Agenda

- List
- Queue
- Set
- Map
- hashcode()
- Date/Calenendr/LocalDate

List Interface

- Ordered/sequential collection.
- Implementations: ArrayList, Vector, Stack, LinkedList, etc.
- List can contain duplicate elements.
- List can contain multiple null elements.
- Elements can be accessed sequentially (bi-directional using Iterator) or randomly (index based).
- List enables searching in the list
- Abstract methods
    - void add(int index, E element)
    - String toString()
    - E get(int index)
    - E set(int index, E element)
    - int indexOf(Object o)
    - int lastIndexOf(Object o)
    - E remove(int index)
    - boolean addAll(int index, Collection<? extends E> c)
    - ListIterator listIterator()
    - ListIterator listIterator(int index)
    - List subList(int fromIndex, int toIndex)
- To store objects of user-defined types in the list, you must override equals() method for the objects.
- It is mandetory while searching operations like contains(), indexOf(), lastIndexOf().

Vector class

- Internally Vector is dynamic array (can grow or shrink dynamically).
- Vector is a legacy collection (since Java 1.0) that is modified to fit List interface.
- Vector is synchronized (thread-safe) and hence slower.
- When Vector capacity is full, it doubles its size.
- Elements can be traversed using Enumeration, Iterator, ListIterator, or using index.
- Primary use
    - Random access
    - Add/remove elements (at the end)
- Limitations
    - Slower add/remove in between the collection
    - Uses more contiguous memory
    - Synchronization slow down performance in single threaded environment
- Inherited from List<>.

Enumeration -- Traversing Vector (Java 1.0)

```
// v is Vector<Integer>
Enumeration<Integer> e = v.elements();
while(e.hasMoreElements()) {
    Integer i = e.nextElement();
    System.out.println(i);
}
```

- Enumeration behaves similar to fail-safe iterator.
- Since Java 1.0
- Methods
  - boolean hasMoreElements()
  - E nextElement()
- only useful when traversing with vector

ArrayList class

- Internally ArrayList is dynamic array (can grow or shrink dynamically).
- When ArrayList capacity is full, it grows by half of its size.
- Elements can be traversed using Iterator, ListIterator, or using index.
- Primary use
  - Random access
  - Add/remove elements (at the end)
- Limitations
  - Slower add/remove in between the collection
  - Uses more contiguous memory
- Inherited from List<>.

LinkedList class

- Internally LinkedList is doubly linked list.
- Elements can be traversed using Iterator, ListIterator, or using index.
- Primary use
  - Add/remove elements (anywhere)
  - Less contiguous memory available
- Limitations:
  - Slower random access
- Inherited from List<>, Deque<>.

Stack

- It is inherited from vector class.
- Generally used to have only the stack operations like push, pop and peek opertaions.
- It is recommended to use the Deque from the queue collection.
- It is synchronized and hence gives low performanance.

Queue Interface

- Represents utility data structures (like Stack, Queue, ...) data structure.
- Implementations: LinkedList, ArrayDeque, PriorityQueue.
- Can be accessed using iterator, but no random access.
- Methods
    - boolean add(E e) - throw IllegalStateException if full.
    - E remove() - throw NoSuchElementException if empty
    - E element() - throw NoSuchElementException if empty
    - boolean offer(E e) - return false if full.
    - E poll() - returns null if empty
    - E peek() - returns null if empty
- In queue, addition and deletion is done from the different ends (rear and front).
- Difference between these methods is first 3 methods throws exception however next 3 methods do not throw exception if operation fails.

Deque interface

- Represents double ended queue data structure i.e. add/delete can be done from both the ends.
- Two sets of methods
    - Throwing exception on failure: addFirst(), addLast(), removeFirst(), removeLast(), getFirst(), getLast().
    - Returning special value on failure: offerFirst(), offerLast(), pollFirst(), pollLast(), peekFirst(), peekLast().
- Can used as Queue as well as Stack.
- Methods
    - boolean offerFirst(E e)
    - E pollFirst()
    - E peekFirst()
    - boolean offerLast(E e)
    - E pollLast()
    - E peekLast()

ArrayDeque class

- Internally ArrayDeque is dynamically growable array.
- Elements are allocated contiguously in memory.
- Time Complexity to add and remove is O(1)

LinkedList class

- Internally LinkedList is doubly linked list.
- Time Complexity to add and remove is O(1)

PriorityQueue class

- Internally PriorityQueue is a "binary heap" (Array implementation of binary Tree) data structure.
- Elements with highest priority is deleted first (NOT FIFO).

- Elements should have natural ordering or need to provide comparator.

Set interface

- Collection of unique elements (NO duplicates allowed).
- Implementations: HashSet, LinkedHashSet, TreeSet.
- Elements can be accessed using an Iterator.
- Abstract methods (same as Collection interface)
    - add() returns false if element is duplicate

HashSet class

- Non-ordered set (elements stored in any order)
- Elements must implement equals() and hashCode()
- Fast execution
- Elements are duplicated in Hashset even if equals() is overriden.
- Its because the hashset dosent compare elements only on the basis of equals().
- Hashset considers elements equal if and only if their hashcode() is same and calling equals() to compare them return true.

LinkedHashSet class

- Ordered set (preserves order of insertion)
- Elements must implement equals() and hashCode()
- Slower than HashSet
- Elements are duplicated in LinkedHashset even if equals() is overriden.
- Its because the LinkedHashset dosent compare elements only on the basis of equals().
- LinkedHashset considers elements equal if and only if their hashcode() is same and calling equals() to compare them return true.

SortedSet interface

- Use natural ordering or Comparator to keep elements in sorted order
- Methods
    - E first()
    - E last()
    - SortedSet headSet(E toElement)
    - SortedSet subSet(E fromElement, E toElement)
    - SortedSet tailSet(E fromElement)

NavigableSet interface

- Sorted set with additional methods for navigation
- Methods
    - E higher(E e)
    - E lower(E e)
    - E pollFirst()
    - E pollLast()

- NavigableSet descendingSet()
- Iterator descendingIterator()

TreeSet class

- Sorted navigable set (stores elements in sorted order)
- Elements must implement Comparable or provide Comparator
- Slower than HashSet and LinkedHashSet
- It is recommended to have consistent implementation for Comparable (Natural ordering) and equals() method i.e. equality and comparison should done on same fields.
- If need to sort on other fields, use Comparator.

```
class Book implememts Comparable<Book> {
    private String isbn;
    private String name;
    // ...
    public int hashCode() {
        return isbn.hashCode();
    }
    public boolean equals(Object obj) {
        if(!(obj instanceof Book))
            return false;
        Book other=(Book)obj;
        if(this.isbn.equals(other.isbn))
            return true;
        return false;
    }
    public int compareTo(Book other) {
        return this.isbn.compareTo(other.isbn);
    }
}
```

```
// Store in sorted order by name
set = new TreeSet<Book>((b1,b2) -> b1.getName().compareTo(b2.getName()));
```

```
// Store in sorted order by isbn (Natural ordering)
set = new TreeSet<Book>();
```

HashTable Data structure

- Hashtable stores data in key-value pairs so that for the given key, value can be searched in fastest possible time.
- Internally hash-table is a table(array), in which each slot(index) has a bucket(collection).
- Load factor = Number of entries / Number of slots.
- Multiple keys can compete for the same slot which can cause the collision

- To avoid the collision two techniques are used

1. Open Adderessing
2. Seperate Chaining

- In Seperate Chaning mechanism to avoid the collision Key-value entries are stored in the same bucket depending on hash code of the "key".
- In java we have readymade/ built-in hashtables

1. HashMap
2. LinkedHashMap
3. TreeMap
4. HashTable (Legacy)
5. Properties (Legacy)

- Here we neeed to calculate the hash value of the key using hash function(Override hashcode method).
- The slot in the table is calculated internaly by slot = key.hashcode()%size
- Examples
    - Key=pincode, Value=city/area
    - Key=Employee, Value=Manager
    - Key=Department, Value=list of Employees

hashCode() method

- Object class has hashCode() method, that returns a unique number for each object (by converting its address into a number).
- To use any hash-based data structure hashCode() and equals() method must be implemented.
- If two distinct objects yield same hashCode(), it is referred as collision. More collisions reduce performance.
- Most common technique is to multiply field values with prime numbers to get uniform distribution and lesser collsions.
- hashCode() overriding rules
    - hash code should be calculated on the fields that decides equality of the object.
    - hashCode() should return same hash code each time unless object state is modified.
    - If two objects are equal (by equals()), then their hash code must be same.
    - If two objects are not equal (by equals()), then their hash code may be same (but reduce performance).

Map interface

- Collection of key-value entries (Duplicate "keys" not allowed).
- Implementations: HashMap, LinkedHashMap, TreeMap, Hashtable, ...
- The data can be accessed as set of keys, collection of values, and/or set of key-value entries.
- Map.Entry<K,V> is nested interface of Map<K,V>.
    - K getKey()
    - V getValue()
    - V setValue(V value)

- Abstract methods

```
* boolean isEmpty()
* int size()
* V put(K key, V value)
* V get(Object key)
* Set<K> keySet()
* Collection<V> values()
* Set<Map.Entry<K,V>> entrySet()
* boolean containsValue(Object value)
* boolean containsKey(Object key)
* V remove(Object key)
* void clear()
* void putAll(Map<? extends K,? extends V> map)
```

- Maps not considered as true collection, because it is not inherited from Collection interface.

HashMap class

- Non-ordered map (entries stored in any order -- as per hash code of key)
- Keys must implement equals() and hashCode()
- Fast execution
- Mostly used Map implementation

LinkedHashMap class

- Ordered map (preserves order of insertion)
- Keys must implement equals() and hashCode()
- Slower than HashSet
- Since Java 1.4

TreeMap class

- Sorted navigable map (stores entries in sorted order of key)
- Keys must implement Comparable or provide Comparator
- Slower than HashMap and LinkedHashMap
- Internally based on Red-Black tree.
- Doesn't allow null key (allows null value though).

Hashtable class

- Similar to HashMap class.
- Legacy collection class (since Java 1.0), modified for collection framework (Map interface).
- Synchronized collection -- Thread safe but slower performance
- Inherited from java.util.Dictionary abstract class (it is Obsolete).

Similarity between Set and Map

- Set in internally using map implementation where it have all the values as null.

- In set the the elements are stored as keys and the corresponsing values are null.
- HashSet = HashMap<K,null>
- LinkedHashSet = LinkedHashMap<K,null>
- TreeSet = TreeMap<K,null>
- in set duplicate elements are not allowed, in map duplicate keys are not allowed
- For HashSet,Hashmap, LinkedHashSet, LinkedHashMap duplication is based on equals() and hashcode() of key
- For TreeSet and TreeMap the duplication is based on comparable of K or Comparator of K given in constructor

Date/ LocalDate/ Calender

- Date and Calender class are in java.util package
- The class Date represents a specific instant in time, with millisecond precision.
- the formatting and parsing of date strings were not standardized it is not recommended to use
- As of JDK 1.1, the Calendar class should be used to convert between dates and time fields and the DateFormat class should be used to format and parse date strings.
- LocalDate is in java.time Package
- It is immutable and threadsafe class.

Java DateTime APIs

- DateTime APIs till Java 7

```
// java.util.Date
Date d = new Date();
System.out.println("Timestamp: " + d.getTime());
// number of milliseconds since 1-1-1970 00:00.
SimpleDateFormat sdf = new SimpleDateFormat("dd-MM-yyyy");
System.out.println("Date: " + sdf.format(d));

// java.util.Date
String str = "28-09-1983";
SimpleDateFormat sdf = new SimpleDateFormat("dd-MM-yyyy");
Date d = sdf.parse(str);
System.out.println(d.toString());

// java.util.Calendar
Calendar c = Calendar.getInstance();
System.out.println(c.toString());
System.out.println("Current Year: " + calendar.get(Calendar.YEAR));
System.out.println("Current Month: " + calendar.get(Calendar.MONTH));
System.out.println("Current Date: " + calendar.get(Calendar.DATE));
```

- Limitations of existing DateTime APIs
    - Thread safety
    - API design and ease of understanding
    - ZonedDate and Time

- Most commonly used java 8 onwards new classes are LocalDate, LocalTime and LocalDateTime.
- LocalDate

```
LocalDate localDate = LocalDate.now();
LocalDate tomorrow = localDate.plusDays(1);
DayOfWeek day = tomorrow.getDayOfWeek();
int date = tomorrow.getDayOfMonth();
System.out.println("Date: " +
tomorrow.format(DateTimeFormatter.ofPattern("dd-MMM-yyyy")));

//LocalDate date = LocalDate.of(1983, 09, 28);
LocalDate date = LocalDate.parse("1983-09-28");
System.out.println("Is Leap Year: " + date.isLeapYear());
```

- LocalTime

```
LocalTime now = LocalTime.now();
LocalTime nextHour = now.plus(1, ChronoUnit.HOURS);
System.out.println("Hour: " + nextHour.getHour());
System.out.println("Time: " +
nextHour.format(DateTimeFormatter.ofPattern("HH:mm")));
```

- LocalDateTime

```
LocalDateTime now = LocalDateTime.now();
LocalDateTime dt = LocalDateTime.parse("2000-01-30T06:30:00");
dt.minusHours(2);
System.out.println(dt.toString());
```

Clone method

- The clone() method is used to create a copy of an object in Java. - It's defined in the java.lang.Object class and is inherited by all classes in Java.
- It returns a shallow copy of the object on which it's called.

```
protected Object clone() throws CloneNotSupportedException
```

- This means that it creates a new object with the same field values as the original object, but the fields themselves are not cloned.
- If the fields are reference types, the new object will refer to the same objects as the original object.
- In order to use the clone() method, the class of the object being cloned must implement the Cloneable interface.
- This interface acts as a marker interface, indicating to the JVM that the class supports cloning.

- It's recommended to override the clone() method in the class being cloned to provide proper cloning behavior.
- The overridden method should call super.clone() to create the initial shallow copy, and then perform any necessary deep copying if required.
- The clone() method throws a CloneNotSupportedException if the class being cloned does not implement Cloneable, or if it's overridden to throw the exception explicitly.