**Java Collection**

**Collection Hierarchy**



**Map Heirarchy**

Java Map Hierarchy

* **We cannot insert Object as key in TreeSet and TreeMap if it doesn’t implement Comparable interface.**

**In below code Employee overrides equals and hashcode method.**

**Employee2 implements Comparable interface.**

public class Test {

public static void main(final String[] args) {

**mapWithObjectWithEqualsAndHashcode**();// {Employee [name=amit, age=20]=2}

**mapWithObjectWithoutEqualsAndHashcode**();// {Employee [name=amit, age=20]=1, Employee [name=amit, age=20]=2}

**treeMapWithObjectWithoutComparable**();// collection.Employee cannot be cast to java.base/java.lang.Comparable

**treeMapWithObjectWithComparable**();// {Employee [name=amit, age=20]=2}

**treeSetWithObjectWithoutComparable**();// collection.Employee cannot be cast to java.base/java.lang.Comparable

**treeSetWithObjectWithComparable**();// [Employee [name=amit, age=20]]

sortMapByKey();// sumit=1 amit=2

sortMapByValue();// sumit=1 amit=2

}

public static void **mapWithObjectWithEqualsAndHashcode**() {

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

Employee e1 = new Employee("amit", 20);

Employee e2 = new Employee("amit", 20);

map.put(e1, 1);

map.put(e2, 2);

System.out.println(map);

}

public static void **mapWithObjectWithoutEqualsAndHashcode**() {

Map<Employee2, Integer> map = new HashMap<>();

Employee2 e1 = new Employee2("amit", 20);

Employee2 e2 = new Employee2("amit", 20);

map.put(e1, 1);

map.put(e2, 2);

System.out.println(map);

}

private static void **treeSetWithObjectWithoutComparable**() {

try {

Set<Employee> employees = new TreeSet<>();

Employee e1 = new Employee("amit", 20);

Employee e2 = new Employee("amit", 20);

employees.add(e1);

employees.add(e1);

System.out.println(employees);

} catch (Exception e) {

System.out.println(e.getMessage());

}

}

private static void **treeSetWithObjectWithComparable**() {

Set<Employee2> employees = new TreeSet<>();

Employee2 e1 = new Employee2("amit", 20);

Employee2 e2 = new Employee2("amit", 20);

employees.add(e1);

employees.add(e1);

System.out.println(employees);

}

public static void **treeMapWithObjectWithoutComparable**() {

try {

Map<Employee, Integer> map = new TreeMap<>();

Employee e1 = new Employee("amit", 20);

Employee e2 = new Employee("amit", 20);

map.put(e1, 1);

map.put(e2, 2);

System.out.println(map);

} catch (Exception e) {

System.out.println(e.getMessage());

}

}

public static void **treeMapWithObjectWithComparable**() {

Map<Employee2, Integer> map = new TreeMap<>();

Employee2 e1 = new Employee2("amit", 20);

Employee2 e2 = new Employee2("amit", 20);

map.put(e1, 1);

map.put(e2, 2);

System.out.println(map);

}

public static void **sortMapByKey**() {

Map<String, Integer> map = new TreeMap<>();

map.put("amit", 2);

map.put("sumit", 1);

map.entrySet().stream().sorted(Map.Entry.comparingByKey(Comparator.reverseOrder())).forEach(System.out::println);

}

public static void **sortMapByValue**() {

Map<String, Integer> map = new TreeMap<>();

map.put("amit", 2);

map.put("sumit", 1);

map.entrySet().stream().sorted(Map.Entry.comparingByValue()).forEach(System.out::println);

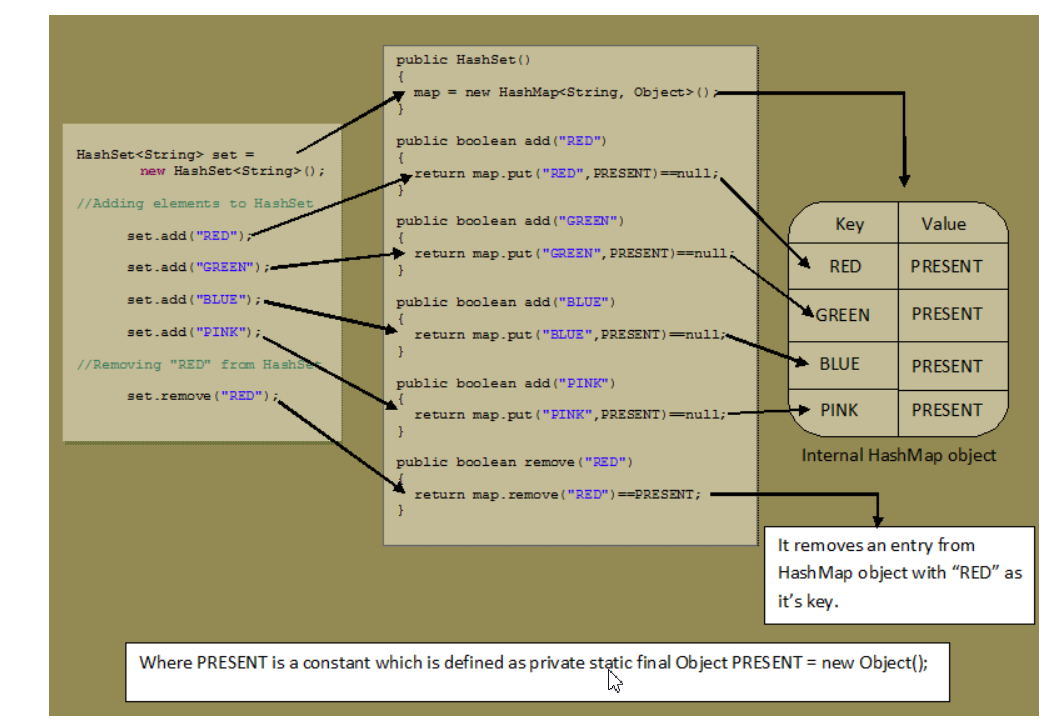
}

}

**Differences between a Stream and a Collection:**

* ***No storage.*** **A stream is not a data structure that stores elements; instead, it conveys elements from a source such as a data structure,** an array, a generator function, or an I/O channel, through a pipeline of computational operations.
* ***Functional in nature.*** **An operation on a stream produces a result, but does not modify its source.** For example, filtering a Stream obtained from a collection produces a new Stream without the filtered elements, rather than removing elements from the source collection.
* ***Laziness-seeking.*** Many stream operations, such as filtering, mapping, or duplicate removal, can be implemented lazily, exposing opportunities for optimization. For example, **"find the first String with three consecutive vowels" need not examine all the input strings.** Stream operations are divided into intermediate (Stream-producing) operations and terminal (value- or side-effect-producing) operations. Intermediate operations are always lazy.
* ***Possibly unbounded.*** **While collections have a finite size, streams need not.** Short-circuiting operations such as limit(n) or findFirst() can allow computations on infinite streams to complete in finite time.
* ***Consumable.*** **The elements of a stream are only visited once during the life of a stream**. Like an Iterator, a new stream must be generated to revisit the same elements of the source.

**How HasSet works internally :**



**Complexity of various data structure.**

