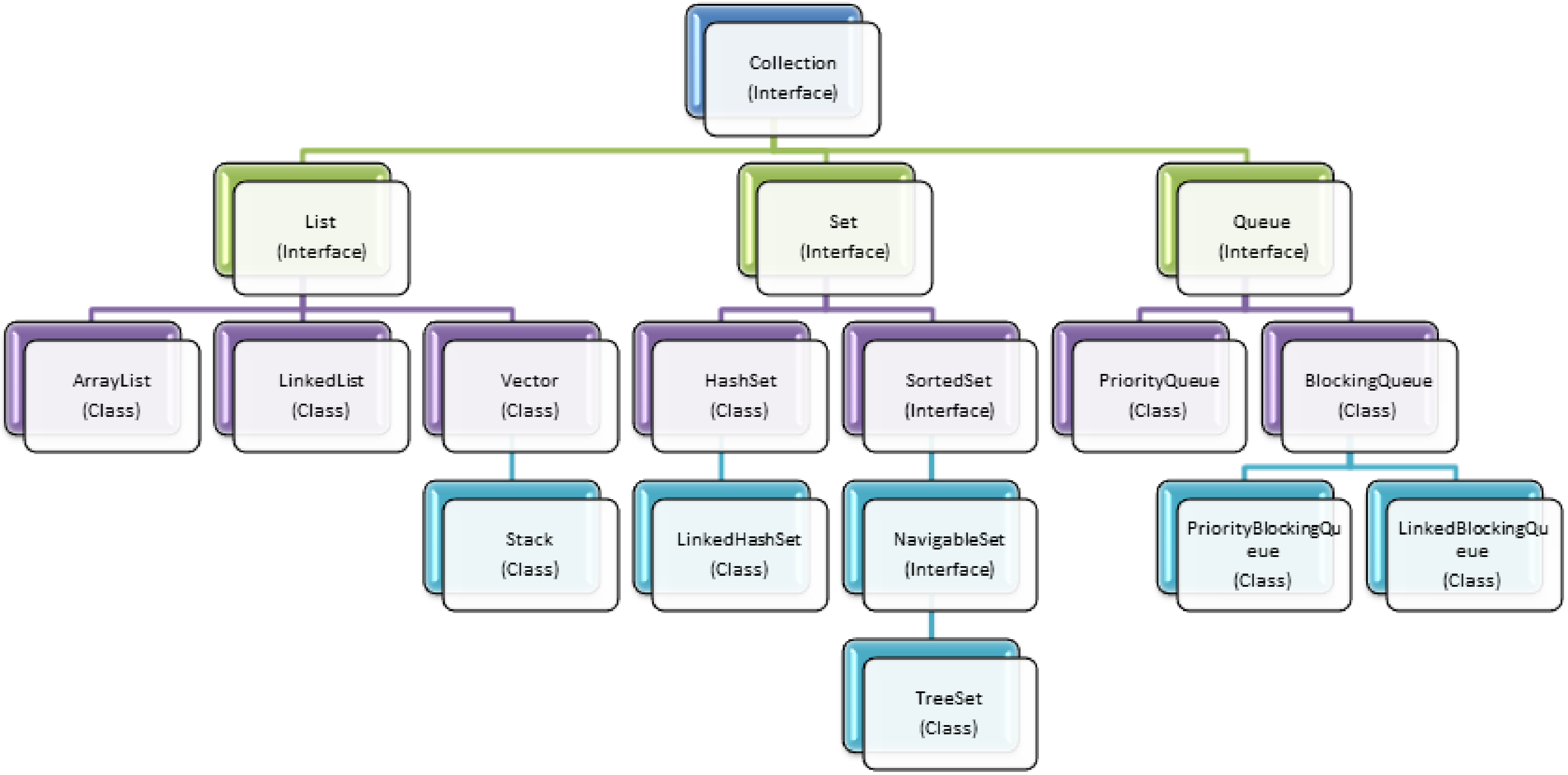
Collections

by Vlad Costel Ungureanu
for Learn Stuff IO



Collection (Interface)
List (Interface)
Set (Interface)
Queue (Interface)
ArrayList (Class)
LinkedList (Class)
Vector (Class)
HashSet (Class)
SortedSet (Interface)
PriorityQueue (Class)
BlockingQueue (Class)
Stack (Class)
LinkedHashSet (Class)
NavigableSet (Interface)
PriorityBlockingQueue (Class)
LinkedBlockingQueue (Class)
TreeSet (Class)

Collections – Operations

- ✓ **Add** objects to the collection
- ✓ **Remove** objects from the collection
- ✓ **Find** out if an object (or group of objects) is in the collection
- ✓ **Retrieve** an object from the collection (without removing it)
- ✓ **Iterate** through the collection, looking at each element (object) one after another

Collections – Equals()

The equals() contract:

- ✓ it is **reflexive** - for any reference value x, x.equals(x) should return true
- ✓ it is **symmetric** - for any reference values x and y, x.equals(y) should return true if and only if y.equals(x) returns true
- ✓ it is **transitive** - for any reference values x, y, and z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) must return true
- ✓ it is **consistent** - for any reference values x and y, multiple invocations of x.equals(y) consistently return true or consistently return false, provided no information used in equals comparisons on the object is modified
- ✓ for any non-null reference value x, x.equals(null) **should return false**

Collections – HashCode()

- If two objects are considered equal, their hash codes must also be equal!

- If the hashCode() method of a class is <u>not</u> overridden:

✓ it will be used the one from Object class which always comes up with a unique number for each object, even if the equals() method is overridden in such a way that two or more objects are considered equal

✓ the object in a collection won't be found unless the same reference to the object is given

Collections – HashCode()

The hashCode() contract:

- ✓ whenever it is invoked on the same object more than once during an execution of a Java application, the hashCode() method must consistently return the same integer, provided no information used in equals() comparisons on the object is modified

- ✓ if two objects are equal according to the equals(Object) method, then calling the hashCode() method on each of the two objects must produce the same integer result

- ✓ it is NOT required that if two objects are unequal according to the equals(java.lang.Object) method, then calling the hashCode() method on each of the two objects must produce distinct integer results; however, producing distinct integer results for unequal objects may improve the performance of Maps

List Interface

- ✓**ArrayList**
  - ✓fast iteration and fast random access
  - ✓it is an ordered collection (by index), but not sorted
  - ✓as of version 1.4, ArrayList implements the RandomAccess interface

- ✓**Vector**
  - ✓it is basically the same as an ArrayList, but Vector methods are synchronized for thread safety
  - ✓Vector is the only class other than ArrayList to implement RandomAccess

- ✓**LinkedList**
  - ✓a LinkedList is ordered by index position, like ArrayList, except that the elements are doubly-linked to one another
  - ✓this linkage gives methods for adding and removing from the beginning or end
  - ✓as of Java 5, the LinkedList class has been enhanced to implement the java.util.Queue interface
  - ✓it supports the common queue methods: peek(), poll() and offer()

Collections – Iteration in Lists

✓ **Index based collections**

```
for (i=0; i < list.size(); i++ ) {
        System.out.println( list.get(i) );
}
```

✓ **Iterators**

```
for (Iterator it = collection.iterator(); it.hasNext(); ) {
        System.out.println(it.next());
        it.remove();
}
```

✓ **"For-each" structure – Object based access**

```
List<Student> students = new ArrayList<Student>();
for (Student student : students) {
        student.setAge(ageValue);
}
```

Set Interface

- ✓ **HashSet**
  - ✓ it uses the hash code of the object being inserted
  - ✓ it is unsorted, unordered
  - ✓ use this class when you want a collection with no duplicates and you don't care about order when you iterate through it
- ✓ **LinkedHashSet**
  - ✓ it is an ordered version of HashSet that maintains a doubly-linked List across all elements
  - ✓ maintains insertion order
- ✓ **TreeSet**
  - ✓ the TreeSet is one of two sorted collections
  - ✓ it guarantees that the elements will be in ascending order, according to natural order
  - ✓ as of Java 6, TreeSet implements NavigableSet

Collections – Iteration in Sets

```
HashSet<String> mySet = new HashSet<String>();

 //add elements to HashSet
mySet.add(“Vlad");
mySet.add(“John");

 Iterator<String> it = mySet.iterator();
 while(it.hasNext()){
    System.out.println(it.next());
 }
```

Map Interface

- ✓ **HashMap**
  - ✓ it is an unsorted, unordered Map
  - ✓ allows one null key and multiple null values
- ✓ **Hashtable**
  - ✓ it is the synchronized counterpart to HashMap
  - ✓ doesn't allow null keys or null values
- ✓ **LinkedHashMap**
  - ✓ maintains insertion order of keys (default) or access order
  - ✓ faster iteration
- ✓ **TreeMap**
  - ✓ it is a sorted Map
  - ✓ as of Java 6, TreeMap implements NavigableMap

Collections – Iteration in Maps

```
Map<Integer, Student> map = new HashMap<>();

map.put(Integer.valueOf(1), new Student("Ungureanu", "Vlad", 1));
map.put(Integer.valueOf(2), new Student(“John", “Doe", 2));

System.out.println(map.get(1));

// key not found - will return null
System.out.println(map.get("1"));

// iterate through map
for (Map.Entry<Integer, Student> entry : map.entrySet()) {
  System.out.println(entry.getKey() + " / " + entry.getValue());
}
```

Sorting Collections

Methods for sorting are given by java.util.Collections and by java.util.Arrays classes:

Collections.sort(List<T> list)
- ✓ it uses the natural order of the elements for sorting
- ✓ an exception is thrown when trying to sort elements of class that doesn’t have defined the natural order
- ✓ to define a sorting order, a class can implement the Comparable interface

Collections.sort(List<T> list, Comparator<? super T> c)
- ✓ sorts the specified list according to the order induced by the specified comparator
- ✓ the Comparator interface gives you the capability to sort a given collection any number of different ways

Assignments

- ✓ Write a class that represents a money transaction between banks.
- ✓ Write a program that stores several transactions in various types of collections: ArrayList, LinkedList, Map, Vector, Set. Make sure that each to add the same transactions(same instance) to each collection.
- ✓ Override equals() and hashCode() for the transaction class.
- ✓ Iterate the collections and for each collection perform the following: add, remove, compare to a specific transaction (using equals), modify properties of a transaction.
- ✓ Observe behavior of the previous operations and calculate for each operation, for each transaction the time each operation took. Write the times in a file and compare times between collection types.

THANK YOU!

Vlad Costel Ungureanu

contact@learnstuffacademy.io