Java Collection Framework

Banu Prakash C

Objective

- Understand the Java Collection Framework
- Use the concrete collection implementations
- Use algorithm classes for sorting, searching collections
- Use Thread-Safe and Read-only Collections

Container (Data Structure)

- A Container is a data structure used for storing objects in an organized way following specific access rules.
- The underlying implementation of various types of containers may vary allowing for flexibility in choosing the right implementation for a given scenario.

Container (Data Structure)

Container classes are expected to implement methods to do the following:

- create a new empty container (constructor),
- report the number of objects it stores (size),
- delete all the objects in the container (clear),
- insert new objects into the container,
- remove objects from it,
- provide access to the stored objects.

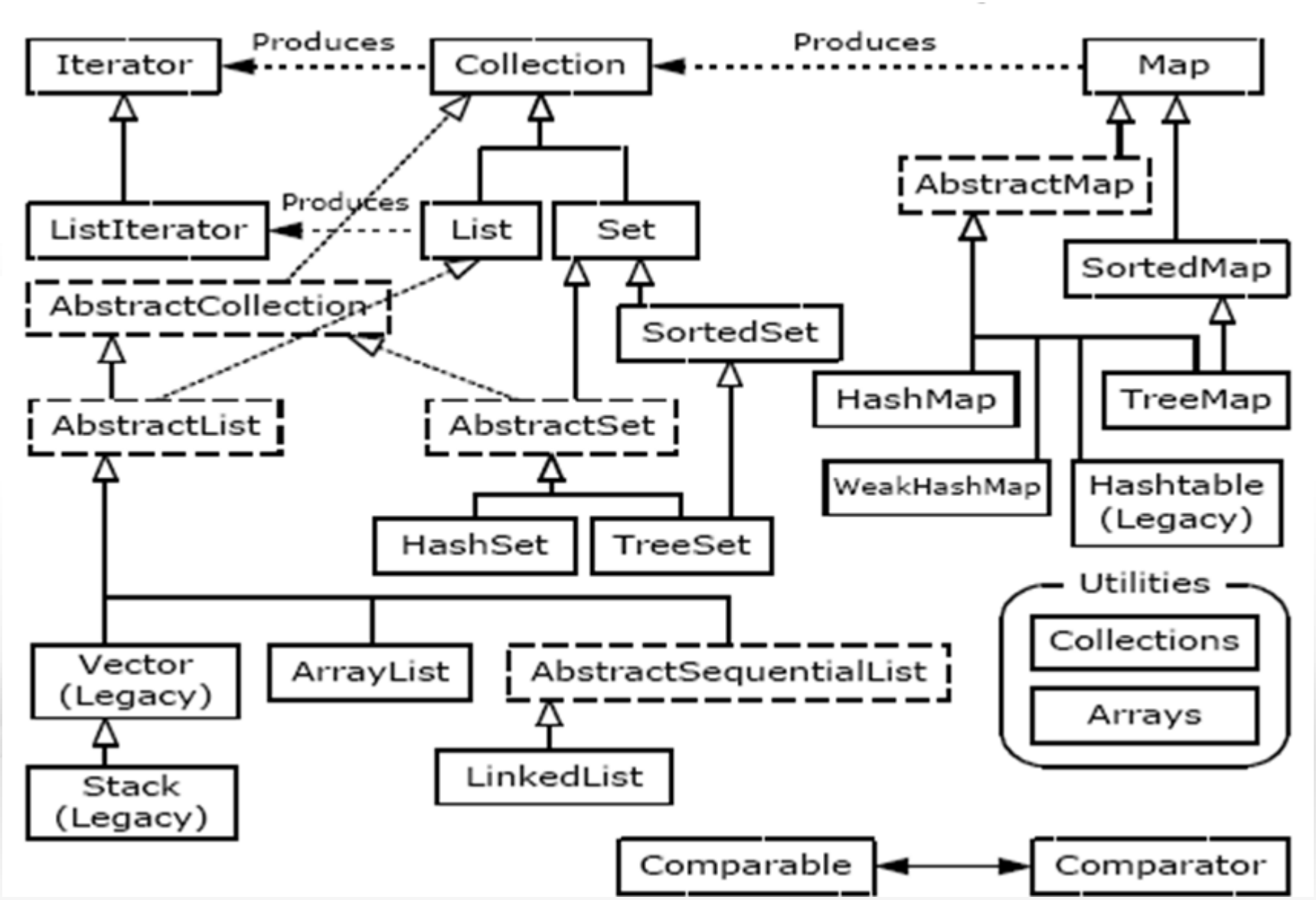
Arrays

- Arrays should always be considered first for holding a group of items because of their efficiency.
- Sometimes arrays are not sophisticated enough for a particular problem.
  - The dimension of an array is determined the moment the array is created, and cannot be changed later on.
  - It cannot grow or shrink.
  - Adding and removing the elements into an array from arbitrary position is very inefficient, for every operation we need to move many elements

Java Collection Framework

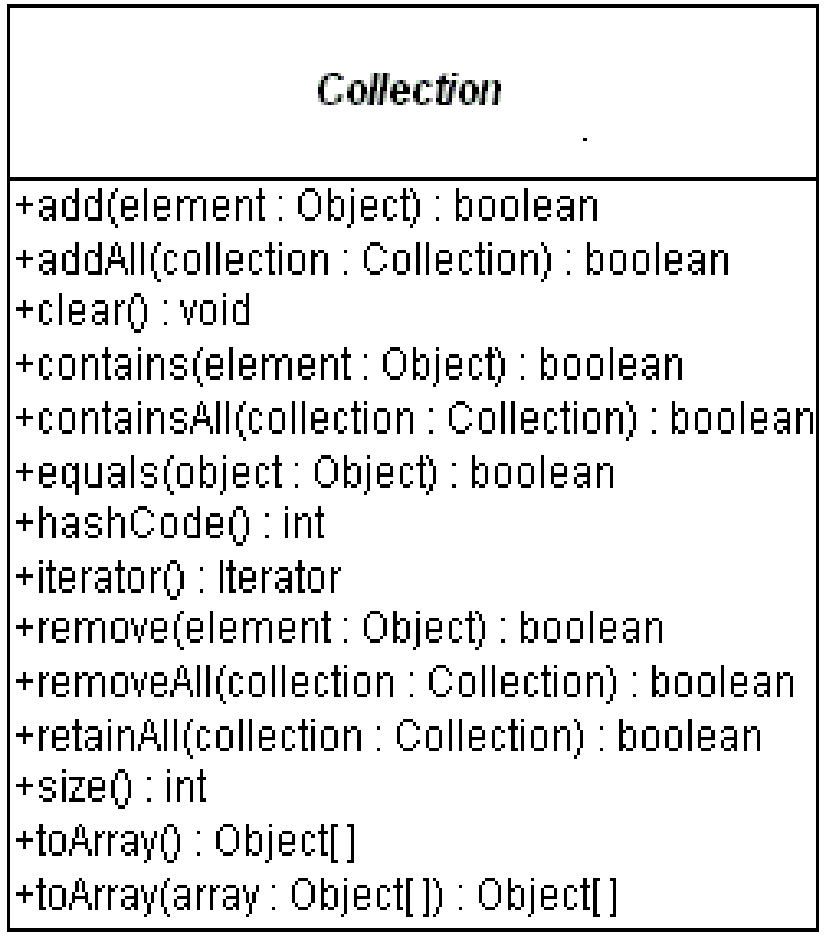
- It is an extensive set of interfaces, implementation classes and algorithm classes.
- Framework is provided in java.util package and comprises of three parts:
  1. Core interfaces
  2. Set of implementations.
  3. Utility methods
- Need for Java Collection Framework
  - Reduces Programming effort
  - Increases performance
  - Provides Interoperability across data structures.

Java Collection Framework



Collection interface

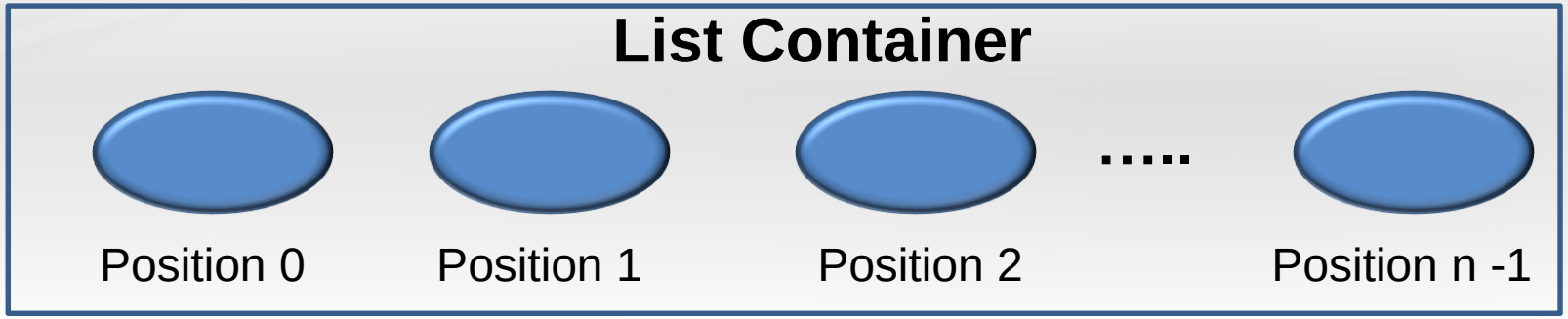
- An Container which supports storing group of objects.
- Can be a Set or List
- Supports basic operations like adding and removing.
- This interface supports query operations.
  - size( ) , iterator( ) and contains( )
- This interface supports group operations.
  - addAll( ), containsAll(), removeAll(), ..



List interface

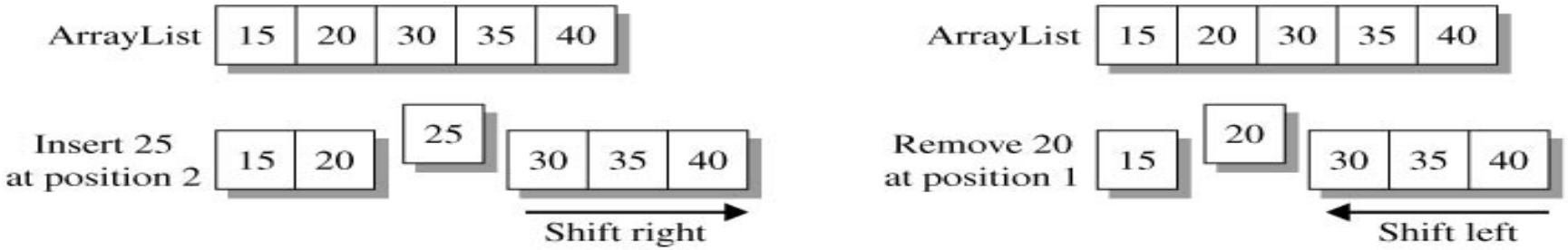
- List interface extends Collection to define a Ordered Collection.
- Permits duplicates and supports position-oriented operations.

- The position-oriented operations include the ability to insert an element or Collection, get an element, as well as remove or change an element.
  - void add(int index, Object element)
  - boolean addAll(int index, Collection collection)
  - Object get(int index)
  - int indexOf(Object element)
  - Object remove(int index)



ArrayList

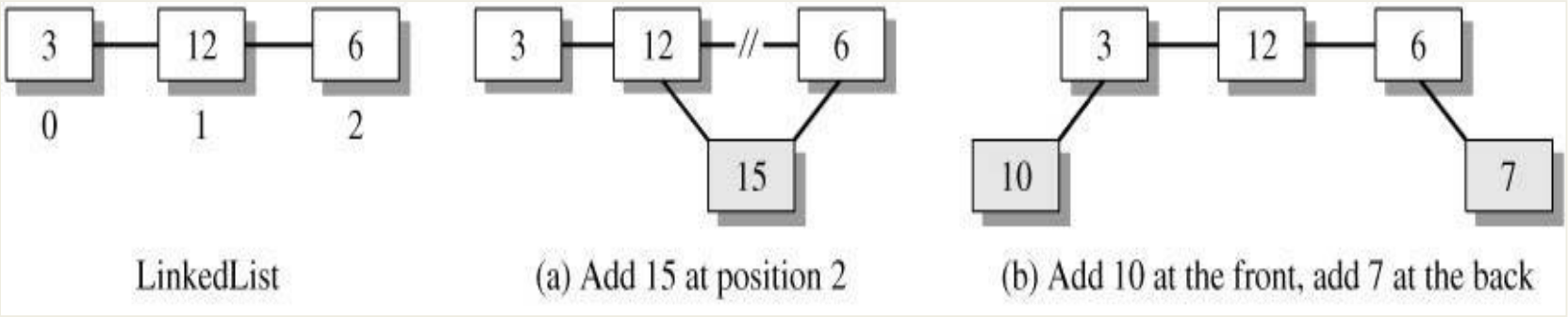
- ArrayList class implements List interface
- Stores element in a contiguous block of memory and automatically expandable.
- The collection efficiency is O(1) if insert and delete element is at the rear of the list.
- Operations at Intermediate positions have O(n) efficiency.



Shifting blocks of elements to insert or remove an ArrayList item.

LinkedList

- LinkedList class implements List interface
- Elements have a value and links that identify adjacent elements in the sequence.
- Inserting or deleting elements are O(1) operations
- LinkedList is not efficient if you need lot of fetching data using their index position.



LinkedList contd..

- Additional methods of LinkedList, By using these new methods, you can easily treat the LinkedList as a stack or queue oriented data structure

```
getFirst( ) : E
getLast( ) : E
addFirst( ) : void
addLast( ) : void
removeFirst( ) : E
removeLast( ) : E
```

ArrayList	**LinkedList**
Random Access	No Random Access
Adding and removing mostly from last.	Adding and removing is required from arbitrary position.

Code Snippet: List usage without generics

```
/*
 * ArrayList, LinkedList and Vector implements List interface.
 * instantiate ArrayList and refer it using List interface.
 * Prior to Java version 1.5. Collections was used without generics.
 */
List list = new ArrayList();
list.add(new String("Java")); // add String
list.add(new Integer(22));  // add Integer
list.add(new Date());       // add Date

/* You need to know what type of object
 * is available at a given index while fetching.
 */

Date d = (Date) list.get(2); //Not Type Safe(can throw ClassCastException)
System.out.println(d);
```

Code Snippet: List traverse without generics

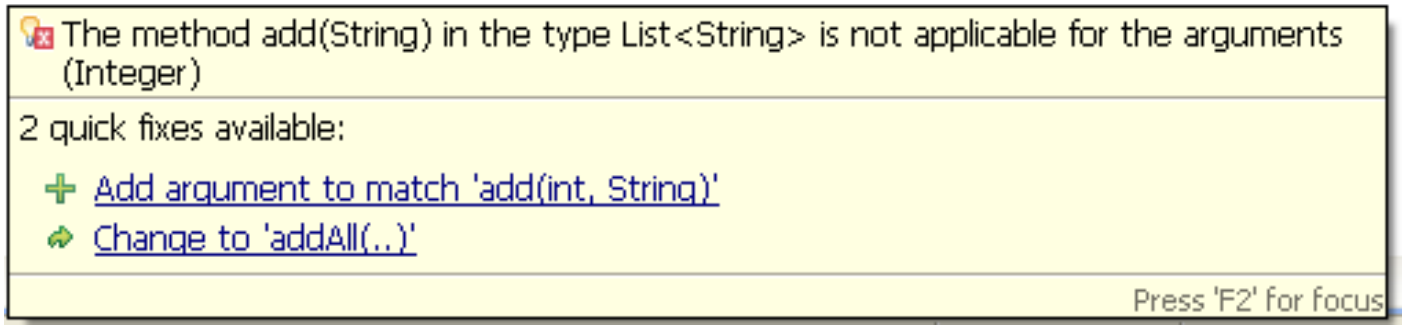
```
/*
 * Traverse through each element using index.
 */

for(int i = 0; i < list.size(); i++) {
    if( list.get(i) instanceof String) {
        String str = (String) list.get(i);
        System.out.println(str);
    } else if( list.get(i) instanceof Date) {
        Date date = (Date) list.get(i);
        System.out.println(date);
    } else if( list.get(i) instanceof Integer) {
        Integer iValue = (Integer) list.get(i);
        System.out.println(iValue);
    }
}
```

Code Snippet: List usage with generics

```
List<String> strList = new ArrayList<String>();
strList.add(new String("Rahul"));
strList.add(new String("Kavitha"));
strList.add(new String("Smitha"));

strList.add(new Integer(22));
```




```
/*
 * traverse through the list using its index.
 */
for (int i = 0; i < strList.size(); i++) {
    String str = strList.get(i);
    // remaining code
}
```

Iterator

- An Iterator provides a useful abstraction for moving through the sequence of objects held in a container.
  - Allows the programmer to obtain objects without knowing about the underlying structure.

Methods

- hasNext( ) : boolean // check if elements exist
- next( ) : Object // return the Element and advance
- remove () : void // remove the currently pointed element.

```
List<String> flavours = new ArrayList<String>();
flavours.add("chocolate");
flavours.add("strawberry");
flavours.add("vanilla");


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

Iterator's are safe

Once Iterator is obtained for a collection it does not allow modification on that collection until the iteration is completed

```
Iterator<String> iter = strList.iterator();
/*
 * TODO
 */
strList.add(new String("Test"));
while(iter.hasNext()){
    System.out.println(iter.next());
}
```

java.util.ConcurrentModificationException

For loop does not fail-fast. It allows modification to collection even while traversing.
The code shown below loops infinitely until it throws StackOverflowException

StackOverflowException

```
for(int i = 0 ; i < strList.size(); i++) {
    strList.add(new String("Test"));
    System.out.println(strList.get(i));
}
```

Example

- Code Example
  - Refer: ListExample.zip
    - Illustrates how to use different implementations of List interface
    - Illustrates different ways of traversing through the list collection

The java.util.Collections algorithm class

- This class contains static methods which operation on collection.
- Some methods used form this class:
  - public static void **sort**(List list)
    - Sorts the specified list into ascending order, according to the *natural ordering* of its elements.
    - All elements in the list must implement the Comparable interface
    - The specified list must be modifiable
  - public static void **sort**(List list, Comparator c)
    - Sorts the specified list according to the order induced by the specified comparator
  - public static int **binarySearch**(List list, Object key)
    - Searches the specified list for the specified object using the binary search algorithm. The list must be sorted into ascending order according to the *natural ordering* of its elements prior to making this call.

The Comparable interface

- Use Comparable interface for natural comparison
  - Every Object will have an natural comparison, for example employee's are by default sorted by their ID's and String's are sorted alphabetically in ascending order.
  - Signature

```
public interface Comparable<T> {
    int compareTo(T object);
}
```

Comparable interface

- For complete source code check Employee.java

```
public class Employee implements Comparable<Employee> {

    private int employeeId;
    private String firstName;
    private double salary;
    //remaining code ...

    /* Natural comparison on Employee is based on their employeeId's
     * @see java.lang.Comparable#compareTo(java.lang.Object)
     */
    @Override
    public int compareTo(Employee employee) {
        return this.getEmployeeId() - employee.getEmployeeId();
    }
```

Example: List and Comparable interface

- Code Example: ListExample2.zip
  - contains Employee.java and ComparableComparatorExample.java
  - Illustrates comparison of user defined entity classes using Comparable interface.
  - Illustrates usage of algorithms present in java.util.Collections class.
- Video :
  - Collections-01.swf
    - Usage of List and Comparable interface

The Comparator interface

- We want different views of employee list, one view sorted by firstName and one by salary,…
- By having Employee implementing Comparable, we get only one chance to implement the compareTo( ) method.
- Solution:
  - Use Comparator interface.
  - A Comparator is external to the element type we're comparing- it's a separate class
  - We can make as many of these as we like.

```
public interface Comparator<T> {
    int compare(T o1, T o2);
}
```

Employee Class revisited

```
public class Employee implements Comparable<Employee> {

    private int employeeId;
    private String firstName;
    private double salary;

    //remaining code ..

    /*
     * static inner class to sort employees by name
     *
     */
    public static class NameComparator implements Comparator<Employee> {
        public int compare(Employee e1, Employee e2) {
            return e1.getFirstName().compareTo(e2.getFirstName());
        }
    }
```

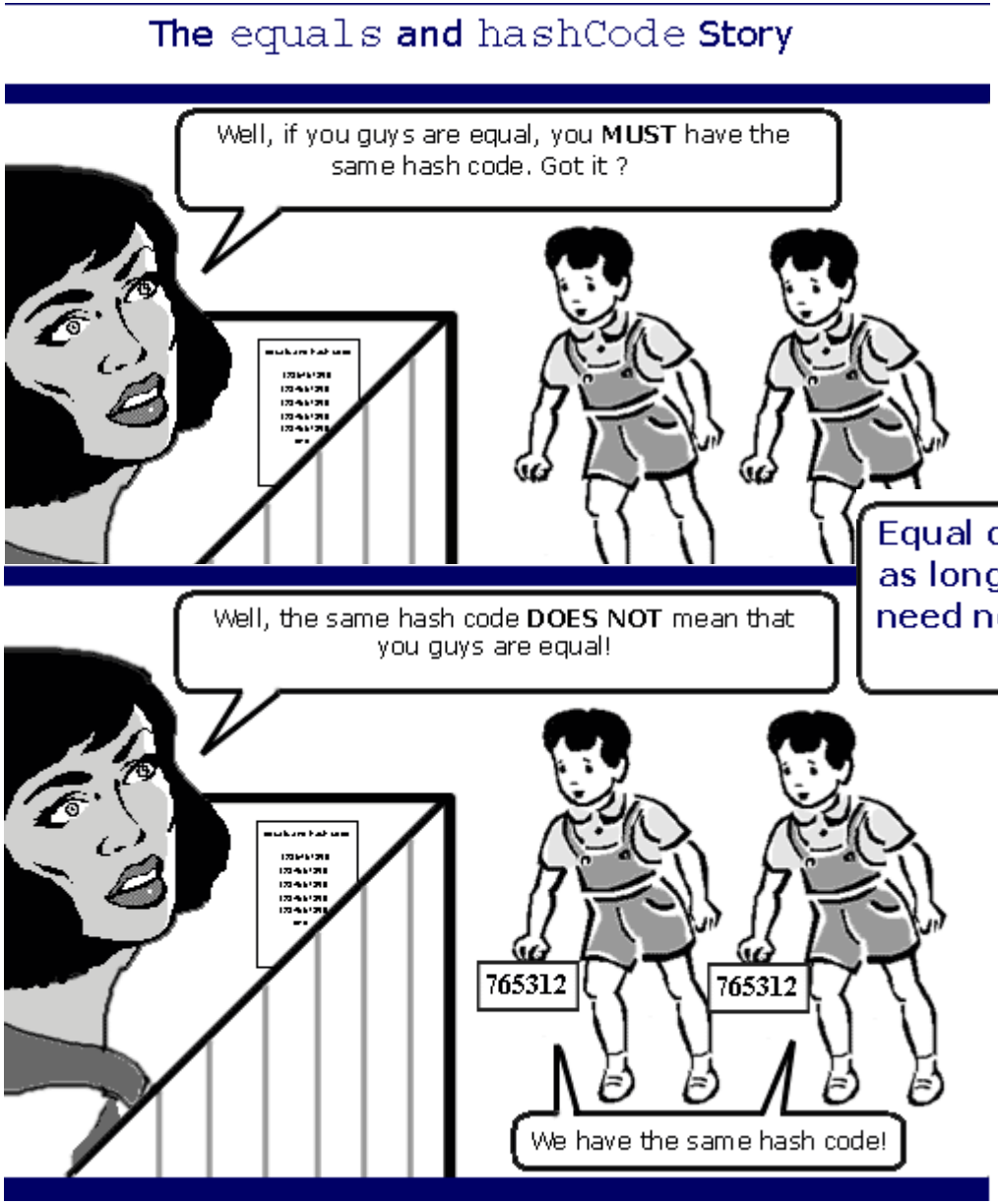
Example: Using Comparable and Comparator.

- Code Example: ListExample3.zip
  - contains Employee.java and ComparableComparatorExample.java
  - Illustrates comparison of user defined entity classes using Comparable and Comparator interface.
  - Illustrates usage of algorithms present in java.util.Collections class.
- Video :
  - Collections-02.swf
    - Usage of List and Comparator interface

Set interface

- A set is just a group of unique items, meaning that the group contains no duplicates.
- The set has no set order, elements can't be found by index.
- Some real-world examples:
  - Set of Java keywords {“abstract”, ”final”, ”volatile”, “import”,…}
  - Set of upper case letters {“A”,”B”,”C”,…}
- The java.util.SortedSet interface extends java.util.Set interface.
  - Unlike a regular set, the elements in a sorted set are sorted using java.util.Comparable or java.util.Comparator interface.

Brief introduction to HashCode.



Equal objects must produce the same hash code as long as they are equal, however unequal objects need not produce distinct hash codes.

HashSet

- The HashSet class implements Set interface.
- More often than not, you will use a HashSet for storing your duplicate-free collection.
- For efficiency, objects added to a HashSet need to implement the hashCode() method in a manner that properly distributes the hash codes.

TreeSet

- SortedSet extends Set interface, which guarantees that the elements in the set are sorted.

- TreeSet is a concrete class that implements the SortedSet interface

- The TreeSet implementation is useful when you need to extract elements from a collection in a sorted manner.
- Use a Tree only if you need the set ordered, else use HashSet.

	HashSet	**TreeSet**
Storage Method	Hash table	Red black Tree
Put Speed	O(1)	O(lg n)
Iteration Order	Arbitrary	Sorted

Example

- Code Example:
  - Refer: SetExample1.zip
    - Contains Employee.java and SetExample.java
    - Illustrates usage of HashSet
  - Refer: SetExample2.zip
    - Contains Employee.java and SetExample.java
    - Illustrates usage of TreeSet
- Video:
  - Collections-03.swf
    - How to use HashSet
  - Collections-04.swf
    - HashSet good practice
  - Collections-05.swf
    - TreeSet with comparable or comparator interface.

Group Operations supported by Collection interface

- boolean containsAll(Collection collection)
  - allows you to discover if the current collection contains all the elements of another collection, a *subset*
- boolean addAll(Collection collection)
  - Ensures all elements from another collection are added to the current collection
- void clear()
  - Removes all elements from the current collection
- void removeAll(Collection collection)
  - Similar to clear, but only removes a subset of elements
- void retainAll(Collection collection)
  - Removes from the current collection those elements not in the other collection, an *intersection*

Map interface

- The Map interface is not an extension of Collection interface.
  - Map storage mechanism varies completely from that of List and Set.
  - Maps store objects by key/value pairs.
- Keys are unique, but values can be duplicated.
  - Examples where Map can be used
    - A dictionary (words mapped to meanings)
    - Windows Registry
    - The map of IP addresses to domain names (DNS)
    - User Login information

Key	Value
Username	James
Password	Secret123#

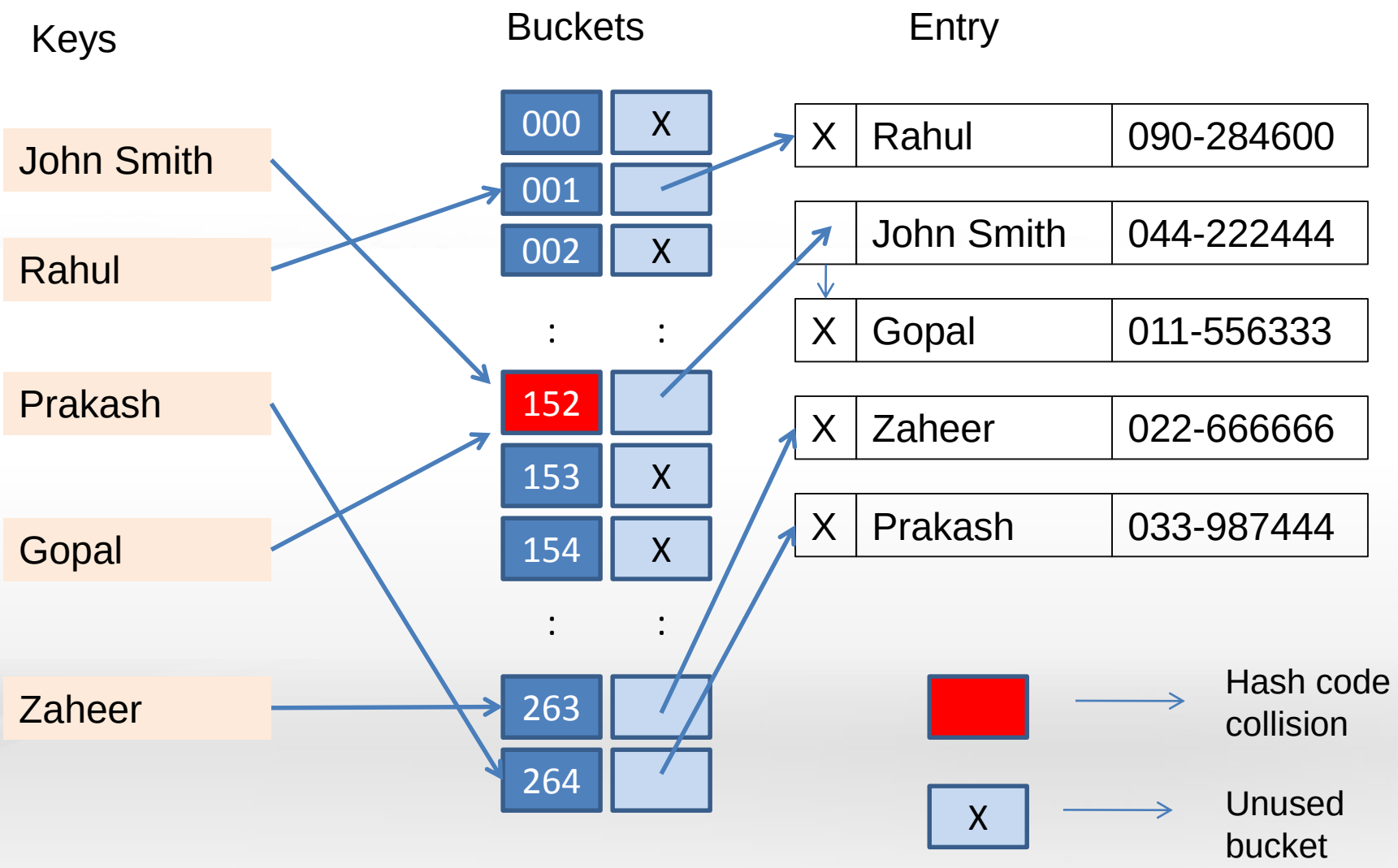
HashMap

- HashMap class implements Map interface.
- Efficiency is O(1) for adding and removing operations.
- Iteration Order is arbitrary.

Some important methods:

Object **put**(*Object*, *Object*)	Associates the specified value with the specified key in this map
Object **get**(*Object*)	Returns the value to which this map maps the specified key.
boolean **containsKey**(*Object*)	Returns true if this map contains a mapping for the specified key.
Set **entrySet**()	Returns a set view of the mappings contained in this map.
Object **remove**(*Object*)	Removes the mapping for this key from this map if present

HashMap


Keys
Buckets
Entry
John Smith
Rahul
Prakash
Gopal
Zaheer
000
001
002
152
153
154
263
264
X
Rahul
090-284600
John Smith
044-222444
Gopal
011-556333
Zaheer
022-666666
Prakash
033-987444
Hash code collision
Unused bucket

TreeMap

- TreeMap implements SortedMap interface
- Storage mechanism is based on Red-Black Tree.
- Efficiency is O(lg n) for adding and removing operations
- TreeMap allows us to retrieve the elements in sorted order defined by the user.
- TreeMap will try to sort the keys in the natural order. Means will consider them as instances of Comparable interface.
- TreeMap allows us to specify an optional Comparator object during its creation. The keys should be implement comparator interface

Map code Snippet

```
/*
 * Create an HashMap and refer it using Map interface.
 * Key for map will be MindTree ID and value will
 * be complete Employee Object.
 */
Map<String, Employee> employeeMap = new HashMap<String, Employee>();
/*
 * put() of Map takes key, value as arguments
 */
employeeMap.put("M100033", new Employee(200,"Karthik",56777.00));
employeeMap.put("M100022", new Employee(345,"Bharath",96777.50));
employeeMap.put("M104522", new Employee(100,"Sujay",39999.00));
employeeMap.put("M100062", new Employee(500,"Zeenat",25000.00));



/*
 * using the Same key will replace the old value with new value
 */
employeeMap.put("M104522", new Employee(320,"Prakash",39999.00));
```

Map Code Snippet

```
/*
 * use Object get(Object key)  to get the value for
 * a key
 */

Employee emp = (Employee) employeeMap.get("M100022");
System.out.println(emp);
```

Map Code Snippet

```
/*
 * Iterate over key set of the Map
 */

Iterator<String> iter = employeeMap.keySet().iterator();

while(iter.hasNext()) {
    String employeeId = iter.next();
    System.out.println(employeeId);
    if(employeeId.equals("M104522")) {
        iter.remove();  // this removes the entire entry
    }
}
```

Map Code Snippet

```
/*
 * Map methods keySet(), values() and entrySet()
 * produces Collection.
 * Traverse through EntrySet of Map.
 * Each Entry has a key and value.
 */

Set<Entry<String,Employee>> entrySet = employeeMap.entrySet();
for(Entry<String, Employee>  entry : entrySet ){
    System.out.println(entry.getKey() + "--->" + entry.getValue());
}
```

Example

- Code Example
  - Refer: MapExample.java
  - Illustrates how to add ,remove and get elements from HashMap
  - Illustrates traversing using KeySet and EntrySet.

Java Legacy Collection Classes

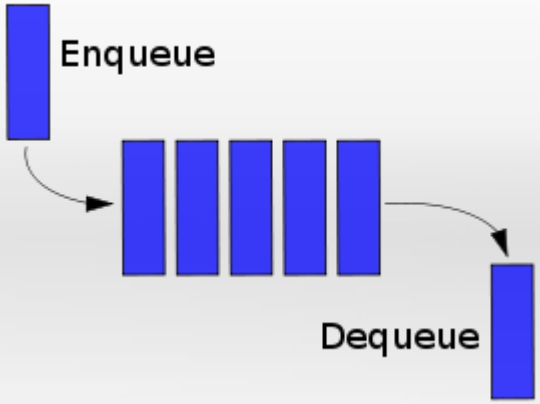
- Vector
  - From Java 2 this class implements List interface, so that it becomes a part of Java Collection Framework.
  - Vector is similar to ArrayList but it is synchronized
  - The Enumerations returned by Vector's elements method are *not* fail-fast but the Iterator returned from Vector are fail-fast.
- Hashtable
  - From Java 2 this class implements Map interface, so that it becomes a part of Java Collection Framework
  - Hashtable is similar to HashMap but it is synchronized
  - In Hashtable any non-null object can be used as a key or as a value, where as HashMap supports null values.

Refer:

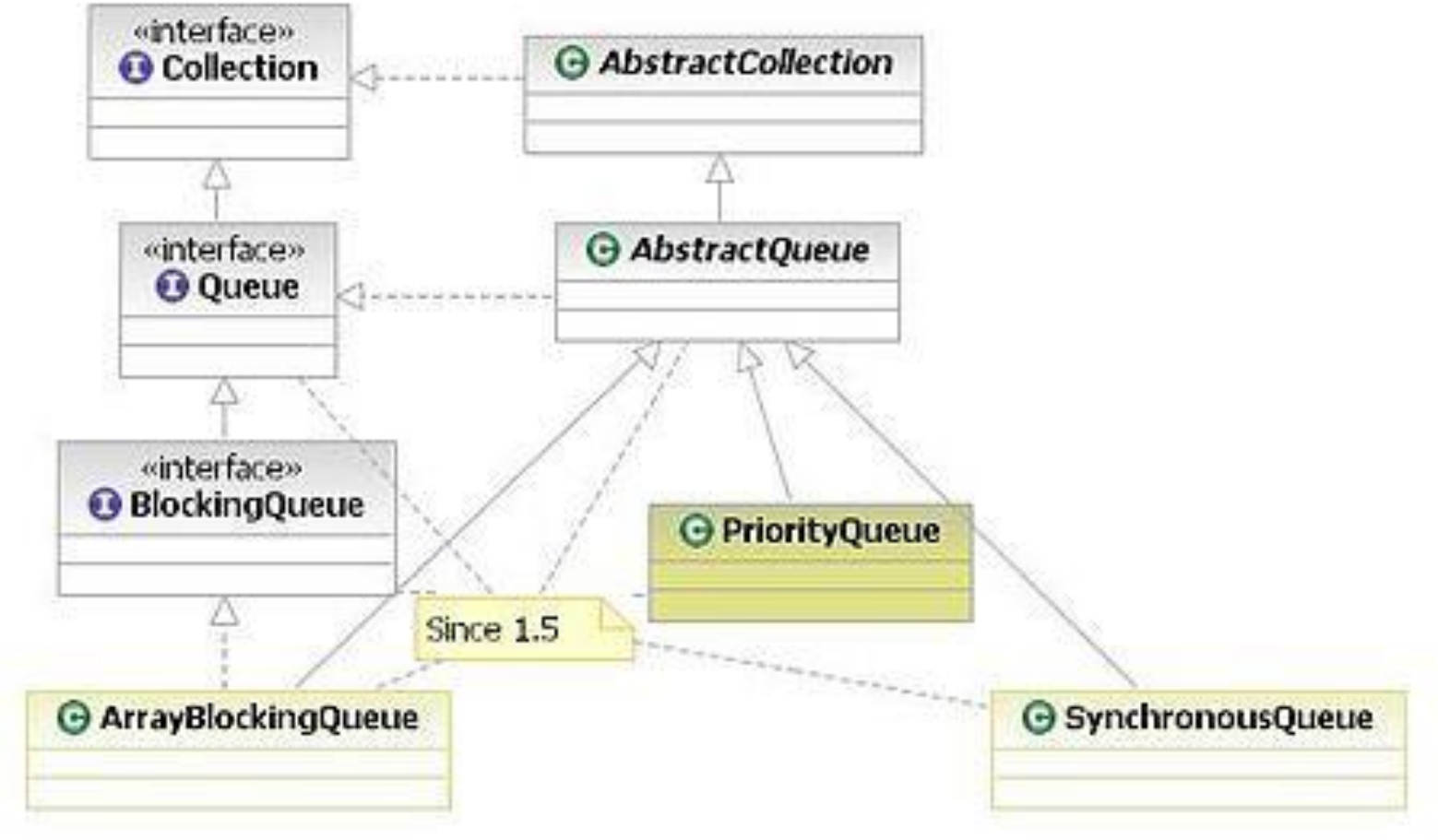
Enumeration : http://goo.gl/8U3FH
Hashtable: http://goo.gl/fchIo

Queue

- Queue is a First-In-First-Out (FIFO) data structure.
- Elements are appended to the end of the queue and are removed from the beginning of the queue
- Items are removed in the same order in which they have been added
- Think of people lining up
  - People join the tail of the queue and wait until they have reached the head of the queue



Queue Hierarchy



PriorityQueue

- The PriorityQueue class implements Queue interface.
- When items are added to a PriorityQueue they are not order by First In, First Out .
- Instead, all items in a PriorityQueue must be comparable (implement the Comparable or Comparator interface) which is used to sort the items in the list.
- An illustrative example of a priority queue is the casualty unit in a large hospital:
  - When a new patient arrives, the severity of its injury or illness is valued; according to this severity, he is inserted into a priority queue.

Code Snippet: PriorityQueue

```
Queue<Patient> patientsQueue = new PriorityQueue<Patient>();

patientsQueue.add(new Patient("Irfan", 66, Severity.MED));
patientsQueue.add(new Patient("Harish", 45, Severity.HIGH));
patientsQueue.add(new Patient("Deepa", 22, Severity.LOW));
patientsQueue.add(new Patient("Preeti", 32, Severity.HIGH));
patientsQueue.add(new Patient("Dinesh", 18, Severity.LOW));
patientsQueue.add(new Patient("Shymala", 75, Severity.HIGH));

for(Patient patient : patientsQueue){
    System.out.println(patient);
}
```

Output: High Priority first, if same priority consider age.

Patient [firstName=Shymala, age=75, severity=HIGH]
Patient [firstName=Preeti, age=32, severity=HIGH]
Patient [firstName=Harish, age=45, severity=HIGH]
Patient [firstName=Irfan, age=66, severity=MED]
Patient [firstName=Dinesh, age=18, severity=LOW]

Refer: Patient.java and PriorityQueueExample.java

Example

- Code Example:
  - Refer: QueueExample.zip
    - Contains Severity.java, Patient.java and PrioriyQueueExample.java
    - Illustrates how PriorityQueue an Queue implementation works.

Queue methods

- Some of the important methods of Queue.
  - boolean **offer**(E o) : inserts an element if possible, otherwise returning false
  - E **poll**() : Retrieves and removes the head of this queue, or null if this queue is empty
  - E **remove**() : This method differs from the poll method in that it throws an exception if this queue is empty.
  - E **peek**() : Retrieves, but does not remove, the head of this queue, returning null if this queue is empty
  - E **element**() : similar to peek, but throws an exception if this queue is empty.

Special collection implementations

- Factory methods for read-only collections
  - Collection unmodifiableCollection(Collection collection)
  - List unmodifiableList(List list)
  - Map unmodifiableMap(Map map)
  - Set unmodifiableSet(Set set)
  - SortedMap unmodifiableSortedMap(SortedMap map)
  - SortedSet unmodifiableSortedSet(SortedSet set)

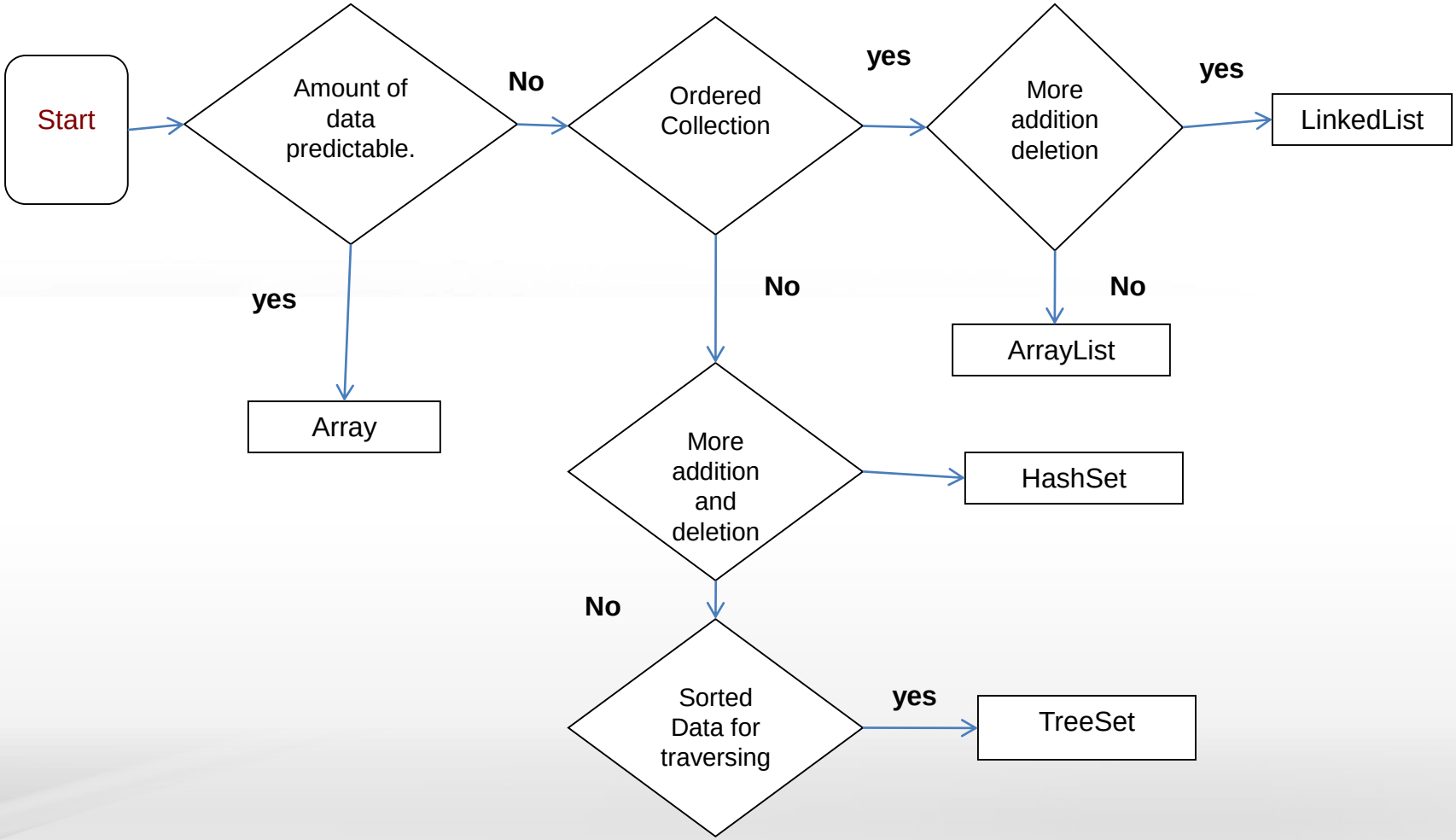
- Example:

```
Set<String> set = new HashSet<String>();
set.add("Pramod");
set.add("Neeraj");
set.add("Banu");

set = Collections.unmodifiableSet(set);
set.add("Jack");
```

java.lang.UnsupportedOperationException

How to choose between Array, Set and List.



Thread-safe collections

- The Collections class provides for the ability to wrap existing collections into synchronized ones.
  - Collection synchronizedCollection(Collection collection)
  - List synchronizedList(List list)
  - Map synchronizedMap(Map map)
  - Set synchronizedSet(Set set)
  - SortedMap synchronizedSortedMap(SortedMap map)
  - SortedSet synchronizedSortedSet(SortedSet set)

- If you are using a collection in a multi-threaded environment, where multiple threads can modify the collection simultaneously, the modifications need to be synchronized

- Usage:

  Set<String> newSet = Collections.*synchronizedSet(set);*

Quiz!

Some quiz questions to reinforce the classroom learning

Quiz

1. ArrayList<Customer> list = new ArrayList<Customer>();

   Which interface should Customer class implement to use the below statement?

   Collections.sort(list);

2. What is the output of

```
Set<String> set = new HashSet<String>();
    set.add("ABC");   set.add("123");   set.add("BCD");

    Collections.sort(set);

for(String s : set) { System.out.print(s + " " ); }
```

Quiz

3. Which collection class is appropriate if lot of add and remove operations from arbitrary position are required and should support sorting?

4. Which method is used to add elements into a Map?

5. Which of the following loops fail-fast (throws ConcurrentModificationException)
   a) For loop using index operation
   b) Enhanced for loop (for each)
   c) Iterator

6. Which class is appropriate for Dictionary storage ?

Explore More!!

Never let your curiosity die!

Explore on these topics

- Apache Commons - http://goo.gl/YntXM
- Explore PECS (Producer Extends Consumer Super) with respect to Generic Collections.
- Explore Red-Black Trees
- Concurrent Collections (Refer JavaDocumentation)
  - java.util.concurrent.ArrayBlockingQueue
  - java.util.concurrent.DelayQueue
- Differentiate between HashMap and WeakHashMap
- Implementing the Iterable Interface
  - Write your own class implementing iterable interface.

References

Contains the reference that will supplement the self learning and will be needed for completing the assignments & practice questions

References

- Java Collection Tutorial from Oracle.com
  - http://goo.gl/t9q02
- **Java Collections**
  - By John Zukowski
- Java Generics and Collections
  - O'Reilly (M Naftalin (Author), P Wadler (Author))
- **Core Java 2, Volume II - Chapter 2: Collections**
- **Code Snippets - http://goo.gl/tEbPM**