Core Java

Agenda

- Q & A
- Java collection framework
  - Queue interface and classes
  - Set interface and classes
  - Map interface and classes
- Stream Programming

Q & A

Method references

- Lambda expression with only a method call in it (no additional logic) -- can be replaced with a method references.
- static method of class
  - Method signature (args+return value) must be same as SAM.
  - `Comparator<T>` -- int compare(T obj1, T obj2);
  - Arrays.sort(arr, comparator);
  - list.sort(comparator);

```
List<Double> list = new ArrayList<>();
Collections.addAll(list, 4.4, 6.6, 1.1, 3.3, 2.2); // internally calls list.add() for each value given
list.sort(new Comparator<Double>() {
    public int compare(Double d1, Double d2) {
        return Double.compare(d1, d2);
    }
});
// OR
list.sort((d1,d2) -> Double.compare(d1, d2));
```

```
// OR
list.sort(Double::compare);
```

- non-static method of class
  - Method signature (args+return value) should be compatible with SAM.
  - The method will be invoked on first argument of SAM.
  - `Comparator<T>` -- int compare(T obj1, T obj2);

```
List<Double> list = new ArrayList<>();
Collections.addAll(list, 4.4, 6.6, 1.1, 3.3, 2.2);
list.sort((d1,d2) -> d1.compareTo(d2));
// OR
list.sort(Double::compareTo);
```

- non-static method on object
  - list.forEach(consumer);
  - `Consumer<T>` -- void accept(T obj);

```
List<Double> list = new ArrayList<>();
Collections.addAll(list, 4.4, 6.6, 1.1, 3.3, 2.2);
list.forEach(new Consumer<Double>() {
    public void accept(Double ele) {
        System.out.println(ele);
    }
});
// OR
list.forEach(ele -> System.out.println(ele));
// OR
list.forEach(System.out::println);
```

- constructor

List -- removeIf()

- list.removeIf(predicate);
- `Predicate<T>` -- boolean test(T obj);

```
List<Double> list = new ArrayList<>();
Collections.addAll(list, 4.4, 6.6, 1.1, 3.3, 2.2);
list.removeIf(new Predicate<Double>() {
    public boolean test(Double ele) {
        return ele > 5.0;
    }
});
// OR
list.removeIf(ele -> ele > 5.0);
```

CopyOnWriteArrayList

- Whenever collection is modified, a new copy is created and changes done in that copy.

List -- indexOf()

- Users List demo
- indexOf() internally calls equals() on each element with given key as argument. If equal, then return index. If element not found, return -1.
- equals() should be overridden for the class, whose objects to be compared.

Collections.max()

- Collections.max(list);
- Collections.max(list, comparator);

```
List<String> list = new ArrayList<>();
Collections.addAll(list, "A", "B", "CDX", "E", "CF");
String max1 = Collections.max(list);
    // Internally use Natural ordering of String to compare
    // returns -- "E"
String max2 = Collections.max(list, (x,y) -> x.length() - y.length());
    // Internally use given comparator
    // returns -- "CDX"
```

Collections class

- Helper/utility class that provides several static helper methods
- Methods

```
List<E> reverse(List<E> list);
List<E> shuffle(List<E> list);
void sort(List<E> list, Comparator<E> cmp)
E max(Collection<E> list, Comparator<E> cmp);
E min(Collection<E> list, Comparator<E> cmp);
List<E> synchronizedList(List<E> list);
```

```
List<Integer> list = new ArrayList<>();
// list is non-synchronized i.e. not thread-safe
list = Collections.synchronizedList(list);
// list is synchronized i.e. thread-safe
```

LinkedList class

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

Stack class

- Legacy collection class, inherited from Vector class.
- Methods
    - boolean empty()
    - E peek()
    - E pop()
    - E push(E item)
    - int search(Object o)
- Synchronized collection -- Thread safe but slower performance
- Use ArrayDeque<> for better performance.
- In stack, addition and deletion is done from the same end (top).

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).

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.

LinkedList class

- Internally LinkedList is doubly linked list.

PriorityQueue class

- Internally PriorityQueue is a "binary heap" 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

LinkedHashSet class

- Ordered set (preserves order of insertion)
- Elements must implement equals() and hashCode()
- Slower than HashSet

SortedSet interface

- Use natural ordering or Comparator to keep elemenrs 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). Key-value entries are stored in the buckets depending on hash code of the "key".
- Load factor = Number of entries / Number of buckets.
- 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).

Assignment

1. Store few books (hardcoded values with yesterday's Book class) in a HashSet and display them using forEach() method. If any book with duplicate isbn is added, what will happen? Books are stored in which order?
2. In above assignment use LinkedHashSet instead of HashSet. If any book with duplicate isbn is added, what will happen? Books are stored in which order?

3. In above assignment use TreeSet instead of LinkedHashSet. Use natural ordering for the Book. If any book with duplicate isbn is added, what will happen? Books are stored in which order?
4. Use TreeSet to store all books in descending order of price. Natural ordering for the Book should be isbn (do not change it). Display them using forEach().
5. In which collection classes null is not allowed? Duplicate null is not allowed? Multiple nulls are allowed?

```
//Collection<String> c = new ArrayList<>();
//Collection<String> c = new LinkedList<>();
//Collection<String> c = new HashSet<>();
//Collection<String> c = new TreeSet<>();
c.add("A");
c.add("B");
c.add("C");
c.add(null);
c.add(null);
c.add(null);
c.forEach(System.out::println);
```

6. Store Books in HashMap<> so that for given isbn, book can be searched in fastest possible time. Do we need to write equals() and hashCode() in Book class? Hint:

```
// declare map
Map<String,Book> map = new HashMap<>();

// case 1: insert in map
Book b = new Book();
// accept book from user
map.put(b.getIsbn(), b);

// case 2: find in map
String isbn = sc.next();
Book f = map.get(isbn);
```

7. Store Stduents in HashMap<> so that, for given roll, Student can be searched in fastest possible time. Do we need to write equals() and hashCode() in Student class? Follow menu-driven approach. Hint:

```
class Student {
    // ...
}
```

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

```
s = new Student();
acceptStudent(s); // implement method in Main class
map.put(s.getRoll(), s);
```

```
roll = sc.nextInt();
s = map.get(roll);
```

```
map.forEach((k,v)->System.out.println(v));
```