

Agenda

- Queue
- Set
- ~~Map~~
- ~~hashCode()~~

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<>.

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 operations.
- It is recommended to use the Deque from the queue collection.
- It is synchronized and hence gives low performance.

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 be 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 overridden.
- 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 overridden.
- 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 be done on same fields.
- If need to sort on other fields, use Comparator.

```
class Book implements 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>();
```

Optional Assignment

1. Store few books in a HashSet and display them using iterator. If any book with duplicate bookid 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 bookid 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 bookid is added, what will happen? Books are stored in which order?
4. In above assignment use TreeSet parameterized ctor that takes comparator. Use price of the Book for ordering in descending order. If any book with duplicate bookid is added, what will happen? Books are stored in which order?