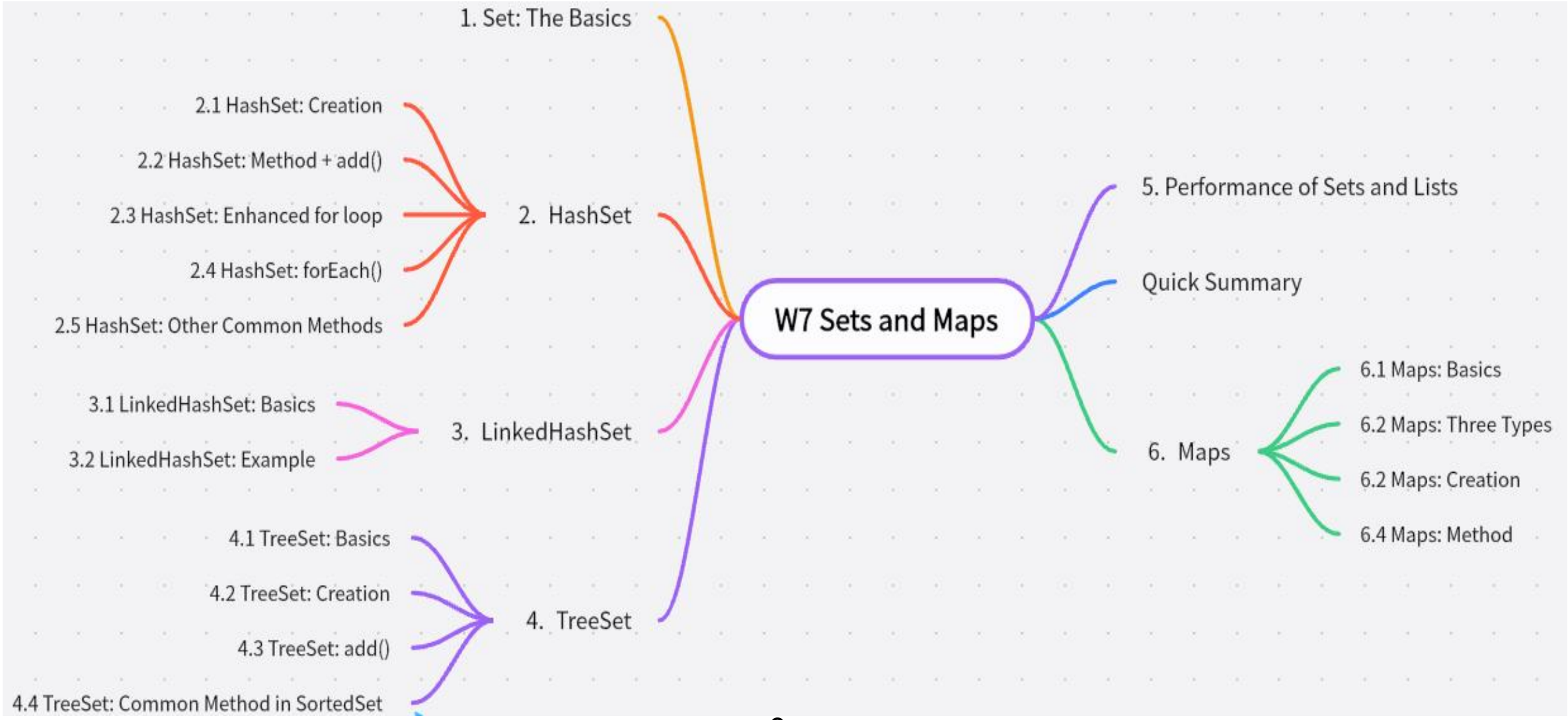
Sets and Maps

CPT 204 - Advanced OO Programming

Content


W7 Sets and Maps
1. Set: The Basics
2. HashSet
2.1 HashSet: Creation
2.2 HashSet: Method + add()
2.3 HashSet: Enhanced for loop
2.4 HashSet: forEach()
2.5 HashSet: Other Common Methods
3. LinkedHashSet
3.1 LinkedHashSet: Basics
3.2 LinkedHashSet: Example
4. TreeSet
4.1 TreeSet: Basics
4.2 TreeSet: Creation
4.3 TreeSet: add()
4.4 TreeSet: Common Method in SortedSet
5. Performance of Sets and Lists
Quick Summary
6. Maps
6.1 Maps: Basics
6.2 Maps: Three Types
6.2 Maps: Creation
6.4 Maps: Method

1. Set: The Basics

- **Set interface**is a sub-interface of **Collection**
- It extends the **Collection,** but does not introduce new methods or constants.
- However, the **Set interface stipulates** that an instance of **Set contains no duplicate elements**
  - That is, **no two elements e1**and **e2**can be in the set such that **e1.equals(e2)**is true

1. Set: The Basics

- You can create a set using one of its three concrete classes: **HashSet**, **LinkedHashSet**, or **TreeSet**
- The concrete classes that implement Set must ensure that **no duplicate elements** can be added to the set
  - HastSet & LinkedHashSet use: **hashcode()** + **equals()**
  - TreeSet use: **compareTo()** or **Comparable**

2.1 HashSet: Creation

- The **HashSet class** is a concrete class that implements **Set**

  1. You can **create an empty hash set using its no-arg constructor**

```
//Create a new, empty HashSet that is designed to store Integer objects.
Set<String> set = new HashSet<>();
```

- The **first diamond operation** ("<>") is called a type parameter or generic type. It specifies the type of elements that the HashSet will store. In this case, the HashSet is specified to store objects of type Integer.
- In the **2nd diamond operation** ("<>"),the compiler infers the generic type from the context, which is typically the same as the type specified in the first diamond operator (Just in simple cases)
- **The parentheses** ("()") is used for calling the constructor of the HashSet class. In this case, it is calling the no-argument constructor of the HashSet class, which creates an empty set.

2.1 HashSet: Creation

- The **HashSet class** is a concrete class that implements **Set**

2. You can create a hash set from an existing collection

```
List<String> list = Arrays.asList("Apple");
// Pass the List to the HashSet constructor
// This will create a HashSet containing the elements from the list
HashSet<String> hashSet = new HashSet<>(list);
```

- We have a List of strings
- <String>, becasue of the list type
- <> ,still same as the pre-specificed type above
- the list (Not List), will be passed to the **parentheses**

2.1 HashSet: Creation

- The **HashSet class** is a concrete class that implements **Set**

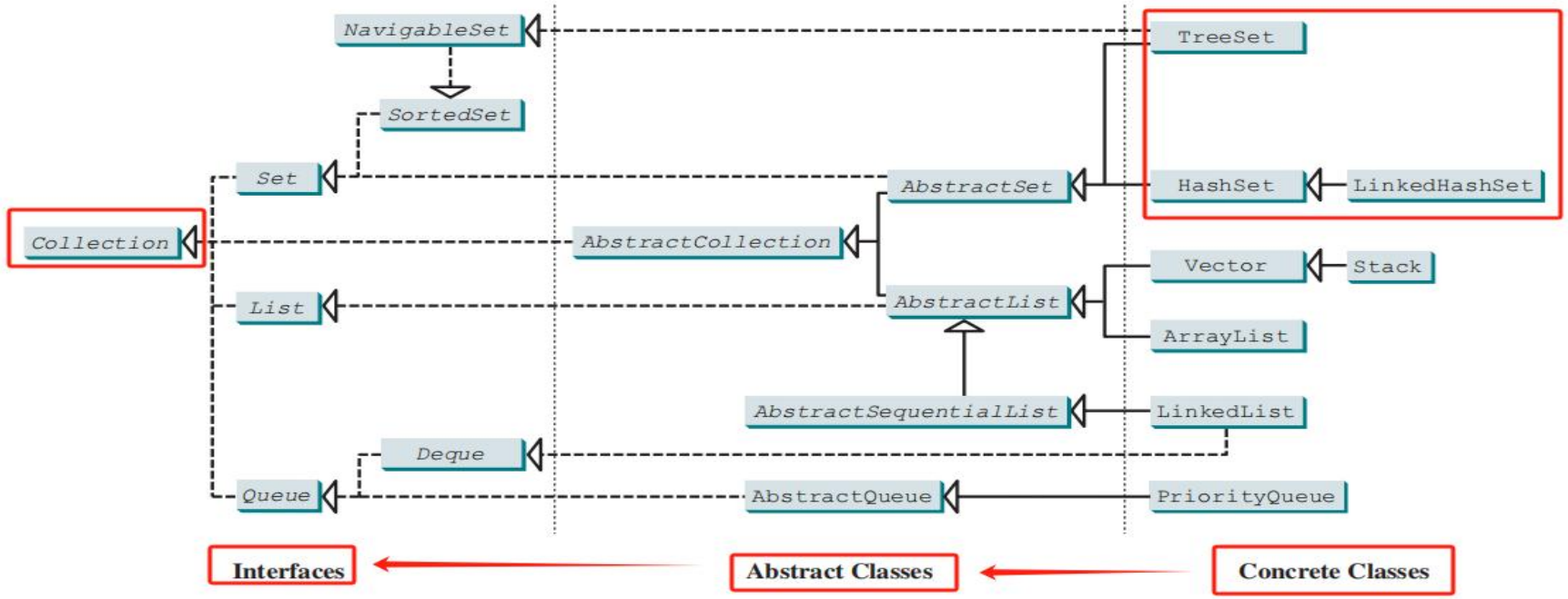
  3 and 4. You can an empty HashSet with the specified initial capacity only (case 3) or initial capacity plus loadFactor (case 4)

```
int initialCapacity = 16;
float loadFactor = 0.75f;
//Case 3;
HashSet<String> hashSet1 = new HashSet<>(initialCapacity);
//Case 4;
HashSet<String> hashSet2 = new HashSet<>(initialCapacity,loadFactor);
```

  - By default, the initial capacity is **16** and the load factor is **0.75**
  - Loadfactor ranges from 0.0 to 1.0, measuring how full the set is allowed to be before its capacity is increased (doubled; x2)
  - E.g., the capacity is **16**and load factor is **0.75**, when the size reaches **12**(**16*0.75 = 12**) the capacity will be doubled to **32 (16x2)**

2.2 HashSet: Method

- The interfaces (e.g., Collection) and abstract classes (e.g., AbstractSet) will be implemented/extended by the concrete classes (i.e., HashSet, LinkedHashSet, TreeSet). So, all the declared methods (e.g., add(), remove(), etc) can be called in a set instance.



2.2 HashSet: add()

- Adding elements to set
- Printed: [San Francisco, Beijing, New York, London, Paris]
- A hash set is **unordered** (because of hashing, W13)

```java
public class TestMethodsInCollection {
    public static void main(String[] args) {
        // Create set1
        java.util.Set<String> set1 = new java.util.HashSet<>();

        // Add strings to set1
        set1.add("London");
        set1.add("Paris");
        set1.add("New York");
        set1.add("San Francisco");
        set1.add("Beijing");

        System.out.println("set1 is " + set1);
        System.out.println(set1.size() + " elements in set1");
```

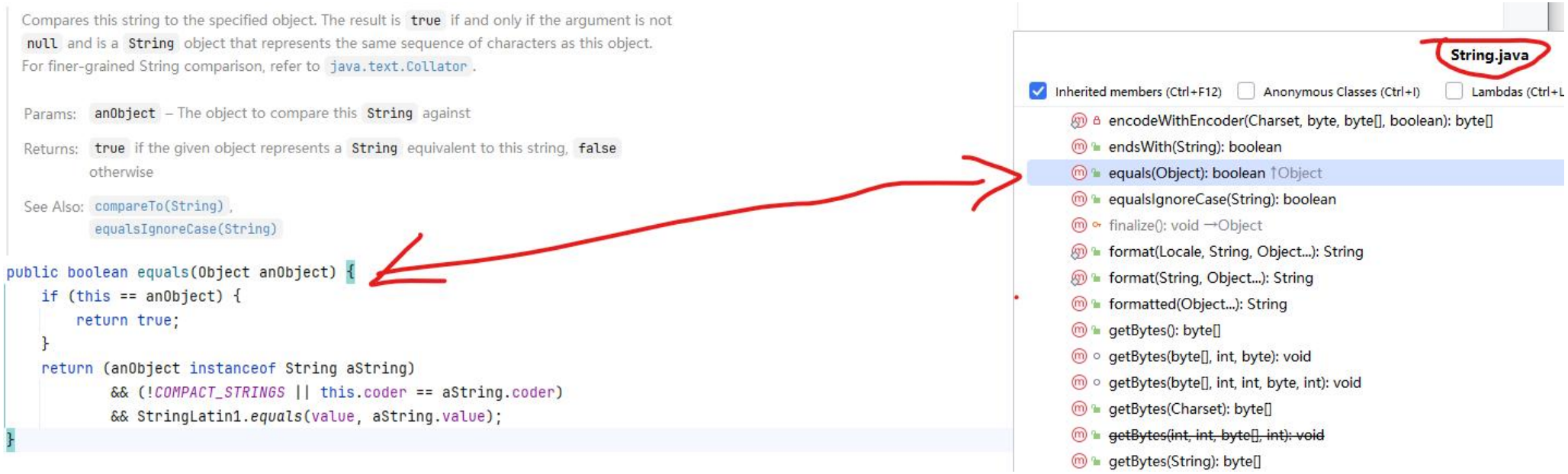
2.2 HashSet: add()

- Adding a duplicated element to set “New York”
- Printed: [San Francisco, Beijing, New York, London, Paris]
- A hash set is **non-duplicated** but **WHY?**

```
7         HashSet<String> set = new HashSet<>();
8
9         // Add strings to the set
10        set.add("London");
11        set.add("Paris");
12        set.add("New York");
13        set.add("San Francisco");
14        set.add("Beijing");
15        set.add("New York");
16
17        System.out.println(set);
```

2.2 HashSet: add()

- Click <String> -> “F4” -> “Ctrl + F12”
- A hash set use **hashcode()** and **equals()** to check duplication
- These 2 methods are "built-in" in the **String object,** as they are part of the standard String class in Java (Same as other objects from the standard Java library like **Integer**)



2.2 HashSet: add()

- **Try**: What if we create customized Objects, like the Person object on the left figure? What would be the printed result on the right?

```java
import java.util.Objects;

public class Person {  4 usages
    private String name;  2 usages
    private int age;  2 usages
    public Person(String name, int age) {  3 usages
        this.name = name;
        this.age = age;
    }
    @Override
    public String toString() {
        return "Person{" +
                "name='" + name + '\'' + ", age=" + age + '}';
    }
}
```

```java
Set<Person> set1 = new HashSet<>();
set1.add(new Person( name: "John", age: 19));
set1.add(new Person( name: "Mary", age: 20));
set1.add(new Person( name: "John", age: 19));
System.out.println(set1);
```

2.3 HashSet: Enhanced for loop

- **Collection interface extends the Iterable interface (Textbook Page 778), so the elements in a set are iterable**
- **Way 1: Enhanced for loop**

  for (declaration : expression) {

  // Statements}

```
// Display the elements in the hash set
for (String s: set) {
    System.out.print(s.toUpperCase() + " ");
}
```

  - **Declaration:** the part where you declare a variable that will hold an element of the array or collection you're iterating over
  - **Expression:** the collection or array you want to iterate over; the target
  - Enhanced for loop is used because a hash set is unordered without index (No [i])

2.4 HashSet: forEach()

- **Collection interface extends the Iterable interface (Textbook Page 778), so the elements in a set are iterable**
- **Way 2: forEach()**
  - **A default method in the Iterable interface**
  - **Set.forEach(e -> System.out.print())**
  - **e** is the parameter passed to the lambda expression. It represents the current element of the set
  - **->** is the lambda arrow which separates the parameters of the lambda expression from its body

```
    // Process the elements using a forEach method
    System.out.println();
    set.forEach(e -> System.out.print(e.toLowerCase() + " "));
}
```

2.5 HashSet: Other Common Methods

```
20          // remove(): Delete a string from set1
21          set1.remove( o: "London");
22          System.out.println("\nset1 is " + set1);
23
24          // size(): the size of the set
25          System.out.println(set1.size() + " elements in set1");
26
27          // contains(): if the set contains a certain element, return T/F
28          System.out.println("\nIs Taipei in set2? "
29                  + set2.contains("Taipei"));
30          // (!) addAll(): add the elements in set1 and set2 together. NO DUPILICATION!
31          // hashcode() and equals() are called
32          set1.addAll(set2);
33          System.out.println("\nAfter adding set2 to set1, set1 is "
34                  + set1);
35          // removeAll(): removing the elements in set 2 from set1
36          set1.removeAll(set2);
37          System.out.println("After removing set2 from set1, set1 is "
38                  + set1);
39          // (?) retainAll(): What is the printed result in this case?
40          set1.retainAll(set2);
41          System.out.println("After retaining common elements in set1 "
42                  + "and set2, set1 is " + set1);
43      }
44  }
45
```

3.1 LinkedHashSet: Basics

- LinkedHashSet extends HashSet with a linked list implementation that supports an ordering of the elements in the set
- Very similar to HashSet, those we have acquired previously (e.g., the set creation and the methods can be called) are applicable to LinkedHashSet
- **Significant Difference:** The elements in a LinkedHashSet can be retrieved in the order in which they were inserted into the set

3.2 LinkedHashSet: Example

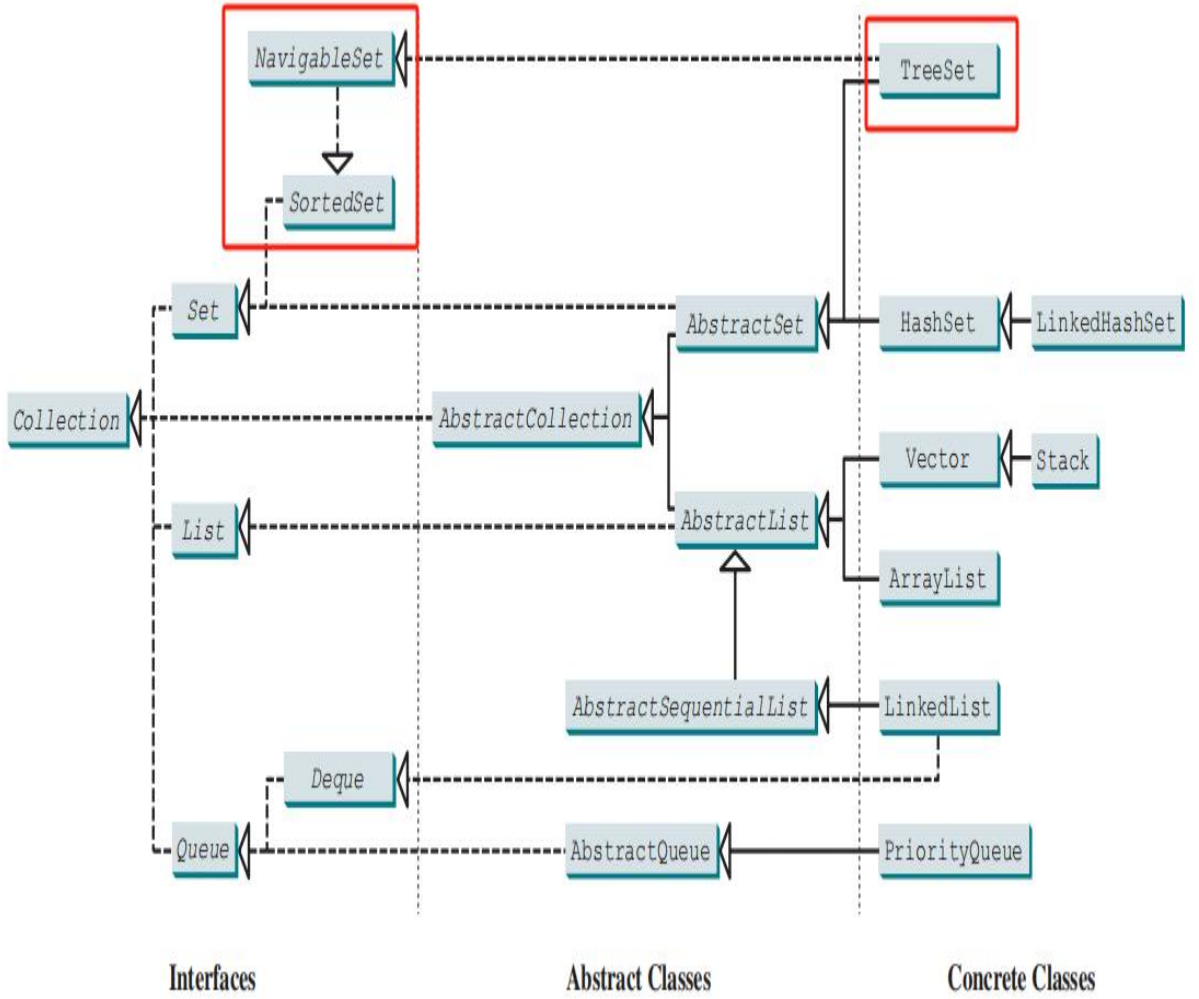
```java
import java.util.*;

public class TestLinkedHashSet {
    public static void main(String[] args) {
        // Create a linked hash set
        Set<String> set = new LinkedHashSet<>();

        // Add strings to the set
        set.add("London");
        set.add("Paris");
        set.add("New York");
        set.add("San Francisco");
        set.add("Beijing");
        set.add("New York");

        System.out.println(set);

        // Display the elements in the hash set
        for (String element: set)
            System.out.print(element.toLowerCase() + " ");
    }
}

```

Printed Result:

```
[London, Paris, New York, San Francisco, Beijing]
london paris new york san francisco beijing
Process finished with exit code 0
```

4.1 TreeSet: Basics

- **`TreeSet`** is a concrete class that implements the **`SortedSet`** and **`NavigableSet`** interfaces
- **`SortedSet`** is asub-interface of **`Set`**, which guarantees that the elements in the set are sorted
- **`NavigableSet`** extends **`SortedSet`** to provide navigation methods (e.g., **`lower(e)`**, **`floor(e)`**,etc)



4.2 TreeSet: Creation

- Empty tree set without arguement, which will be sorted in ascending order according to the **natural ordering** of its elements. (Due to the implementation of **SortedSet** interface)
  - TreeSet<String> treeSet = new TreeSet<>();
- Tree set with other collections, being sorted by the natural ordering.
  - TreeSet<String> treeSet = new TreeSet<>(list)
- Tree set with customized comparator,where we can define the orders
  - TreeSet<String> treeSet = new TreeSet<>(Comparator.reverseOrder())
- Tree set with the same elements and the same ordering as the specified sorted set
  - // If we already have a set sorted according to a specific rule

```
SortedSet<String> originalSet = new TreeSet<>(String.CASE_INSENSITIVE_ORDER);
    // We take this way to create another tree set with same elements and same ordering
TreeSet<String> copiedSet = new TreeSet<>(originalSet);
```

4.3 TreeSet: add()

- Similar to a hash set, the duplicted elements would not be added to the tree set
- Instead of hashcode() and equals(), this is because the “built-in” compareTo() in String and Integer and other wrapper classes in Java's standard library.
- The difference is due to the bottomed data structure, hash set -> Hashing (W13), tree set -> Tree (11&12).

```java
import java.util.TreeSet;

public class TreeSetTest {
    public static void main(String[] args) {
        // Create a tree set
        TreeSet<String> treeSet = new TreeSet<>();

        // Add elements
        treeSet.add("Apple");
        treeSet.add("Banana");
        treeSet.add("Cherry");
        treeSet.add("Apple");  // Duplicated element will not be inserted
        treeSet.add("Date");
        treeSet.add("Banana"); // Duplicated element will not be inserted

        // Print
        System.out.println("TreeSet contents: " + treeSet);
    }
}
```

4.3 TreeSet: add()

- Similar to a hash set, the duplicted elements would not be added to the tree set
- Instead of hashcode() and equals(), this is because the “built-in” compareTo() in String and Integer and other wrapper classes in Java's standard library.
- The difference is due to the bottomed data structure, hash set -> Hashing (W13), tree set -> Tree (11&12).

```java
import java.util.TreeSet;

public class TreeSetTest {
    public static void main(String[] args) {
        // Create a tree set
        TreeSet<String> treeSet = new TreeSet<>();

        // Add elements
        treeSet.add("Apple");
        treeSet.add("Banana");
        treeSet.add("Cherry");
        treeSet.add("Apple");  // Duplicated element will not be inserted
        treeSet.add("Date");
        treeSet.add("Banana"); // Duplicated element will not be inserted

        // Print
        System.out.println("TreeSet contents: " + treeSet);
    }
}
```

4.4 TreeSet: Common Method in SortedSet

- first(): return the first element in the set
- last(): return the last element in the set
- headSet(): find the elements that are less than or equal to the given toElement (i.e., "New York")
- tailSet(): find the elements that are equal to or bigger than the given toElement (i.e., "New York")

- More intuitivelly, consider the following:

TreeSet<Integer> numbers = new TreeSet<>(Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9));

SortedSet<Integer> headSet = numbers.headSet(5);

```
1   import java.util.*;
2
3   public class TestTreeSet {
4       public static void main(String[] args) {
5           // Create a hash set
6           Set<String> set = new HashSet<>();
7
8           // Add strings to the set
9           set.add("London");
10          set.add("Paris");
11          set.add("New York");
12          set.add("San Francisco");
13          set.add("Beijing");
14          set.add("New York");
15
16          TreeSet<String> treeSet = new TreeSet<>(set);
17          System.out.println("Sorted tree set: " + treeSet);
18
19          // Use the methods in SortedSet interface
20          System.out.println("first(): " + treeSet.first());
21          System.out.println("last(): " + treeSet.last());
22          System.out.println("headSet(\"New York\"): " +
23                  treeSet.headSet( toElement: "New York"));
24          System.out.println("tailSet(\"New York\"): " +
25                  treeSet.tailSet( fromElement: "New York"));
```

4.4 TreeSet: Common Method in Navigable

```
27        TreeSet<Integer> treeSet1 = new TreeSet<>(Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9));
28        // Use the methods in NavigableSet interface
29        System.out.println("lower(\"5\"): " + treeSet1.lower( e: 5));
30        System.out.println("higher(\"5\"): " + treeSet1.higher( e: 5));
31        System.out.println("floor(\"5\"): " + treeSet1.floor( e: 5));
32        System.out.println("ceiling(\"5\"): " + treeSet1.ceiling( e: 5));
33        System.out.println("pollFirst(): " + treeSet1.pollFirst());
34        System.out.println("pollLast(): " + treeSet1.pollLast());
35        System.out.println("New tree set: " + treeSet1);
```

- **lower():** Returns the greatest element in this set strictly less than the given element (4)
- **higher():** Returns the least element in this set strictly greater than the given element (?)
- **floor():** Returns the greatest element in this set less than or equal to the given element (5)
- **ceiling():** Returns the least element in this set greater than or equal to the given element (?)
- **pollFirst():** Retrieves and removes the first (lowest) element
- **pollLast():** Retrieves and removes the last (highest) element

5. Performance of Sets and Lists

```
import java.util.*;

public class SetListPerformanceTest {
  static final int N = 50000;

  public static long getTestTime(Collection<Integer> c) {
    long startTime = System.currentTimeMillis();

    // Test if a number is in the collection
    for (int i = 0; i < N; i++)
      c.contains((int)(Math.random() * 2 * N));

    return System.currentTimeMillis() - startTime;
  }

  public static long getRemoveTime(Collection<Integer> c) {
    long startTime = System.currentTimeMillis();

    for (int i = 0; i < N; i++)
      c.remove(i);

    return System.currentTimeMillis() - startTime;
  }
```

5. Performance of Sets and Lists

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

  // Add numbers 0, 1, 2, ..., N - 1 to an array list
  //  to populate all data structures
  List<Integer> list = new ArrayList<>();
  for (int i = 0; i < N; i++)
    list.add(i);
  Collections.shuffle(list); // Shuffle the array list

  // Create a hash set, and test its performance
  Collection<Integer> set1 = new HashSet<>(list);
  System.out.println("Member test time for hash set is " +
    getTestTime(set1) + " milliseconds");
  System.out.println("Remove element time for hash set is " +
    getRemoveTime(set1) + " milliseconds");

  // Create a linked hash set, and test its performance
  Collection<Integer> set2 = new LinkedHashSet<>(list);
  System.out.println("Member test time for linked hash set is "
      + getTestTime(set2) + " milliseconds");
  System.out.println("Remove element time for linked hash set is "
      + getRemoveTime(set2) + " milliseconds");
```

5. Performance of Sets and Lists

```
    // Create a tree set, and test its performance
    Collection<Integer> set3 = new TreeSet<>(list);
    System.out.println("Member test time for tree set is " +
      getTestTime(set3) + " milliseconds");
    System.out.println("Remove element time for tree set is " +
      getRemoveTime(set3) + " milliseconds\n");


    // Create an array list, and test its performance
    Collection<Integer> list1 = new ArrayList<>(list);
    System.out.println("Member test time for array list is " +
      getTestTime(list1) + " milliseconds");
    System.out.println("Remove element time for array list is " +
      getRemoveTime(list1) + " milliseconds");

    // Create a linked list, and test its performance
    Collection<Integer> list2 = new LinkedList<>(list);
    System.out.println("Member test time for linked list is " +
      getTestTime(list2) + " milliseconds");
    System.out.println("Remove element time for linked list is " +
      getRemoveTime(list2) + " milliseconds");
  }
}
```

5. Performance of Sets and Lists

```
Member test time for hash set is 20 milliseconds
Remove element time for hash set is 27 milliseconds

Member test time for linked hash set is 27 milliseconds
Remove element time for linked hash set is 26
milliseconds

Member test time for tree set is 47 milliseconds
Remove element time for tree set is 34 milliseconds


Member test time for array list is 39802 milliseconds
Remove element time for array list is 16196 milliseconds

Member test time for linked list is 52197 milliseconds
Remove element time for linked list is 14870 milliseconds
```

5. Performance of Sets and Lists

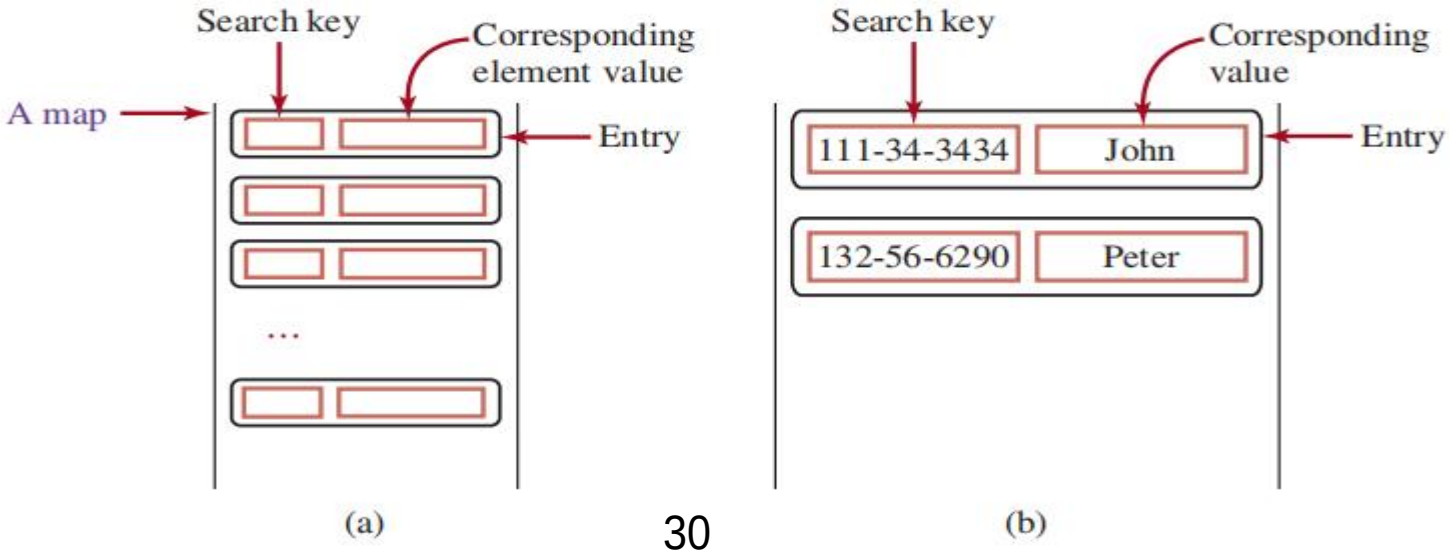
- Sets are more efficient than lists for storing nonduplicate elements
- Lists are useful for accessing elements through the index
- Sets **do not support indexing** because the elements in a set are unordered
  - To traverse all elements in a set, use a **for-each** loop or iterator

Quick Summary

- HashSet, LinkedHashSet, and TreeSet are all implementations of the Set interface in Java, which means they all share the fundamental characteristic of **not allowing duplicate elements**.
- **HashSet:**
  - Ordering: It does not guarantee any order of iteration.
  - Internal Structure: Backed by a hash table.
- **LinkedHashSet:**
  - Ordering: Maintains a doubly-linked list running through all its entries, which defines the iteration ordering, which is normally the order in which elements were inserted into the set (insertion-order).
  - Internal Structure: Backed by a hash table with a linked list running through it
- **TreeSet:**
  - Ordering: Guarantees that elements will be sorted in ascending element order, according to the natural ordering of the elements, or by a Comparator provided at set creation time.
  - Internal Structure: Backed by a tree

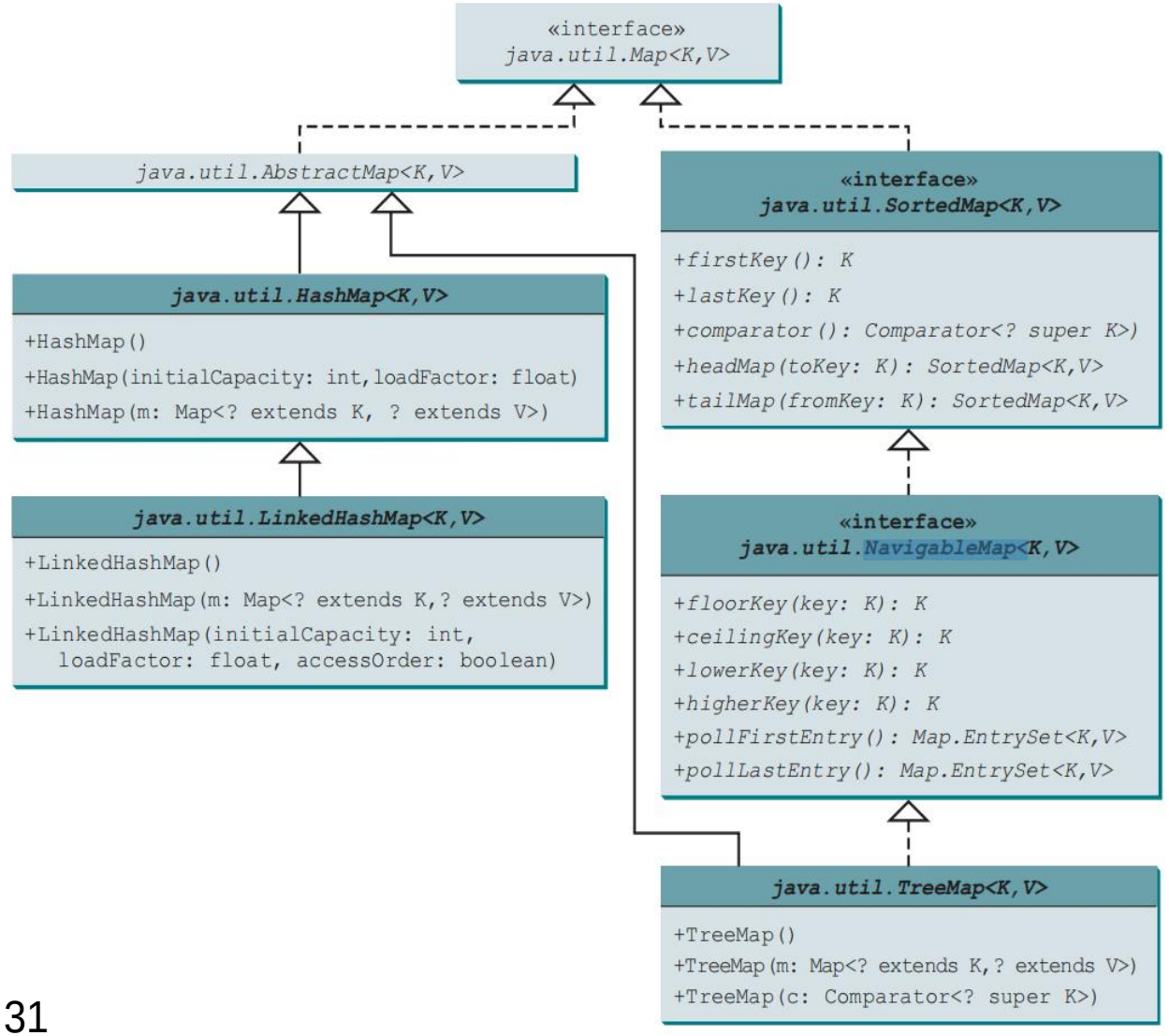
6.1 Maps: Basics

- A **map** is a container object that stores a collection of key/value pairs.
- It enables fast retrieval, deletion, and updating of the pair through the key. A map stores the values along with the keys.
- In List, the indexes are integers. In Map, the keys can be any objects.
  - A map cannot contain duplicate keys.
  - Each key maps to one value.



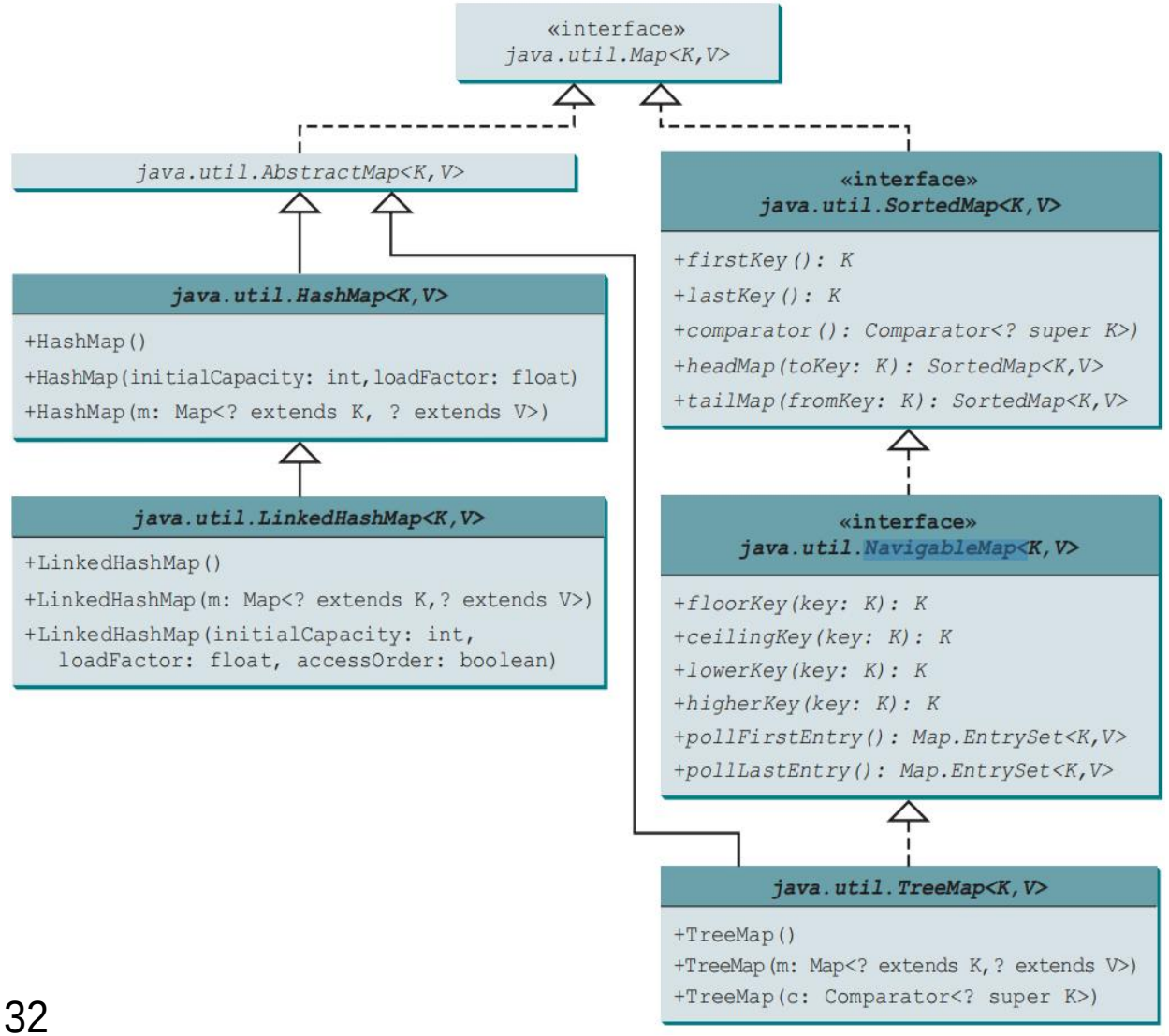
6.2 Maps: Three Types

- There are three types of maps: **HashMap**, **LinkedHashMap**, and **TreeMap**.
- Still, they ensure the map instances **non-duplicated** using hashcode() and equals() (for HashMap and LinkedHashMap) as well as the compareTo()/Comparator (for TreeMap).
- The HashMap, LinkedHashMap, and TreeMap classes are three concrete implementations of the Map interface, with TreeMap additionally implements SortedMap and NavigableMap



6.2 Maps: Three Types

- There are three types of maps: **HashMap**, **LinkedHashMap**, and **TreeMap**.
- Still, they ensure the map instances **non-duplicated** using hashcode() and equals() (for HashMap and LinkedHashMap) as well as the compareTo()/Comparator (for TreeMap).
- The HashMap, LinkedHashMap, and TreeMap classes are three concrete implementations of the Map interface, with TreeMap additionally implements SortedMap and NavigableMap



6.3 Maps: Creation

- We can create new hash/linked hash/tree maps with arguement (m: Map<? extends K,? extends V>)
- It indicates that the constructor accepts a Map <X, Y> (i.e., smallMap) where X is a subclass of K, and Y is a subclass of V (See figure).
- It would also work when X is exactly K, and Y is exactly V
  - Map<Integer, String> smallMap = new HashMap<>();
  - Map<Integer, String> largerMap = new HashMap<>(smallMap);

```java
import java.util.HashMap;
import java.util.Map;

public class HashMapExample {
    public static void main(String[] args) {
        // Create a map with Integer keys and String values
        Map<Integer, String> smallMap = new HashMap<>();
        smallMap.put(10, "Ten");
        smallMap.put(20, "Twenty");
        // Create a new HashMap using the constructor that accepts another map
        // In this case, smallMap's keys and values are instances of Number and Object.
        Map<Number, Object> largerMap = new HashMap<>(smallMap);

        System.out.println("Contents of largerMap: " + largerMap);
    }
}
```

subclass subclass

```
C:\Users\NINGMEI\.jdks\openjdk-22\bin\java.ex
Contents of largerMap: {20=Twenty, 10=Ten}

Process finished with exit code 0
```

6.3 Maps: Creation

- Usually, and similar to LinkedHashSet, LinkedHashMap extends HashMap with a linked-list implementation that supports retreiving elements in the **insertion order**.
- In LinkedHashMap, there is a constructor arguement (initialCapacity: int, loadFactor: float, accessOrder: boolean)
- Once it is set to be LinkedHashMap(initialCapacity, loadFactor, **true**), the created LinkedHashMap would allow us to retreive elements in the order in which they were last accessed, from least recently to most recently accessed (**access order**).

```
// Create a LinkedHashMap
Map<String, Integer> linkedHashMap =
        new LinkedHashMap<>( initialCapacity: 16, loadFactor: 0.75f, accessOrder: true);
linkedHashMap.put("Smith", 30);
linkedHashMap.put("Anderson", 31);
linkedHashMap.put("Lewis", 29);
linkedHashMap.put("Cook", 29);
// Display the map before any element is accessed
System.out.println("\nDisplay before any access");
System.out.println(linkedHashMap);
// Access Lewis to get his Age
System.out.println("\nThe age for " + "Lewis is " +
        linkedHashMap.get("Lewis"));
// Display the map after an element is accessed
System.out.println("After an element is accessed the entries in LinkedHashMap are\n\n");
System.out.println(linkedHashMap);
```

```
Display before any access
{Smith=30, Anderson=31, Lewis=29, Cook=29}

The age for Lewis is 29
After an element is accessed the entries in LinkedHashMap are
```

Least Recent → Most Recent

```
{Smith=30, Anderson=31, Cook=29, Lewis=29}
```

6.4 Maps: Method

- **A hash map** is unordered, similar to a hash set
- **A tree map** is ordered by the keys of the involved elements (alphabetically in this case)
- **A linked hash map** can be ordered by the insertion order, and by the access order (accessOrder: True)
- **get():** Returns the value to which the specified key is mapped
- **forEach():**Performs the given action for each entry in this map until all entries have been processed or the action throws an exception.

  void forEach(BiConsumer<? super K, ? super V> action)
  - return nothing (void), just perform the action
  - K is the map's key, V is the map's value
  - Basically, forEach((key,value) -> action)

```
public class TestMap {
    public static void main(String[] args) {
        // Create a HashMap; Output {Lewis=29, Smith=30, Cook=29, Anderson=31}
        Map<String, Integer> hashMap = new HashMap<>();
        hashMap.put("Smith", 30);
        hashMap.put("Anderson", 31);
        hashMap.put("Lewis", 29);
        hashMap.put("Cook", 29);
        System.out.println("Display entries in HashMap");
        System.out.println(hashMap + "\n");

        // Create a TreeMap from the preceding HashMap; Output {Anderson=31, Cook=29, Lewis=29, Smith=30}
        Map<String, Integer> treeMap = new TreeMap<>(hashMap);
        System.out.println("Display entries in ascending order of key");
        System.out.println(treeMap);

        // Create a LinkedHashMap
        Map<String, Integer> linkedHashMap =
                new LinkedHashMap<>( initialCapacity: 16, loadFactor: 0.75f, accessOrder: true);
        linkedHashMap.put("Smith", 30);
        linkedHashMap.put("Anderson", 31);
        linkedHashMap.put("Lewis", 29);
        linkedHashMap.put("Cook", 29);
        // Display the map before any element is accessed
        System.out.println("\nDisplay before any access");
        System.out.println(linkedHashMap);
        // Access Lewis to get his Age
        System.out.println("\nThe age for " + "Lewis is " +
                linkedHashMap.get("Lewis"));
        // Display the map after an element is accessed
        System.out.println("After an element is accessed the entries in LinkedHashMap are\n\n");
        System.out.println(linkedHashMap);

        // Display each entry with name and age
        System.out.print("\nNames and ages are ");
        treeMap.forEach(
                (name, age) -> System.out.print(name + ": " + age + " "));
    }
}
```

6.4 Maps: Method

- Visit HashMaps4fun.java to practice a bit

```java
import java.util.HashMap;
import java.util.Map;
import java.util.Set;
import java.util.Collection;

public class HashMaps4Fun {
    public static void main(String[] args) {
        // Create a new HashMap
        Map<String, Integer> hashMap = new HashMap<>();

        // Put some key-value pairs into the map
        hashMap.put("One", 1);
        hashMap.put("Two", 2);
        hashMap.put("Three", 3);

        // Test containsKey method
        System.out.println("Does hashMap contain 'Two'? " + hashMap.containsKey("Two"));

        // Test containsValue method
        System.out.println("Does hashMap contain value '3'? " + hashMap.containsValue(3));

        // Test entrySet method
        Set<Map.Entry<String, Integer>> entries = hashMap.entrySet();
        System.out.println("Entry set: " + entries);
```

6.4 Maps: Method

- Visit HashMaps4fun.java to practice a bit

```java
public class TreeMaps4Fun {
    public static void main(String[] args) {
        // Create a TreeMap and add some entries
        NavigableMap<String, Integer> treeMap = new TreeMap<>();
        treeMap.put("Apple", 3);
        treeMap.put("Banana", 2);
        treeMap.put("Cherry", 5);
        treeMap.put("Date", 4);
        treeMap.put("Elderberry", 1);

        // Testing SortedMap methods
        System.out.println("First key: " + treeMap.firstKey());
        System.out.println("Last key: " + treeMap.lastKey());
        System.out.println("HeadMap (keys less than 'Date'): " + treeMap.headMap( toKey: "Date"));
        System.out.println("TailMap (keys greater than or equal to 'Date'): " + treeMap.tailMap( fromKey: "Date"));

        // Testing NavigableMap methods
        System.out.println("Lower key than 'Cherry': " + treeMap.lowerKey("Cherry"));
        System.out.println("Floor key of 'Cherry': " + treeMap.floorKey("Cherry"));
        System.out.println("Ceiling key of 'Cherry': " + treeMap.ceilingKey("Cherry"));
        System.out.println("Higher key than 'Cherry': " + treeMap.higherKey("Cherry"));

        // Polling entries
        System.out.println("Poll first entry: " + treeMap.pollFirstEntry());
        System.out.println("Poll last entry: " + treeMap.pollLastEntry());
        System.out.println("TreeMap after polling: " + treeMap);
    }
}
```