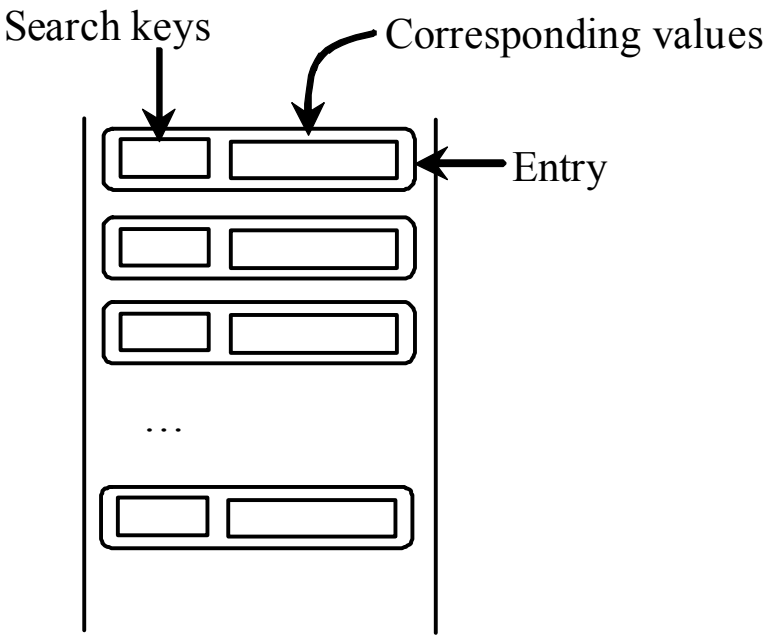
Java Collection
lecture-23

The Map Interface

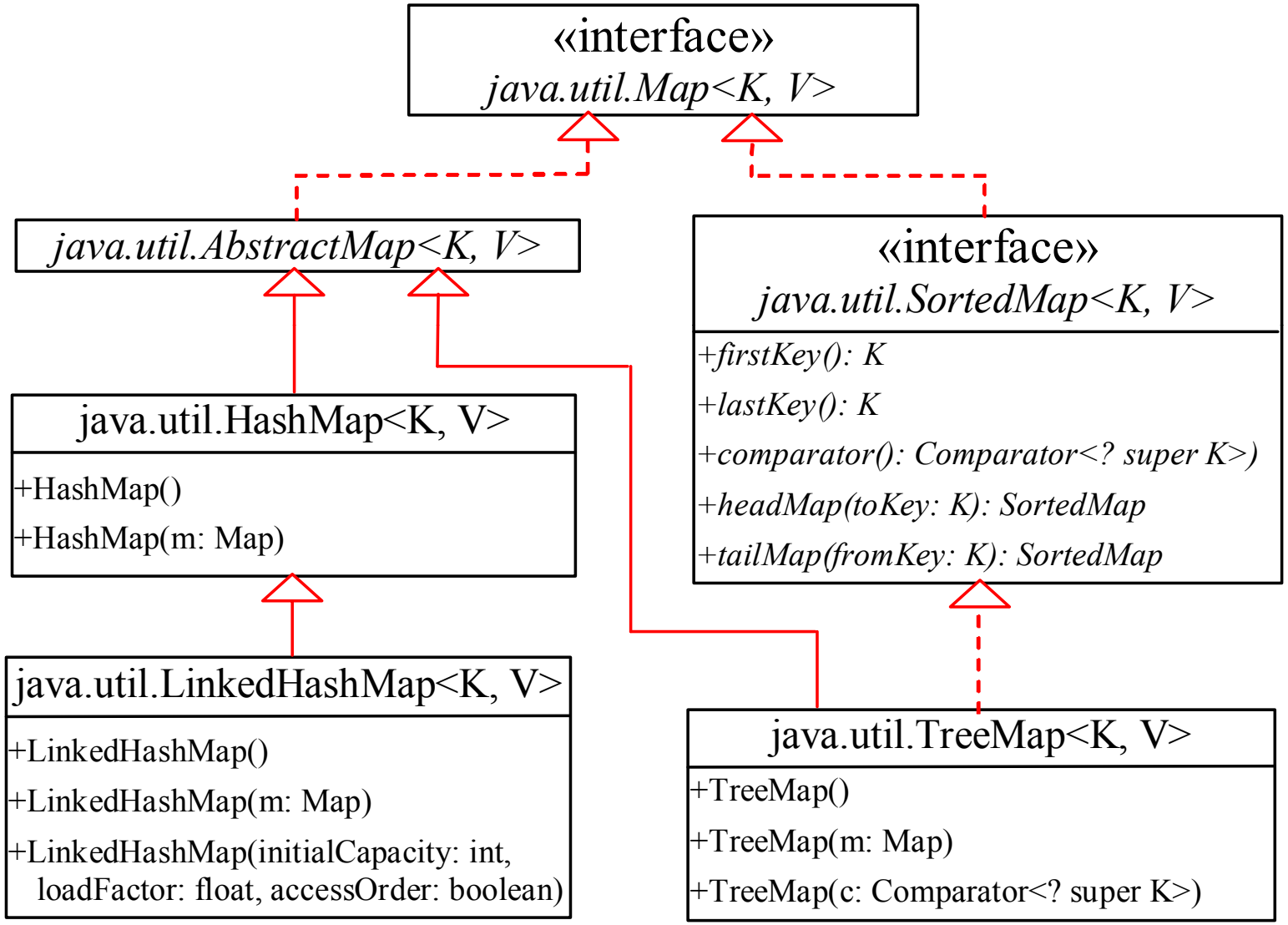
The Map interface maps keys to the elements. The keys are like indexes. In List, the indexes are integer. In Map, the keys can be any objects.



The Map Interface UML Diagram

java.util.Map<K, V>	
+clear(): void	Removes all mappings from this map.
+containsKey(key: Object): boolean	Returns true if this map contains a mapping for the specified key.
+containsValue(value: Object): boolean	Returns true if this map maps one or more keys to the specified value.
+entrySet(): Set	Returns a set consisting of the entries in this map.
+get(key: Object): V	Returns the value for the specified key in this map.
+isEmpty(): boolean	Returns true if this map contains no mappings.
+keySet(): Set<K>	Returns a set consisting of the keys in this map.
+put(key: K, value: V): V	Puts a mapping in this map.
+putAll(m: Map): void	Adds all the mappings from m to this map.
+remove(key: Object): V	Removes the mapping for the specified key.
+size(): int	Returns the number of mappings in this map.
+values(): Collection<V>	Returns a collection consisting of the values in this map.

Concrete Map Classes



Map Operations

- The alteration operations allow you to add and remove key-value pairs from the map. Both the key and value can be null. However, you should not add a Map to itself as a key or value.

- Object put(Object key, Object value)
- Object remove(Object key)
- void putAll(Map mapping)
- void clear()

Map Operations

- The query operations allow you to check on the contents of the map:
- Object get(Object key)
- boolean containsKey(Object key)
- boolean containsValue(Object value)
- int size()
- boolean isEmpty()

Map Operations

- The last set of methods allow you to work with the group of keys or values as a collection.

- public Set keySet()
- public Collection values()
- public Set entrySet()

- Because the collection of keys in a map must be unique, you get a Set back. Because the collection of values in a map may not be unique, you get a Collection back

HashMap and TreeMap

•The HashMap and TreeMap classes are two concrete implementations of the Map interface.

•The HashMap class is efficient for locating a value, inserting a mapping, and deleting a mapping.

•The TreeMap class, implementing SortedMap, is efficient for traversing the keys in a sorted order.

- Depending upon the size of your collection, it may be faster to add elements to a HashMap, then convert the map to a TreeMap for sorted key traversal.

- Using a HashMap requires that the class of key added have a well-defined hashCode() implementation. With the TreeMap implementation, elements added to the map must be sortable

Example: Using HashMap and TreeMap

This example creates a hash map that maps borrowers to mortgages.

The program first creates a hash map with the borrower's name as its key and mortgage as its value.

The program then creates a tree map from the hash map, and displays the mappings in ascending order of the keys.

Example: HashMap and TreeMap

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

    // Create a TreeMap from the previous HashMap
    Map<String, Integer> treeMap =
      new TreeMap<String, Integer>(hashMap);
    System.out.println("\nDisplay entries in ascending order of key");
    System.out.println(treeMap);
```

output

- Display entries in HashMap
- {Smith=30, Lewis=29, Anderson=31, Cook=29}

- Display entries in ascending order of key
- {Anderson=31, Cook=29, Lewis=29, Smith=30}

HashSet and TreeSet

- The Collections Framework provides two general-purpose implementations of the Set interface: HashSet and TreeSet.
- More often than not, you will use a HashSet for storing your duplicate-free collection.
- For efficiency, objects added to a HashSet need to implement the hashCode() method in a manner that properly distributes the hash codes.
- The TreeSet implementation is useful when you need to extract elements from a collection in a sorted manner.
- In order to work properly, elements added to a TreeSet must be sortable

Example:HashSet & TreeSet

```
import java.util.*;
public class SetExample {
public static void main(String args[]) {
Set set = new HashSet();
set.add("Bernadine");
set.add("Elizabeth");
set.add("Gene");
set.add("Elizabeth");
set.add("Clara");
System.out.println(set);
Set sortedSet = new TreeSet(set);
System.out.println(sortedSet);
}
}
```

output

- [Gene, Clara, Bernadine, Elizabeth]
- [Bernadine, Clara, Elizabeth, Gene

AbstractSet class

- The AbstractSet class overrides the *equals() and hashCode() methods to ensure* two equal sets return the same hash code.
- Two sets are equal if they are the same size and contain the same elements.
- By definition, the hash code for a set is the sum of the hash codes for the elements of the set.
- Thus, no matter what the internal ordering of the sets, two equal sets will report the same hash code.

ListIterator

The ListIterator interface extends the Iterator interface to support bi-directional access, as well as adding or changing elements in the underlying collection.

Notice that the ListIterator is originally positioned beyond the end of the list (list.size()), as the index of the first element is 0.

```
ListIterator iterator = list.listIterator(list.size());
while (iterator.hasPrevious()) {
Object element = iterator.previous();
// Process element
}
```

Sorting

- Classes like String and Integer now implement the Comparable interface to provide a natural sorting order.
- For those classes without a natural order, or when you desire a different order than the natural order, you can implement the Comparator interface to define your own.
- To take advantage of the sorting capabilities, the Collections Framework provides two interfaces that use it:
  - SortedSet and SortedMap.

The Comparator Interface

Sometimes you want to insert elements of different types into a tree set. The elements may not be instances of Comparable or are not comparable. You can define a comparator to compare these elements. To do so, create a class that implements the java.util.Comparator interface. The Comparator interface has two methods, compare and equals.

The Comparator Interface

public int compare(Object element1, Object element2)

Returns a negative value if element1 is less than element2, a positive value if element1 is greater than element2, and zero if they are equal.

public boolean equals(Object element)

Returns true if the specified object is also a comparator and imposes the same ordering as this comparator.

Example

```
import java.util.Comparator;
public class GeometricObjectComparator implements
    Comparator<GeometricObject {
public int compare(GeometricObject o1, GeometricObject o2) {
    double area1 = o1.getArea();
 double area2 = o2.getArea();
    if (area1 < area2) return -1;
     else if (area1 == area2) return 0;
     else return 1; } }
```

Example: The Using Comparator to Sort Elements in a Set

Write a program that demonstrates how to sort elements in a tree set using the Comparator interface. The example creates a tree set of geometric objects. The geometric objects are sorted using the compare method in the Comparator interface.

example

```
import java.util.*;
public class TestTreeSetWithComparator {
 public static void main(String[] args) {
// Create a tree set for geometric objects using a comparator
Set<GeometricObject> set = new TreeSet<GeometricObject>(new
    GeometricObjectComparator());
set.add(new Rectangle(4, 5));
set.add(new Circle(40));
set.add(new Circle(40));
set.add(new Rectangle(4, 1));
// Display geometric objects in the tree set
System.out.println("A sorted set of geometric objects");
for (GeometricObject element: set)
        System.out.println("area = " + element.getArea()); }
```