Java's Collection Framework

Another use of polymorphism via interfaces

Rick Mercer

More useful Polymorphism

- Where we are at?
  - Considering examples of Java's polymorphism via interfaces
- Where are we going?
  - Consider a framework that is
    - Full of examples of polymorphism through interfaces
    - Reusable
      - Note: Some of you have seen a few of these slides

Outline

- Java's Collection Framework
  - Unified architecture for representing and manipulating collections
- Collection framework contains
  - Interfaces (ADTs): specification not implementation
  - Concrete implementations as classes
  - Polymorphic Algorithms to search, sort, find, shuffle, ...
- Algorithms are *polymorphic*:
  - the same method can be used on many different implementations of the appropriate collection interface. In essence, algorithms are reusable functionality.

The Core Collection interfaces

there are others

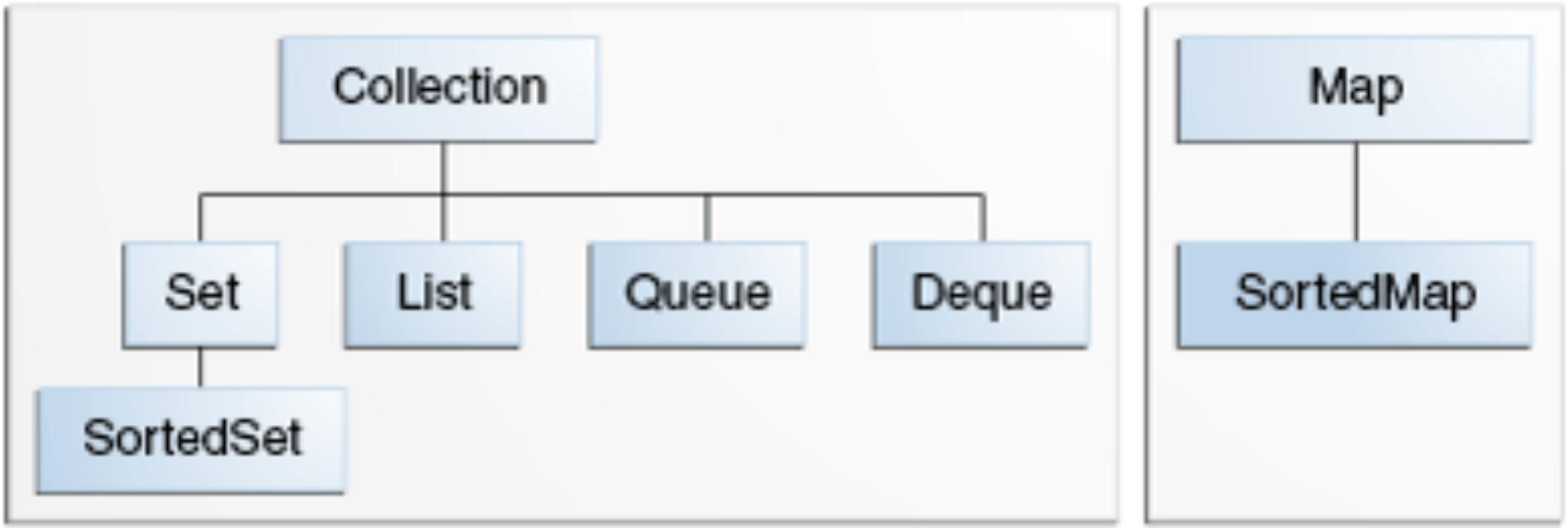


Image from the Java Tutorial

Abstract Data Type

- Abstract data type (ADT) is a specification of the behaviour (methods) of a type
  - Specifies method names to add, remove, find
  - Specifies if elements are unique, indexed, accessible from only one location, mapped,...
  - An ADT shows no implementation
    - no structure to store elements, no implemented algorithms
- What Java construct nicely specifies ADTs?

List<E>, an ADT written as a Java interface

- **`interface`** `List<E>` defines a collection with a first element, a last, and distinct predecessors and successors, can insert anywhere
  - duplicates that "`equals`" each other are allowed

```
List<String> list = new ArrayList<>();
list.add("Abc");
list.add(0, "Def");
assertEquals("Def", list.get(0));
assertEquals("Abc", list.get(1));
```

Iterators

- Iterators provide a general way to traverse all elements in a collection

```
List<String> list = new ArrayList<>();
list.add("1-FiRsT");
list.add("2-SeCoND");
list.add("3-ThIrD");
Iterator<String> itr = list.iterator();
while (itr.hasNext())
  System.out.println(itr.next().toLowerCase());
```

Output

```
1-first
2-second
3-third
```

New way to visit elements: Java's Enhanced `for` *Loop*

- General form

  for (*Type element* **:** *collection***) {**

  element is the next thing visited each iteration

  }

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

Polymorphic Algorithms

- Java has *polymorphic* algorithms to provide functionality for different types of collections

```
void sort(List<T> list)
void reverse(List<T> list)
void swap(List<T> list, int i, int j)
boolean replaceAll(List<T> list, T oldVal, T newVal)
void rotate(List<?> list, int distance) // ? == any type
```

- And since `List` extends **`interface`** `Collection`

```
int frequency(Collection<?> c, Object o)
public static <T> Collection<T> // T is lower bound
    unmodifiableCollection(Collection<? extends T> c)
```

Lower Bound / Upper Bound / Wild Card

- `<E extends Comparable <E>>`
  - Extends is used as an upper bound of the given type
  - E must be Comparable or anything that implements Comparable or extends that interface
- `<? super Integer>`
  - The wildcard `?` is any type
  - **`super`**, means a superclass or super interface
  - Possible types for `Integer`: `Number` and `Object`
- This topic will NOT be on the test

But this will *intuitive enough?*

```
// Assume list has these added: A, B, C, then A
assertEquals("[A, B, C, A]", list.toString());
assertEquals("A", Collections.min(list));
assertEquals("C", Collections.max(list));
assertEquals(2, Collections.frequency(list, "A"));

Collections.swap(list, 0, 1);
assertEquals("[B, A, C, A]", list.toString());
Collections.sort(list);
assertEquals("[A, A, B, C]", list.toString());
assertEquals(2, Collections.binarySearch(list, "B"));
int index = Collections.binarySearch(list, "A");
assertTrue(index == 0 || index == 1);

Collections.rotate(list, 2);
assertEquals("[B, C, A, A]", list.toString());
```

Set and SortedSet

- **`interface`** `Set<E>`
  - `add addAll remove size` but no get!
- Two classes that `implement Set<E>`
  - `TreeSet`: values stored in order, O(log n)
  - `HashSet`: values in a hash table, no order, O(1)
- `SortedSet` extends `Set` by adding methods E

```
E first()  E last()
SortedSet<E> tailSet(E fromElement),
SortedSet<E> headSet(E fromElement)
SortedSet<E> subSet(E fromElement, E toElement)
```

TreeSet elements are in order

```
Set<String> names = new TreeSet<>();

names.add("Dakota");
names.add("Devon");
names.add("Chris");
names.add("Chris"); // not added

for (String name : names)
  System.out.print(name + " "); // Chris Dakota Devon
```

What if change to `HashSet<>()`

Chris Devon Dakota

The Map Interface (ADT)

- Map describes a type that stores a collection of elements that consists of a *key* and a *value*
- A Map associates (maps) a key the it's value
- The keys must be unique
  - the values need not be unique
  - **`put`** destroys one with same key

`interface Map<K, V>`

`public V put(K key, V value)`

- associates key to value and stores mapping

`public V get(Object key)`

- associates the value to which key is mapped or null

`public boolean containsKey(Object key)`

- returns true if the Map already uses the key

`public V remove(Object key)`

- returns previous value associated with specified key, or null if there was no mapping for key.

`Collection<V> values()`

- get a collection you can iterate over

keySet and Values

```
Map<String, Integer> rankings = new HashMap<>();
rankings.put("M", 1);
rankings.put("A", 2);
rankings.put("P", 3);
Set<String> keys = rankings.keySet();
System.out.println(keys.getClass());
System.out.println(keys);          // [P, A, M]
Collection<Integer> values = rankings.values();
System.out.println(values.getClass());
System.out.println(values);        // [3, 2, 1]
```

Change to `Tree`

Output

```
class java.util.HashMap$KeySet
[P, A, M]
class java.util.HashMap$Values
[3, 2, 1]
```

```
class java.util.TreeMap$KeySet
[A, M, P]
class java.util.TreeMap$Values
[2, 1, 3]
```

`interface Queue<E>`

`boolean` `add(E e)` Inserts e into this queue

`E element()` Retrieves, but does not remove, the head of this queue

`boolean` `offer(E e)` Inserts e into this queue

`E peek()` Retrieves, but does not remove, the head of this queue, or returns null if this queue is empty

`E poll()` Retrieves and removes the head of this queue, or returns null if this queue is empty

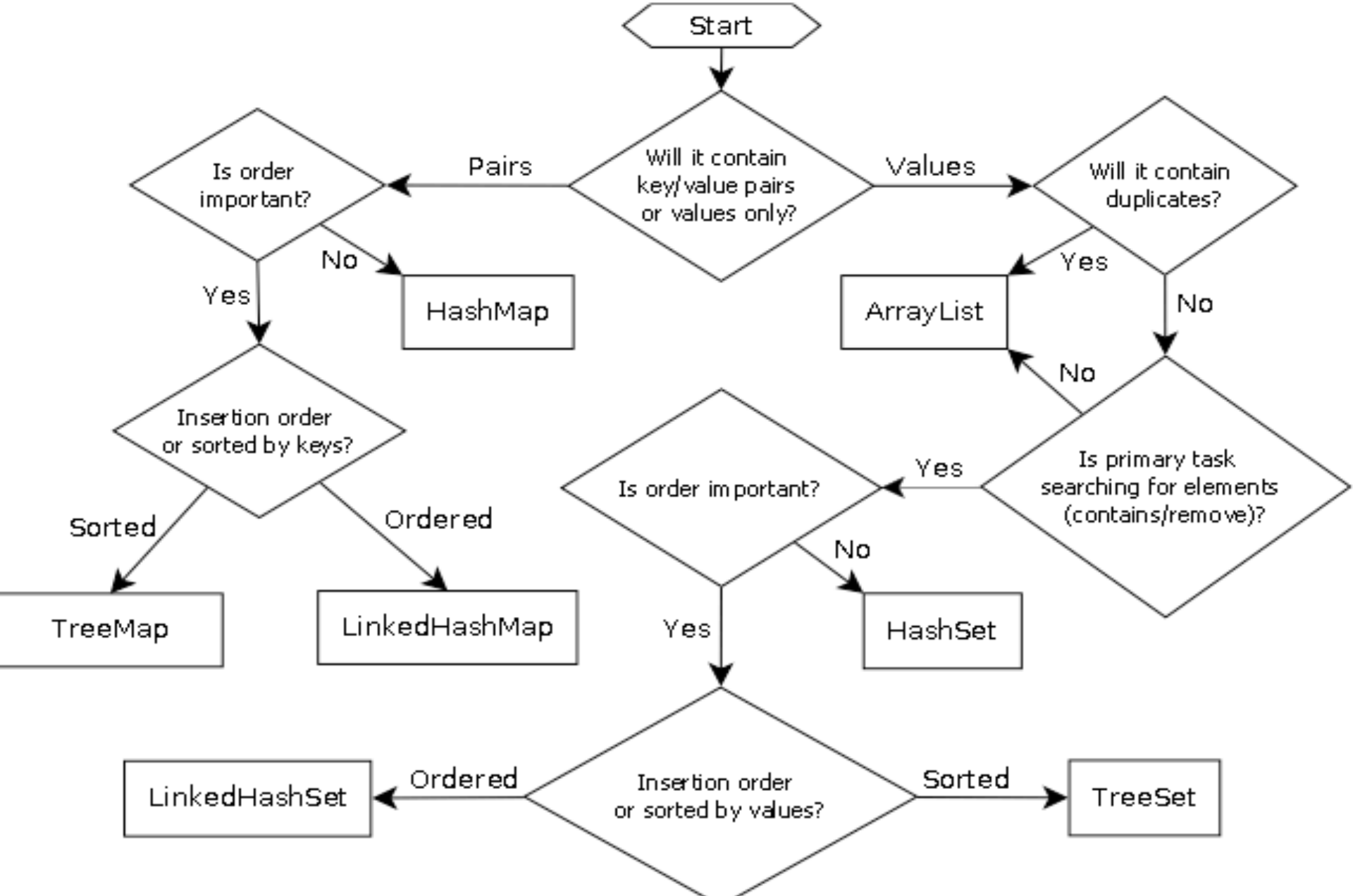
`E remove()` Retrieves and removes the head of this queue

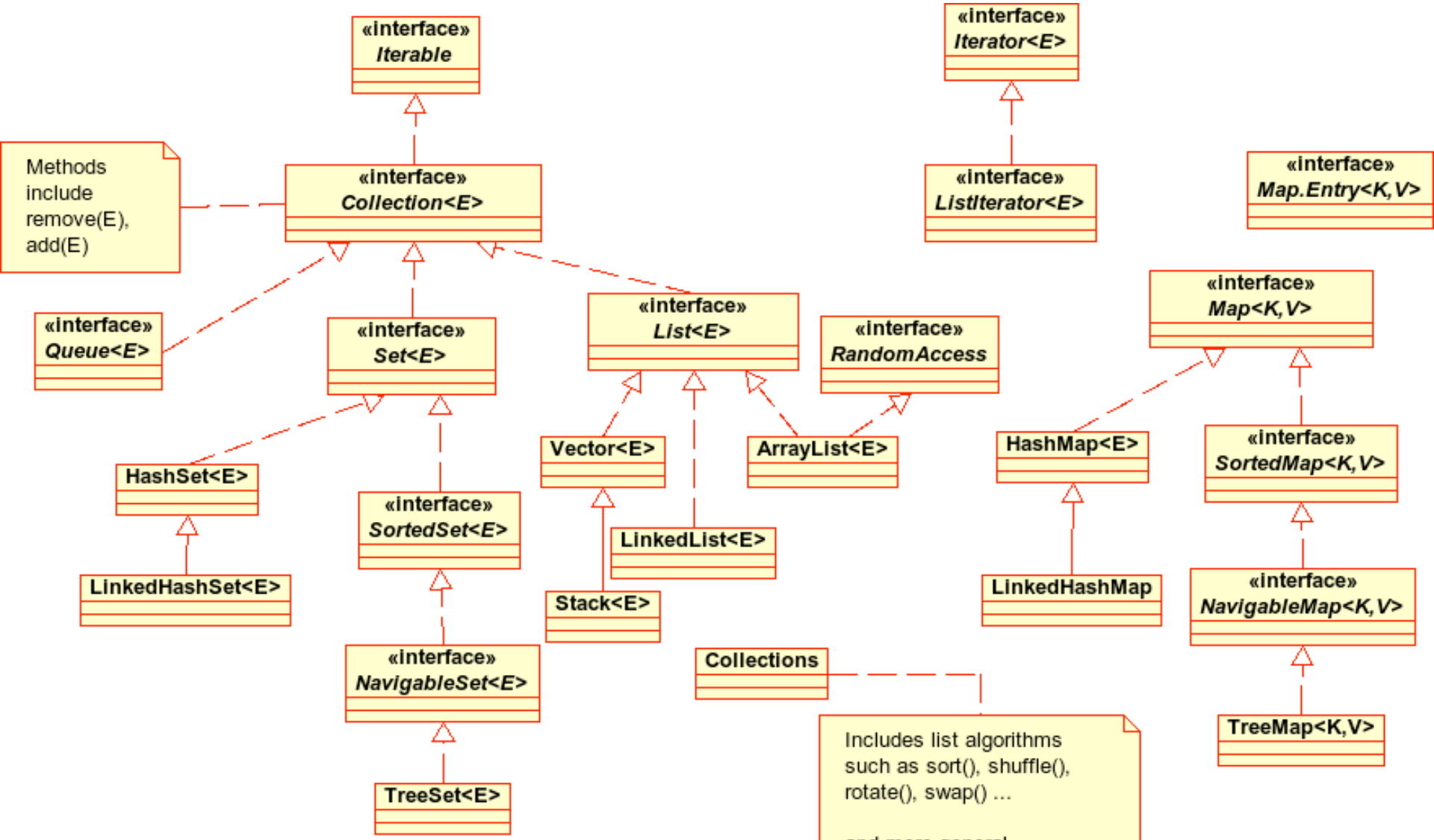
A thread safe FIFO queue

```
ArrayBlockingQueue<Double> numberQ =
        new ArrayBlockingQueue<>(40);

numberQ.add(3.3);
numberQ.add(2.2);
numberQ.add(5.5);
numberQ.add(4.4);
numberQ.add(7.7);

assertEquals(3.3, numberQ.peek(), 0.1);
assertEquals(3.3, numberQ.remove(), 0.1);
assertEquals(2.2, numberQ.remove(), 0.1);
assertEquals(5.5, numberQ.peek(), 0.1);
assertEquals(3, numberQ.size());
```

http://www.sergiy.ca/guide-to-selecting-appropriate-map-collection-in-java/

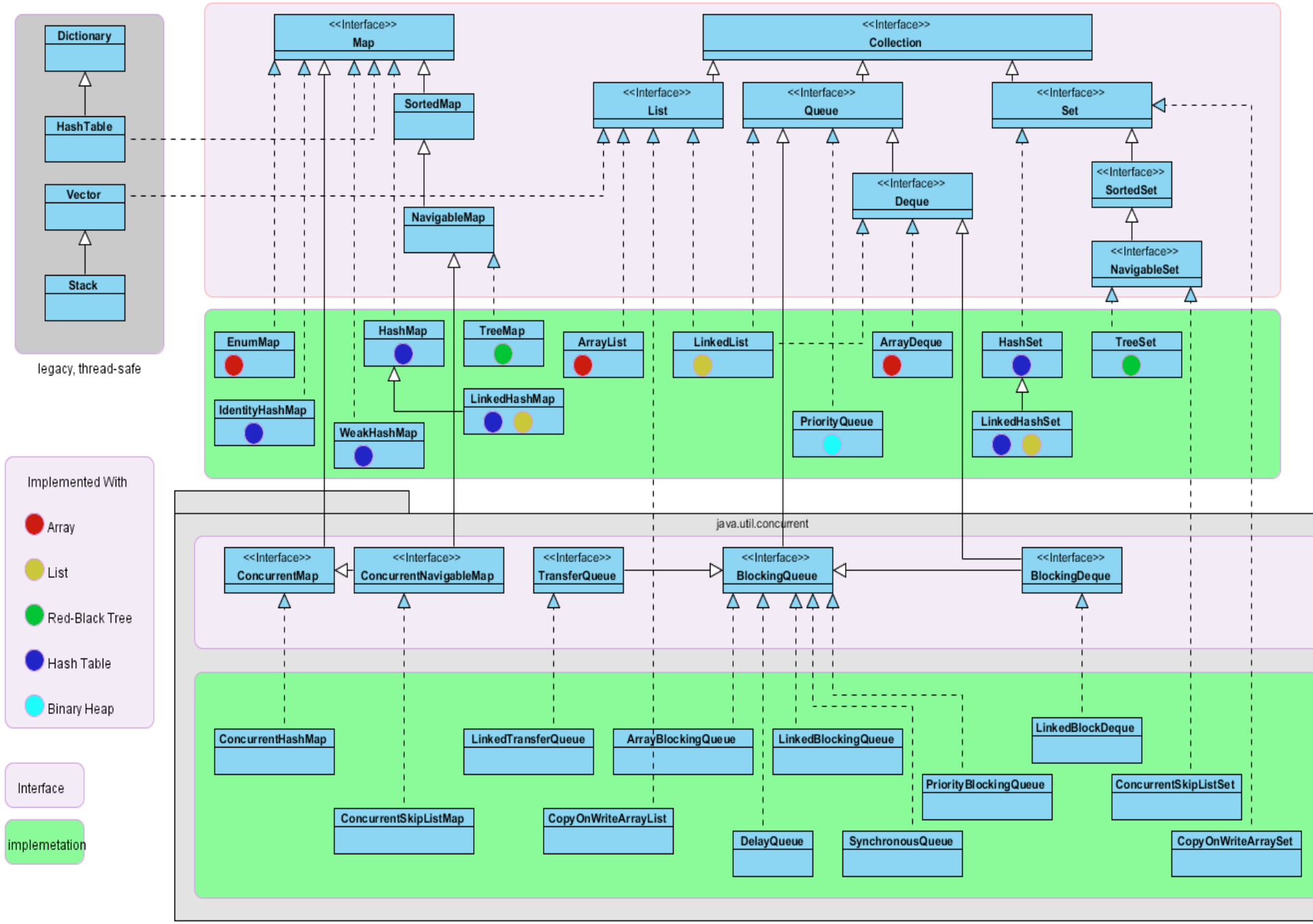
Java Map/Collection Cheat Sheet




«interface»
Iterable
«interface»
Iterator<E>
Methods
include
remove(E),
add(E)
«interface»
Collection<E>
«interface»
ListIterator<E>
«interface»
Map.Entry<K,V>
«interface»
Queue<E>
«interface»
Set<E>
«interface»
List<E>
«interface»
RandomAccess
«interface»
Map<K,V>
HashSet<E>
«interface»
SortedSet<E>
Vector<E>
ArrayList<E>
HashMap<E>
«interface»
SortedMap<K,V>
LinkedHashSet<E>
LinkedList<E>
Stack<E>
LinkedHashMap
«interface»
NavigableMap<K,V>
«interface»
NavigableSet<E>
Collections
TreeMap<K,V>
TreeSet<E>
Includes list algorithms
such as sort(), shuffle(),
rotate(), swap() ...
and more general
algorithms

Java Collection Cheat Sheet

http://pierrchen.blogspot.com



There's more…

- Previous slides have a lot, but not all

http://docs.oracle.com/javase/7/docs/api/java/util/Collection.html

- Should the Java array object be considered?
- And then, what if you want a graphical view of a list *and we will…*
  - Use `interface ListModel<E>`

Interface ListModel

```
public interface ListModel<E> {
  public int getSize();
  public String toString();
  public E getElementAt(int index);
  public void addListDataListener(ListDataListener l);
  public void removeListDataListener(ListDataListener l);
}
```

- The argument type for **`JList`**
  - A graphical view of a list

`setModel(ListModel<String> model)`