Agenda

- Set
- Map
- hashcode()
- Java 8 Interfaces
- Functional Interfaces
- Annoymous Inner Classes
- ~~Lambda Expressions~~

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 buckets.
- 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

Java 8 Interface

- Before Java 8 Interfaces are used to design specification/standards. It contains only declarations – public abstract.

```
interface Geometry {
    /*public static final*/ double PI = 3.14;
    /*public abstract*/ int calcRectArea(int length, int breadth);
    /*public abstract*/ int calcRectPeri(int length, int breadth);
}
```

- As interfaces doesn't contain method implementations, multiple interface inheritance is supported (no ambiguity error).
- Interfaces are immutable. One should not modify interface once published.
- Java 8 added many new features in interfaces in order to support functional programming in Java. Many of these features also contradicts earlier Java/OOP concepts.
- From Java 8 onwards we can now also add the method body in the interface, however the methods should be either default or static

1. Default methods

- Java 8 allows default methods in interfaces. If method is not overridden, its default implementation in interface is considered.
- This allows adding new functionalities into existing interfaces without breaking old implementations e.g. Collection, Comparator, ...

```
interface Emp {
    double getSal();
    default double calcIncentives() {
            return 0.0;
        }
    }
    class Manager implements Emp {
        // ...
        // calcIncentives() is overridden
        double calcIncentives() {
            return getSal() * 0.2;
        }
    }
    class Clerk implements Emp {
        // ...
        // calcIncentives() is not overridden -- so method of interface is
considered
    }
```

```
    new Manager().calcIncentives(); // return sal * 0.2
    new Clerk().calcIncentives(); // return 0.0
```

- However default methods will lead to ambiguity errors as well, if same default method is available from multiple interfaces. Error: Duplicate method while declaring class.
- Superclass same method get higher priority. But super-interfaces same method will lead to error.
- Super-class wins! Super-interfaces clash!!

```
    interface Displayable {
        default void show() {
            System.out.println("Displayable.show() called");
        }
    }
    interface Printable {
        default void show() {
            System.out.println("Printable.show() called");
        }
    }
    class FirstClass implements Displayable, Printable { // compiler error:
duplicate method
        // ...
    }
    class Main {
        public static void main(String[] args) {
            FirstClass obj = new FirstClass();
            obj.show();
        }
    }
```

```
    interface Displayable {
        default void show() {
            System.out.println("Displayable.show() called");
        }
    }
    interface Printable {
        default void show() {
            System.out.println("Printable.show() called");
        }
    }
    class Superclass {
        public void show() {
            System.out.println("Superclass.show() called");
        }
    }
    class SecondClass extends Superclass implements Displayable, Printable {
        // ...
    }
    class Main {
        public static void main(String[] args) {
            SecondClass obj = new SecondClass();
            obj.show(); // Superclass.show() called
        }
    }
```

- A class can invoke methods of super interfaces using InterfaceName.super.

```
    interface Displayable {
        default void show() {
            System.out.println("Displayable.show() called");
        }
    }
    interface Printable {
        default void show() {
            System.out.println("Printable.show() called");
        }
    }
```

2. Static methods

- Before Java 8, interfaces allowed public static final fields.
- Java 8 also allows the static methods in interfaces.
- They act as helper methods and thus eliminates need of helper classes like Collections, ...

```
  interface Emp {
  double getSal();
  public static double calcTotalSalary(Emp[] a) {
      double total = 0.0;
      for(int i=0; i<a.length; i++)
                  total += a[i].getSal();
      return total;
```

```
    }
}
```

Functional Interfaces

- If interface contains exactly one abstract method (SAM), it is said to be functional interface.
- It may contain additional default & static methods. E.g. Comparator, Runnable, ...
- @FunctionalInterface annotation does compile time check, whether interface contains single abstract method. If not, raise compile time error.

```
  @FunctionalInterface // okay
  interface Foo {
      void foo(); // SAM
  }
```

```
@FunctionalInterface // okay
interface FooBar1 {
    void foo(); // SAM
    default void bar() {
        /*... */
    }
}
```

```
@FunctionalInterface // NO -- error
interface FooBar2 {
    void foo(); // AM
    void bar(); // AM
}
```

```
@FunctionalInterface // NO -- error
interface FooBar3 {
    default void foo() {
        /*... */
    }
    default void bar() {
        /*... */
    }
}
```

```
@FunctionalInterface    // okay
interface FooBar4 {
    void foo(); // SAM
    public static void bar() {
        /*... */
    }
}
```

- Functional interfaces forms foundation for Java lambda expressions and method references.

Built-in functional interfaces

- New set of functional interfaces given in java.util.function package.
  - `Predicate<T>`: test: T -> boolean
  - `Function<T,R>`: apply: T -> R
  - `BiFunction<T,U,R>`: apply: (T,U) -> R
  - `UnaryOperator<T>`: apply: T -> T
  - `BinaryOperator<T>`: apply: (T,T) -> T
  - `Consumer<T>`: accept: T -> void
  - `Supplier<T>`: get: () -> T
- For efficiency primitive type functional interfaces are also supported e.g. IntPredicate, IntConsumer, IntSupplier, IntToDoubleFunction, ToIntFunction, ToIntBiFunction, IntUnaryOperator, IntBinaryOperator.

Annonymous Inner Class

- Creates a new class inherited from the given class/interface and its object is created.
- If in static context, behaves like static member class. If in non-static context, behaves like non-static member class.
- Along with Outer class members, it can also access (effectively) final local variables of the enclosing method.

```
    // (named) local class
    class EmpnoComparator implements Comparator<Employee> {
        public int compare(Employee e1, Employee e2) {
            return e1.getEmpno() - e2.getEmpno();
        }
    }
    Arrays.sort(arr, new EmpnoComparator());    // anonymous obj of local class
```

```
// Anonymous inner class
Comparator<Employee> cmp = new Comparator<Employee>() {
    public int compare(Employee e1, Employee e2) {
        return e1.getEmpno() - e2.getEmpno();
    }
};
Arrays.sort(arr, cmp);
```

```java
// Anonymous object of Anonymous inner class.
Arrays.sort(arr, new Comparator<Employee>() {
    public int compare(Employee e1, Employee e2) {
        return e1.getEmpno() - e2.getEmpno();
    }
});
```