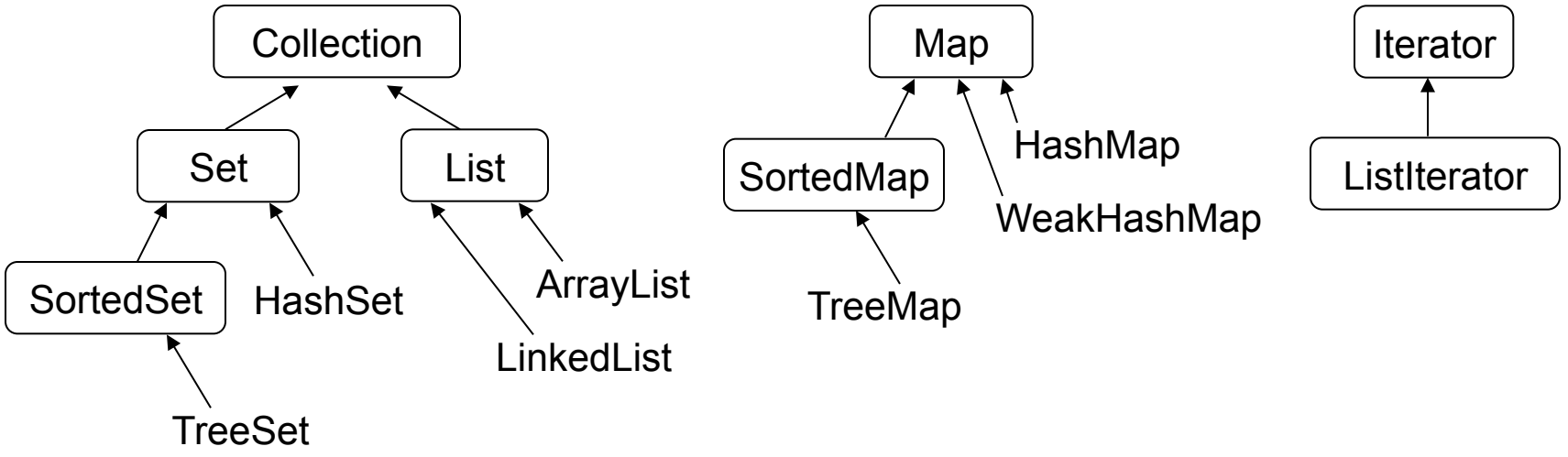
Collections

By Pakita Shamoi

Concept

- A collection is a data structure – actually, an object – to hold other objects, which let you store and organize objects in useful ways for efficient access
- Check out the `java.util` package! Lots of interfaces and classes providing a general collection framework.
- Overview of the interfaces and concrete classes in the *collection framework*



Root interface – Collection (1)

- Methods working with an individual collection
  - `public int size()`
  - `public boolean isEmpty()`
  - `public boolean contains(Object elem)`
  - `public boolean add(Object elem)`
    - Depends on whether the collection allows duplicates
  - `public boolean remove(Object elem)`
  - `public boolean equals(Object o)`
  - `public Iterator iterator()`
  - `public void clear()`
    - Remove all elements from this collection
  - `public Object[] toArray()`
    - Returns a new array containing references to all the elements of the collection
  - `public Object[] toArray(Object[] dest)`

Root interface – Collection (2)

- Primary methods operating together with another collection:
  - `public boolean containsAll(Collection coll)`
  - `public boolean addAll(Collection coll)`
    - Returns true if any addition succeeds. **Which logical operation is it?**
  - `public boolean removeAll(Collection coll)`
    - Returns true if any removal succeeds. **Which logical operation is it?**
  - `public boolean retainAll(Collection coll)`
    - Removes from the collection all elements that are not elements of `coll`. **Which logical operation is it?**

- The SDK does NOT provide any direct implementations of the `Collection` interface
  - Most of the actual collection types implement this interface, usually by implementing an extended interface such as `Set` or `List`

Iteration - Iterator

- The `Collection` interface defines an `iterator` method to return an object implementing the `Iterator` interface.
  - It can access the elements in a collection without exposing its internal structure.
  - There are **NO** guarantees concerning the order in which the elements are returned
- Three defined methods in Iterator interface
  - `public boolean` **`hasNext()`** – returns true if the iteration has more elements
  - `public Object` **`next()`** – returns the next element in the iteration
    - An exception will be thrown if there is no next element
    - What's returned is an `Object` object. You may need special casting!
  - `public void` **`remove()`** – remove from the collection the element last returned by the iteration
    - can be called only once per call of `next`, otherwise an exception is thrown

classical routine of using iterator:

```
public void removeLongStrings (Collection coll, int
  maxLen) {
        Iterator it = coll.iterator();
        while ( it.hasNext() ) {
           String str = (String)it.next();
           if (str.length() > maxLen)
           it.remove()
        }
  }
```

Iteration - ListIterator

- `ListIterator` interface extends `Iterator` interface. It adds methods to manipulate an ordered `List` object during iteration
- Methods:
  - `public boolean` **`hasNext()`**`/ public boolean` **`hasPrevious()`**
  - `public Object` **`next()`**`/ public Object` **`previous()`**
  - `public Object` **`nextIndex()`**`/ public Object` **`previousIndex()`**
    - When it's at the end of the list, `nextIndex()` will return `list.size()`
    - When it's at the beginning of the list, `previousIndex()` will return -1
  - public void **`remove()`** – remove the element last returned by `next()` or `previous()`
  - public void **`add(Object o)`** – insert the object o into the list in front of the next element that would be returned by `next()`, or at the end if no next element exists
  - public void **`set(Object o)`** – set the element last returned by `next()` or `previous()` with `o`

Potential problem of Iterator/ListIterator

- They do NOT provide the ***snapshot*** guarantee – if the content of the collection is modified when the iterator is in use, it can affect the values returned by the methods
- A snapshot will return the elements as they were when the `Iterator / ListIterator` object was created
- If you really need a snapshot, make a simple copy of the collection

```
import java.util.*;
public class IteratorTest {
  public static void main (String args[]) {
    ArrayList a = new ArrayList();
    a.add("1");
    a.add("2");
    a.add("3");

    Iterator it = a.iterator();
    while(it.hasNext()) {
      String s = (String)(it.next());
      if(s.equals("1")) {
        a.set(2,"changed");
      }

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

Output?

1
2
changed

```
import java.util.*;

public class IteratorTest2 {
  public static void main (String args[]) {
    ArrayList a = new ArrayList();
    a.add("1");
    a.add("2");
    a.add("3");

    Iterator it = a.iterator();

    a.add("4");

    while(it.hasNext()) {
      String s = (String)(it.next());
      System.out.println(s);
    }
  }
}
```

%> javac IteratorTest2.java
%> java IteratorTest2

Exception in thread "main" java.util.ConcurrentModificationException

Traversing the collection

- So, there are two schemes of traversing collections:
  - Iterator (already covered)
  - for-each
    - The for-each construct allows you to concisely traverse a collection or array using a for loop

```
for (Object o: collection)
    System.out.println(o);
```

Example:

```
for (String str: myHashSet)
    System.out.println(str);
```

List

- A **`List`** is an ordered Collection which allows duplicate elements. Its element indices range from 0 to (`list.size()`**-1**)
- It adds several methods for an ordered collection
- The interface `List` is implemented by these classes:

1. **`ArrayList, Vector:`** a resizable-array implementation of the `List` interface
   - Adding or removing elements at the end, or getting an element at a specific position is simple – *O(1)*
   - Adding or removing element from the middle is expensive – *O(n-i)*
   - Can be efficiently scanned **by using the indices** without creating an `Iterator` object, so it's good for a list which will be scanned frequently
2. **`LinkedList:`** a doubly-linked list
   - Getting an element at position *i* is more expensive – *O(i)*
   - A good base for lists where most of the actions are not at the end

Set and SortedSet

- The **`Set`** interface provides a more specific contract for its methods, but adding no new methods of its own. A `Set` is a `Collection` that contains ***UNIQUE*** elements.

- The **`SortedSet`** extends `Set` to specify an additional contract – iterators on such a set return the elements in a specified order
  - By default it will be the elements' *natural order* which is determined by the implementation of `Comparable` interface
  - You can specify a `Comparator` object to order the elements instead of the natural order

- There are two widely used implementations of `Set` in the collection framework:
  - **HashSet** – a `Set` implemented using a hashtable
  - **TreeSet** – a `SortedSet` implemented in a balanced tree structure

Map and SortedMap

- The **`Map`** interface does not extend `Collection` interface because a `Map` contains **key-value pairs**, not only keys. Duplicate keys are not allowed in a `Map`. It's implemented by classes `HashMap` and `TreeMap`.

- There are methods to view the map using collections. For example: `public Set` **`keySet()`** and `public Collection` **`values()`**.
  - The collections returned by these methods are backed by the Map, so removing an element from one these collections removes the corresponding key/value pair from the map
  - You cannot add elements to these collections
  - If you iterate through the key or value sets, they may return values from their respective sets in any order

- Interface **`SortedMap`** extends `Map` and maintains its keys in sorted order. Class **`TreeMap`** implements `SortedMap`.

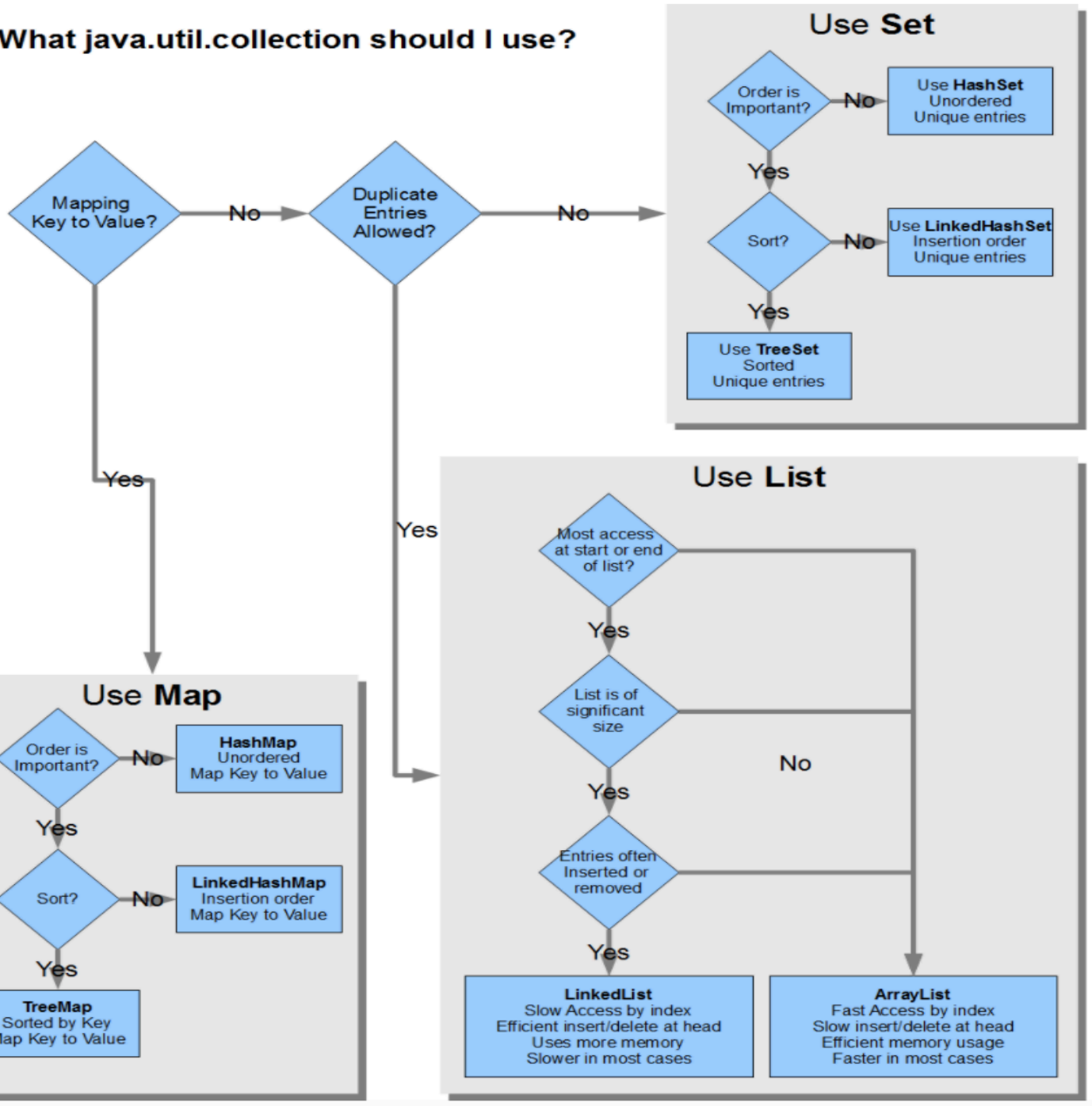
Collections class

- Collections is an utility class that provides static methods
- The first argument is the collection on which the operation is to be performed
  - Collections.sort(c);
  - Collections.shuffle(c);
  - Collections.reverse(c);
  - Collections.fill(c, “abc”);
  - Collections.binarySearch(c, “OOP");
  - Collections.frequence(c, “OOP");
  - Collections.disjoint(c1, c2);

"bread and butter" collection types

Collection type	Functionality	Typical uses
List	• Essentially a variable-size array; • You can usually add/remove items at any arbitrary position; • The order of the items is well defined (i.e. you can say what position a given item goes in in the list).	Most cases where you just need to store or iterate through a "bunch of things" and later iterate through them.
Set	• Things can be "there or not" – when you add items to a set, there's no notion of *how many times* the item was added, and usually no notion of ordering.	• Remembering "which items you've already processed", e.g. when doing a web crawl; • Making other *yes-no decisions* about an item, e.g. "is the item a word of English", "is the item in the database?" , "is the item in this category?" etc.
Map	• Stores an *association* or mapping between "keys" and "values"	Used in cases where you need to say "for a given X, what is the Y"? It is often useful for implementing in-memory caches or indexes. For example: • For a given user ID, what is their cached name/User object? • For a given IP address, what is the cached country code? • For a given string, how many instances have I seen?
Queue	• Like a list, but where you **only ever access the *ends*** of the list (typically, you add to one end and remove from the other).	• Often used in **managing tasks** performed by different threads in an application (e.g. one thread receives incomming connections and puts them on a queue; other "worker" threads take connections off the queue for processing); • For **traversing hierarchical structures** such as a filing system, or in general where you need to remember "**what data to process next**", whilst also adding to that list of data; • Related to the previous point, queues crop up in various algorithms, e.g. build the encoding tree for *Huffman compression*.

What java.util.collection should I use?



HW, as usually ☺

- Check out the following classes and interfaces at java API, and make examples using some common methods
- Also prove that the utility methods in Collections class behave correspondingly (make examples)