstract SequentialList
HashSet
EnumSet
TreeSet
PriorityQueue
ArrayQueue
LinkedList
ArrayList
Linked HashSet
AbstractMap
HashMap
TreeMap
EnumMap
Weak HashMap
Identity HashMap
Linked HashMap

9.2.5 Queues and Deques

- A queue allows efficiently to add elements at the tail and remove elements from the head.
- A double-ended queue, or *deque*, allows efficiently to add or remove elements at the head and tail.
- Adding elements in the middle is not supported.

9.2.6 Priority Queues

- A priority queue retrieves elements in sorted order after they were inserted in arbitrary order.
  - whenever calling the remove method, you get the smallest element currently in the priority queue.
- A priority queue can either hold elements of a class that implements the Comparable interface or a Comparator object
- A typical use for a priority queue is job scheduling.
  - Each job has a priority.
  - Jobs are added in random order.
  - Whenever a new job can be started, the highest priority job is removed from the queue.


Abstract Collection
AbstractList
AbstractSet
Abstract Queue
Abstract SequentialList
HashSet
EnumSet
TreeSet
PriorityQueue
ArrayQueue
LinkedList
ArrayList
Linked HashSet
AbstractMap
HashMap
TreeMap
EnumMap
Weak HashMap
Identity HashMap
Linked HashMap

9.3 Maps

- A set is a collection that lets you quickly find an existing element.
- Usually, you have some key information, and you want to look up the associated element =>The map data structure serves that purpose.

9.3.1 Basic Map Operations

Two general-purpose implementations for maps: HashMap and TreeMap

- A hash map hashes the keys
- A tree map uses an ordering on the keys to organize them in a search tree.

Full code: map/MapTest.java

```
Map<String, Employee> staff = new HashMap<>();
// HashMap implements Map
Employee harry = new Employee("Harry Hacker");
staff.put("987-98-9996", harry);

String id = "987-98-9996";
e = staff.get(id);
// gets harry

staff.forEach((k, v) ->
System.out.println("key=" + k + ", value=" + v));
```

9.3.3 Map Views

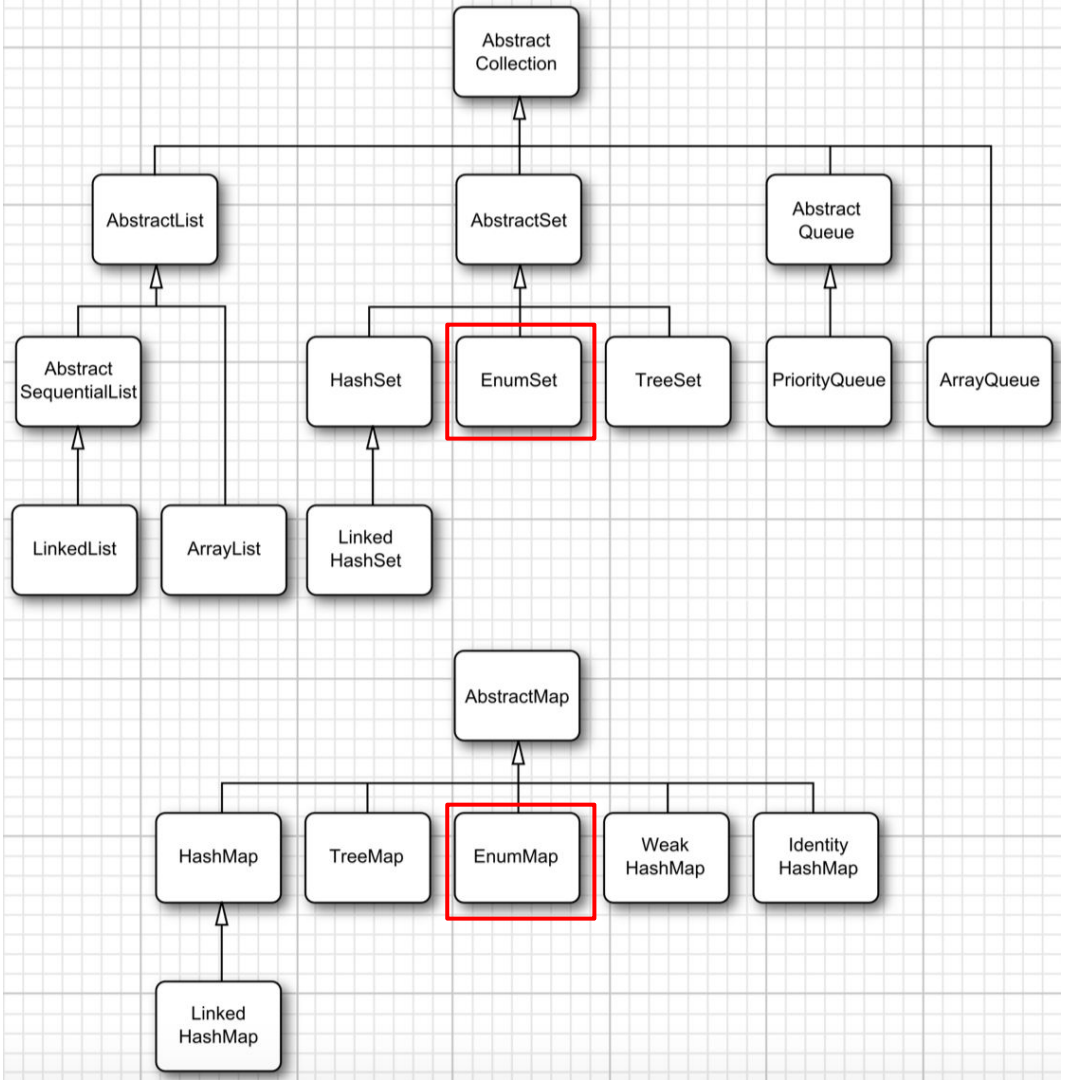
- The collections framework does not consider a map itself as a collection.
- Views of the map - objects that implement the Collection interface
- There are three views:
  - the **set of keys**: Set<K> keySet()
  - the **collection of values**: Collection<V> values()
  - the **set of key/value pairs**: Set<Map.Entry<K, V>> entrySet()

```
Set<String> keys = staff.keySet();
for (String key : keys){
     // do something with key }

for (Map.Entry<String, Employee> entry :
staff.entrySet()) {
     String k = entry.getKey();
     Employee v = entry.getValue();
     // do something with k, v
}
```

9.3.5 Linked Hash Maps

The LinkedHashMap classes remember in which order you inserted items.

```
public class LinkedHashMapTest {
        public static void main(String[] args) {
                Map<String, Employee> staff = new LinkedHashMap<>();
                staff.put("144-25-5464", new Employee("Amy Lee"));
                staff.put("567-24-2546", new Employee("Harry Hacker"));
                staff.put("157-62-7935", new Employee("Gary Cooper"));
                staff.put("456-62-5527", new Employee("Francesca Cruz"));

                System.out.println(staff);
                staff.remove("567-24-2546");
                staff.put("456-62-5527", new Employee("Francesca Miller"));
                System.out.println(staff.get("157-62-7935"));
                // iterate through all entries in the insert order:
                staff.forEach((k, v) -> System.out.println("key=" + k + ", value=" + v));
        }
}
```


Abstract Collection
AbstractList
AbstractSet
Abstract Queue
Abstract SequentialList
HashSet
EnumSet
TreeSet
PriorityQueue
ArrayQueue
LinkedList
ArrayList
Linked HashSet
AbstractMap
HashMap
TreeMap
EnumMap
Weak HashMap
Identity HashMap
Linked HashMap

9.3.6 Enumeration Sets and Maps

The EnumSet is an efficient set implementation with elements that belong to an enumerated type.

9.3.6 Enumeration Sets and Maps

The EnumSet is an efficient set implementation with elements that belong to an enumerated type.

```
import java.util.EnumSet;

public class EnumSetMap {
	enum Weekday {
		MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
	};

	public static void main(String[] args) {
		EnumSet<Weekday> always = EnumSet.allOf(Weekday.class);
		EnumSet<Weekday> never = EnumSet.noneOf(Weekday.class);
		EnumSet<Weekday> workday = EnumSet.range(Weekday.MONDAY, Weekday.FRIDAY);
		EnumSet<Weekday> mwf = EnumSet.of(Weekday.MONDAY, Weekday.WEDNESDAY, Weekday.FRIDAY);

		System.out.println(always);
		System.out.println(never);
		System.out.println(workday);
		System.out.println(mwf);
	}
}
```

9.3.6 Enumeration Sets and Maps

An EnumMap is a map with keys that belong to an enumerated type.

```
import java.util.EnumMap;

public class EnumMapTest {
        enum Weekday {
                MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
        };

        public static void main(String[] args) {
                EnumMap<Weekday, Employee> personInCharge = new EnumMap<>(Weekday.class);
                personInCharge.put(Weekday.MONDAY, new Employee("An"));
                personInCharge.put(Weekday.TUESDAY, new Employee("Binh"));

                // Retrieving value from EnumMap
      System.out.println("Key : " + Weekday.MONDAY + " Value: "
                        + personInCharge.get(Weekday.MONDAY));
        }
}
```

9.4 Views and Wrappers

9.4.1 Lightweight Collection Wrappers

- The static asList method of the Arrays class returns a List wrapper around a plain Java array.

```
Card[] cardDeck = new Card[52];
List<Card> cardList = Arrays.asList(cardDeck);
```

- The returned object is not an ArrayList.
- It is a view object with **get** and **set** methods that access the underlying array.
- All methods that would change the size of the array throw an UnsupportedOperationException.

9.4 Views and Wrappers

9.4.2 Subranges

Form subrange views for a number of collections

```
List group2 = staff.subList(10, 20);
```

Apply any operations to the subrange, and they automatically reflect the entire list

```
group2.clear(); // staff reduction, group2 becomes empty
```

9.4 Views and Wrappers

9.4.2 Subranges

The SortedSet interface declares three methods:

SortedSet<E> subSet(E from, E to)
SortedSet<E> headSet(E to)
SortedSet<E> tailSet(E from)

return the subsets of all elements that are larger than or equal to from and strictly smaller than to.

SortedMap<K, V> subMap(K from, K to)
SortedMap<K, V> headMap(K to)
SortedMap<K, V> tailMap(K from)

return views into the maps consisting of all entries in which the keys fall into the specified ranges.

9.4 Views and Wrappers

9.4.3 Unmodifiable Views

The Collections class has methods that produce unmodifiable views of collections.

- These views add a runtime check to an existing collection.
- If an attempt to modify the collection is detected, an exception is thrown and the collection remains untouched.

Collections.unmodifiableCollection
Collections.unmodifiableList
Collections.unmodifiableSet
Collections.unmodifiableSortedSet
Collections.unmodifiableMap
Collections.unmodifiableSortedMap

9.4 Views and Wrappers

```
import java.util.*;

public class UnmodifiableTest {
	public static void main(String[] argv) throws Exception {
		try {

			List<String> list = new ArrayList<String>(); // creating object of ArrayList<Character>

			list.add("X"); list.add("Y"); // populate the list


			System.out.println("Initial list: " + list); // printing the list

			List<String> immutablelist = Collections.unmodifiableList(list); // getting unmodifiable list

			System.out.println("Unmodifiable list: " + immutablelist); // printing the list

			immutablelist.add("Z"); // add element to the immutable list

		} catch (UnsupportedOperationException e) {
			System.out.println("Exception thrown : " + e);
		}
	}
}
```

9.4 Views and Wrappers

9.4.4 Synchronized Views

Use the view mechanism to make regular collections thread safe; access the map object from multiple threads

```
Map<String, Employee> map = Collections.synchronizedMap(new HashMap<String, Employee>());
```

9.4 Views and Wrappers

9.4.5 Checked Views

Checked views are intended as debugging support for a problem that can occur with generic types.

```
ArrayList<String> strings = new ArrayList<>();
List<String> safeStrings = Collections.checkedList(strings, String.class);
safeStrings.add(new Date()); // checked list throws a ClassCastException
```

9.5 Algorithms

Generic collection interfaces have a great advantage: you only need to implement your algorithms once.

```
public static <T extends Comparable> T max(Collection<T> c) {
        if (c.isEmpty()) throw new NoSuchElementException();
        Iterator<T> iter = c.iterator();
        T largest = iter.next();
        while (iter.hasNext()){
                T next = iter.next();
                if (largest.compareTo(next) < 0)
                largest = next;
        }
        return largest;
}
```

=> compute the maximum of a linked list, an array list, or an array, with a single method

9.5 Algorithms

9.5.1 Sorting and Shuffling

- Java dumps all elements into an array, sorts the array, and then copies the sorted sequence back into the list.
- a bit slower than QuickSort, however, it is stable and doesn't switch equal elements.

```
public class ShuffleSortTest
{
   public static void main(String[] args)
   {
      List<Integer> numbers = new ArrayList<>();
      for (int i = 1; i <= 49; i++)
         numbers.add(i);

      Collections.shuffle(numbers);

      List<Integer> winningList = numbers.subList(0, 6);
      Collections.sort(winningList);

      System.out.println(winningList);
   }
}
```

9.5 Algorithms

9.5.2 Binary Search

- Searches for the specified object using the binary search algorithm.
- The list must be sorted into ascending order prior to making this call.
- The binarySearch algorithm reverts to a linear search if you give it a linked list.

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class SearchTest {
	public static void main(String[] args)
	{
		List<Integer> al = new ArrayList<Integer>();
		al.add(10);
		al.add(1);
		al.add(2);
		al.add(3);
		al.add(20);

		Collections.sort(al);

		int index = Collections.binarySearch(al, 10);
		System.out.println(index);

		index = Collections.binarySearch(al, 13);
		System.out.println(index);
	}
}
```

9.5 Algorithms

9.5.3 Simple Algorithms

- The Collections class contains several simple but useful algorithms.
  - min, max
  - copy
  - fill
  - addAll, replaceAll,
  - …
- They make life easier for the programmer reading the code

9.5 Algorithms

9.5.4 Bulk Operations

There are several operations that copy or remove elements “in bulk.”

```
// coll1, coll2 as collections
coll1.removeAll(coll2); // removes all elements from coll1 that are present in coll2
coll1.retainAll(coll2); // removes all elements from coll1 that are not present in coll2.
```

9.5 Algorithms

9.5.5 Converting between Collections and Arrays

Large portions of the Java platform API were designed before the collections framework was created.

=> translate between traditional arrays and the more modern collections

```
public class ConvertingTest {
        public static void main(String[] args) {
                String[] values = { "a", "b", "c" };
                HashSet<String> staff = new HashSet<>(Arrays.asList(values));
                System.out.println(staff);
                String[] staffArr = staff.toArray(new String[staff.size()]);
                System.out.println(staffArr[0]);
        }
}
```