G52APR

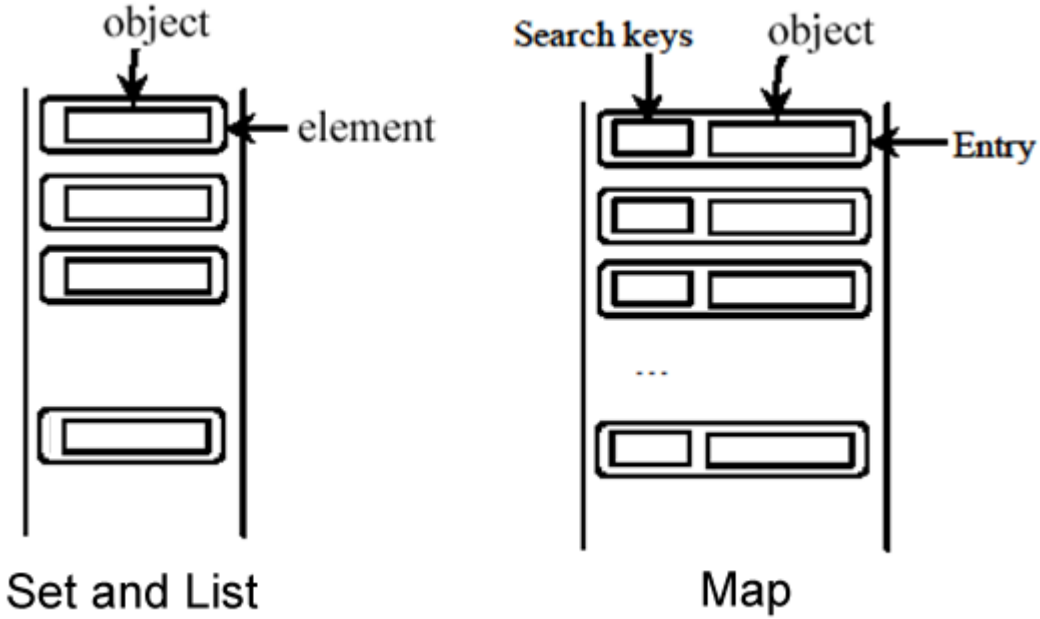
Application Programming

Java Collections

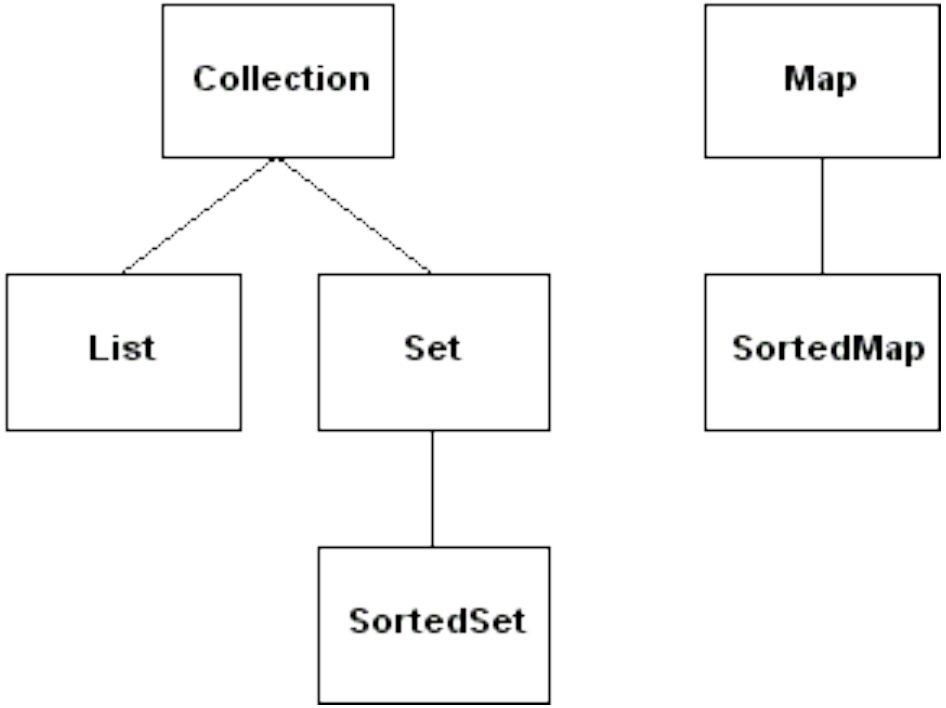
Colin Higgins

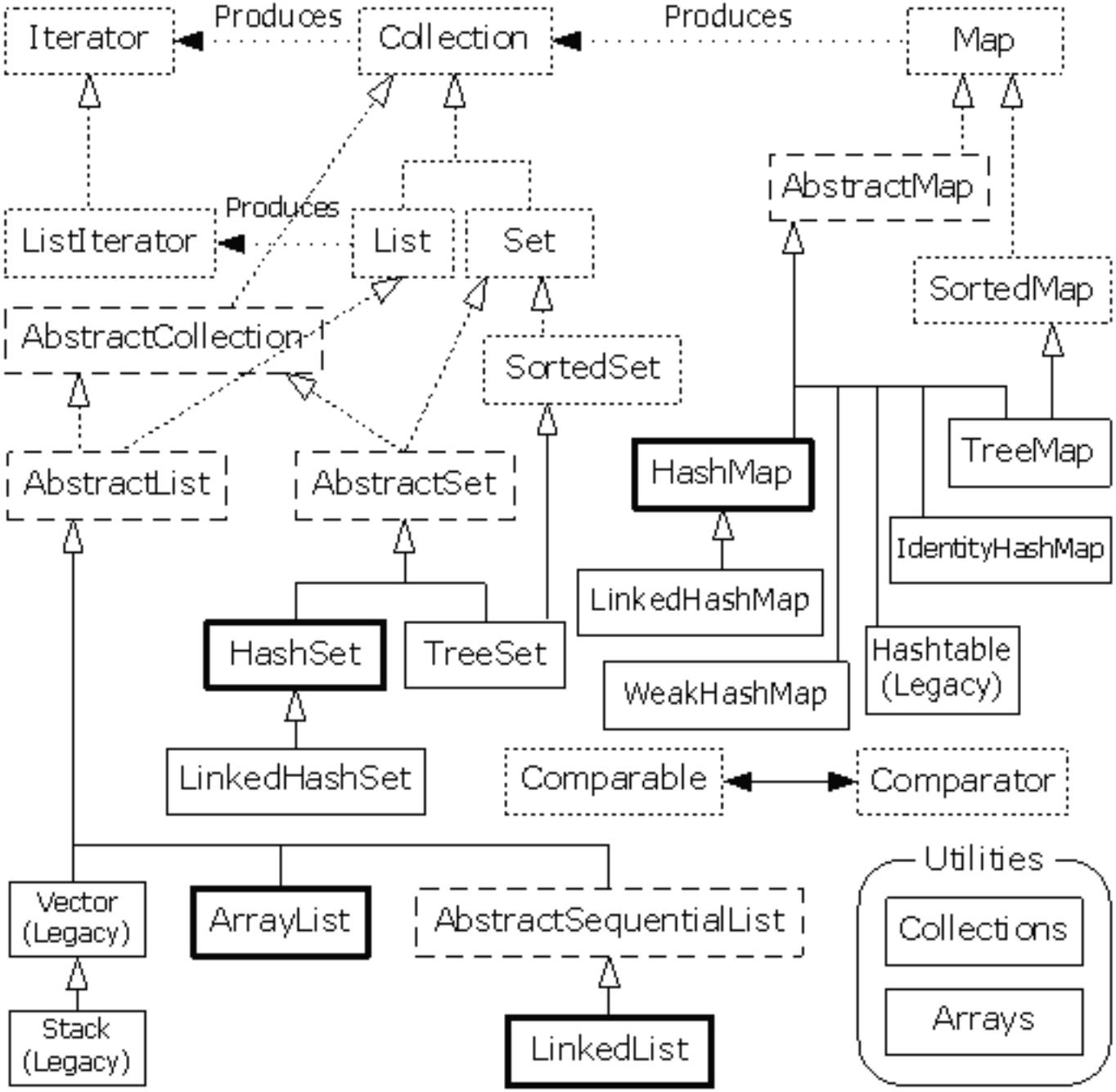
What are Java Collections

- A collection is an object that represents a group of objects
- Three types of Java collections: Set, List, and Map

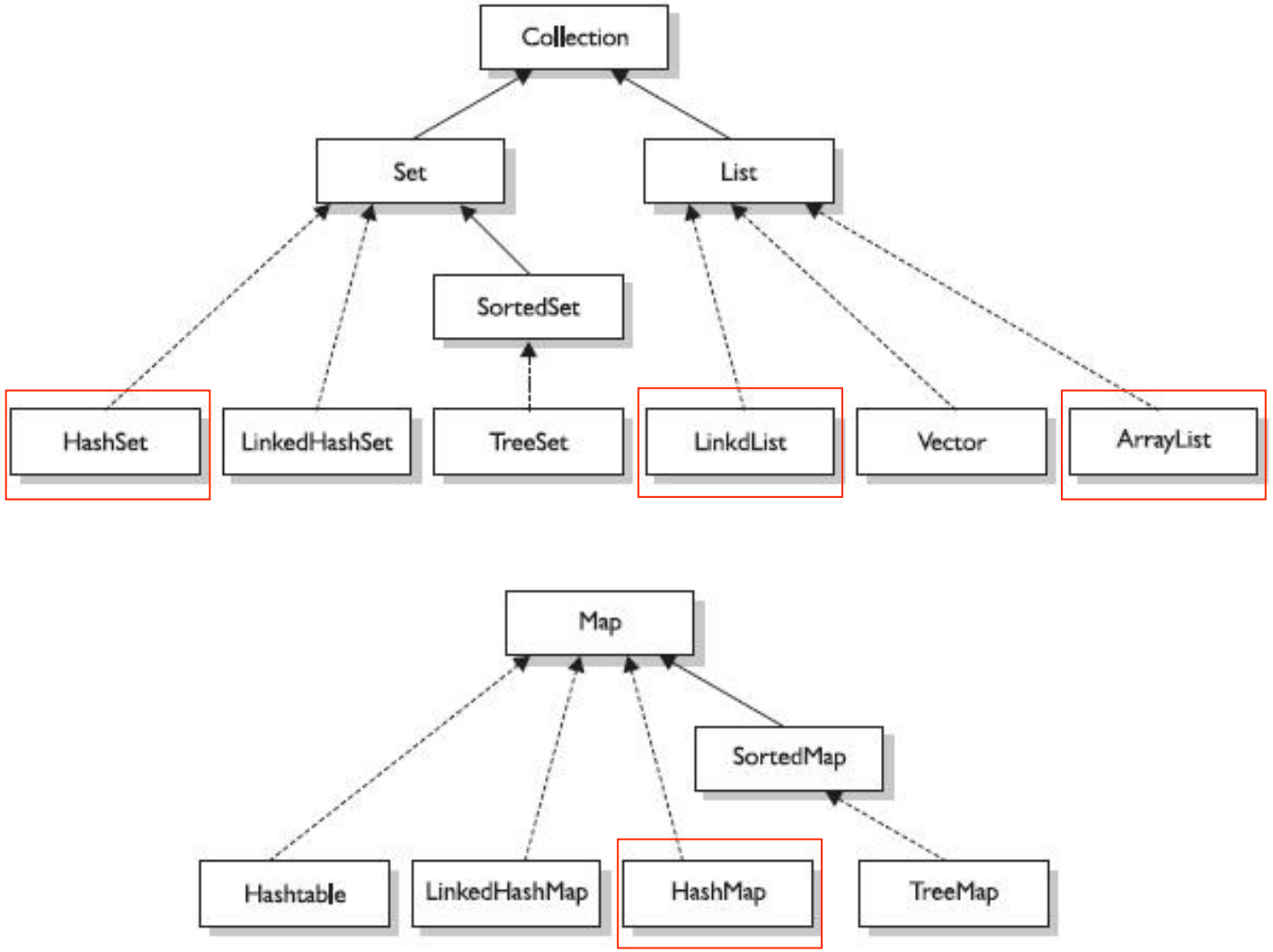


Interface hierarchy




Iterator
Produces
Collection
Produces
Map
ListIterator
Produces
List
Set
AbstractMap
SortedMap
AbstractCollection
SortedSet
AbstractList
AbstractSet
HashMap
TreeMap
IdentityHashMap
LinkedHashMap
HashSet
TreeSet
WeakHashMap
Hashtable
(Legacy)
LinkedHashSet
Comparable
Comparator
Utilities
Collections
Arrays
Vector
(Legacy)
ArrayList
AbstractSequentialList
Stack
(Legacy)
LinkedList

The collections class and interface hierarchy



What are Java Collections

- A collection is an object that represents a group of objects
- A number of pre-packaged implementations of common ‘container’ classes
- Collections API offers a structured paradigm for manipulating groups of objects in an ordered fashion, and employs design patterns (e.g., the iterator pattern) that make common programming tasks much simpler.

Why use Collections

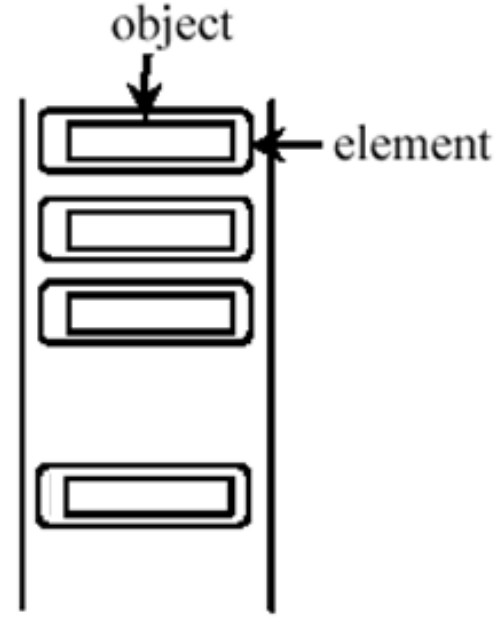
- Reduces programming effort
- Increases performance
- Provides interoperability between unrelated APIs
- Reduces the effort required to learn APIs
- Reduces the effort required to design and implement APIs
- Fosters software reuse

Java Collections

- Several implementations associated with each of the basic interfaces
- Each has its own advantages/disadvantages
- Sets
  - HashSet, TreeSet (SortedSet)
- Lists
  - ArrayList, LinkedList
- Maps
  - HashMap, TreeMap (SortedMap)

Set and List

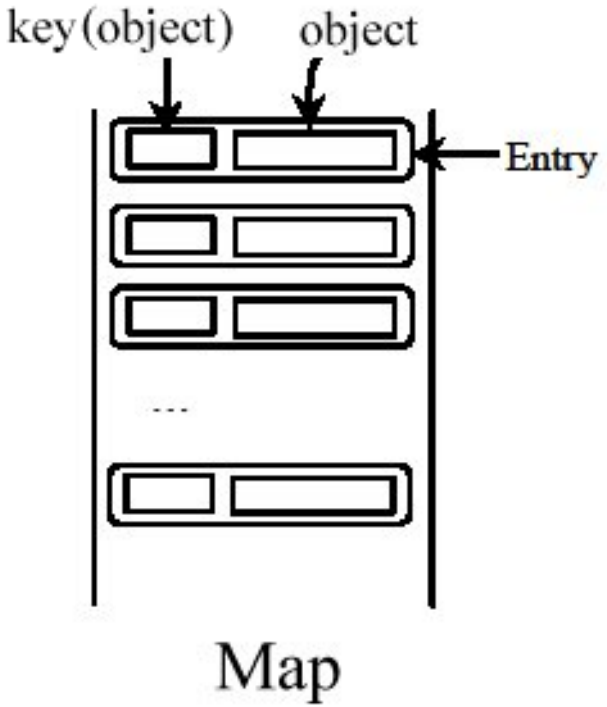
- Set: No duplicate elements permitted. May or may not be ordered.
- List: A sequence. Duplicates are generally permitted.



Set and List

Map

- Map: A mapping from keys to objects. Each key can also be an object
- No duplicate keys permitted
- The group of keys is a set



Java Collections -- classes

- Concrete classes that implement the interfaces
  - HashSet, TreeSet
  - ArrayList, LinkedList
  - HashMap, TreeMap

- Create a collection object: Generally hold references to the interface and not the specific class

```
Set mySet = new HashSet();
List myList = new ArrayList();
List otherList = new ArrayList(5);
Map database = new HashMap();
Collection aSet = new TreeSet();
```

Generics

- Allows abstraction of classes over Types
  - Common Example - Containers (Collections):
  - Container of <What> type
- Enables compiler to enforce Type Safety
  - Moves error checking from run-time (ClassCastException) to compile-time
- Improved Robustness
  - And Readability (?)
  - Yes, once you are familiar with syntax

Simple Example

- Before

```
List thingList = new ArrayList();
thingList.add( aThing );
Thing t = (Thing) thingList.get( 0 );
thingList.add( notAThing );
Thing t2 = (Thing) thingList.get( 1 ); // exception
```

- With Generics

```
List<Thing> thingList = new ArrayList<Thing>();
thingList.add( aThing );
Thing t = thingList.get( 0 ); // no cast needed
thingList.add( notAThing );   // compile error
```

More than just fewer casts or less typing!

…or Rearranged Clutter? It's more than that...

- Compile-time type checking
  - Errors show up where errant object is inserted into collection
    - Rather than ClassCastException when it is retrieved
    - Error occurs when (compile) and where the problem is
- Enforcement of relationships
  - Between various objects used by classes or methods
- Declaration of programmer's intent
  - Code says how this instance of the class is to be used

Basic generics syntax

- Example - `java.util.Map<K,V>`
- Map of Keys (K) and Values (V)
  - K and V are Parameterized Types
    - Convention is to use single-character capitalized symbols
  - `Map.java` uses K and V rather than "normal" types (Object)

```
public V put( K key, V val );
```

Put a K/V(key/value) pair in the map

```
public V get( K key );
```

Get V (value) for the key

```
public Collection<V> values();
```

Returns a Collection of V (value) types

```
public Set<Map.Entry<K,V>> entrySet();
```

Returns a Set of Map.Entry containing K's and V's

Classes with generics

- Concrete classes that implement the interfaces
  - HashSet, TreeSet
  - ArrayList, LinkedList
  - HashMap, TreeMap
  - etc

- Create a collection object: Generally hold references to the interface and not the specific class

```
Set<E> mySet = new HashSet<E>();
List<E> myList = new ArrayList<E>();
List<E> otherList = new ArrayList(E>(5);
Map<K, V> database = new HashMap<K, V>();
Collection<E> aSet = new TreeSet<E>();
```

Defining generic types - examples

```
public interface List<E> {
   void add(E x);
   Iterator<E> iterator();
}

public interface Iterator<E> {
   E next();
   boolean hasNext();
}

public interface Map<K,V> {
   V put(K key, V value);
}
```

Java Collections – Adding Items

- For Sets and Lists, use **`add`**`()`

```
List<String> myList = new ArrayList<String>();
myList.add("A String");
myList.add("Other String");
```

- For Maps, use **`put()`**

```
Map<String, String> myMap = new HashMap<String,
   String>();
myMap.put("google", "http://www.google.com");
mpMap.put("yahoo", "http://www.yahoo.com");
```

Collection – Copying

- Very easy, just use addAll()

```
List<T> myList = new ArrayList<T>();
//assume we add items to the list

List<T> otherList = new ArrayList<T>();
otherList.addAll(myList);
```

Collections – Getting Individual Items

- Use **`get`**()
- Lists

```
String s = myList.get(1); //get first element
String s2 = myList.get(10); //get tenth element
```

- Maps…

```
String s = myMap.get("google");
String s2 = myMap.get("yahoo");
```

Collections – Getting all items

- For Lists, we could use a **`for`** loop, and loop through the list to **`get`**`()` each item
- But this doesn't work for Sets and Maps.
- To allow generic handling of collections, Java defines an object called an *Iterator*
  - An object whose function is to walk through a Collection of objects and provide access to each object in sequence

Collections – Getting all items

- Get an iterator using the `iterator()` method
- Iterator objects have three methods:
  - `next()` – gets the next item in the collection
  - `hasNext()` – tests whether it has reached the end
  - `remove()` – removes the item just returned
- Basic iterators only go forwards
  - Lists objects have a ListIterator that can go forward and backward

Iterator examples

```
Set<String> aSet = new HashSet<String>();
Iterator<String> itr = aSet.iterator();
while(itr.hasNext()) {
    String element = itr.next();
    System.out.print(element + " ");
}
List<String> aList = new ArrayList<String>();
for(Iterator<String> itr = aList.iterator();
   itr.hasNext(); )
{
   System.out.println( itr.next() );
}
For (String s : aList ) {
   System.out.println( s );
}
```

Collections – Other Functions

- The `java.util.Collections` class has many useful methods for working with collections
  - min, max, sort, reverse, search, shuffle
- Virtually all require your objects to implement an extra interface, called `Comparable`

Collections – Comparable<E>

- The Comparable interface labels objects that can be compared to one another.
  - Allows sorting algorithms to be written to work on any kind of object
  - so long as they support this interface
- Single method to implement

```
public int compareTo(Object o);
```

- Returns
  - A negative number if parameter is less than the object
  - Zero if they're equal
  - A positive number if the parameter is greater than the object

Collections – Comparator

- Like Comparable, but is a stand-alone object used for comparing other objects
  - Useful when you want to use *your* criteria, not that of the implementor of the object.
  - Or altering the behaviour of a system
- Again has single method:

```
public int compare(Object obj1, Object
  obj2)
```

Comparator Example

- In this example, String comparison method `compareTo()` is adopted to override `compare()`

```
public class AlphaComparison implements
   Comparator<String> {

  public int compare(String obj1, String obj2) {
    String s1 = o1.toLowerCase();
    String s2 = o2.toLowerCase();
    return s1.compareTo(s2);
  }

}
```

The **Set** Interface

- The **Set** interface is used to represent an unordered collection of objects. The two concrete classes in this category are **HashSet** and **TreeSet**.
- A set is in some ways a stripped-down version of a list. Both structures allow you to add and remove elements, but the set form does not offer any notion of index positions. All you can know is whether an object is present or absent from a set.
- The Collections Framework provides two general-purpose implementations of the Set interface: HashSet and TreeSet. The **HashSet** class is built on the idea of hashing; the **TreeSet** class is based on a structure called a *binary tree*.

HashSet and TreeSet

- More often than not, you will use a HashSet for storing your duplicate free collection.
- HashSet allows at most one null element.
- HashSet is faster than other implementations of Set (TreeSet and LinkedHashSet).
- The add method of Set returns false if you try to add a duplicate element.
- 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

HashSet & TreeSet example

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

Output: [Bernadine, Elizabeth, Gene, Clara]
[Bernadine, Clara, Elizabeth, Gene]

Set Operations

```
public interface Set<E> extends Collection<E> {
// Basic operations
int size();
boolean isEmpty();
boolean contains(Object element);
boolean add(E element);
boolean remove(Object element);
Iterator<E> iterator();
// Bulk operations
boolean containsAll(Collection<?> c);
boolean addAll(Collection<? extends E> c);
boolean removeAll(Collection<?> c);
boolean retainAll(Collection<?> c);
void clear();
// Array Operations
Object[] toArray();
<T> T[] toArray(T[] a); }
```

The List interface

- A list is an ordered collection.
- Lists may contain duplicate elements.
- List interface includes operations for:
  - **Positional access**: manipulates elements based on their numerical position in the list
  - **Search**: searches for a specified object in the list and returns its numerical position
  - **Iteration**: extends Iterator semantics to take advantage of the list's sequential nature
  - **Range-view**: performs arbitrary *range operations* on the list.

List Operations

```
public interface List<E> extends Collection<E> {
// Positional access
E get(int index);
E set(int index, E element);
boolean add(E element);
void add(int index, E element);
E remove(int index);
boolean addAll(int index, Collection<? extends E> c);
// Search
int indexOf(Object o);
int lastIndexOf(Object o);
// Iteration
ListIterator<E> listIterator();
ListIterator<E> listIterator(int index);
// Range-view
List<E> subList(int from, int to);
```

List Iterators

- List's iterator returns the elements of the list in proper sequence
- **`ListIterator`** allows you to traverse the list in either direction, modify the list during iteration, and obtain the current position of the iterator.

```
hasNext();next();remove();
previous(); hasPrevious(); nextIndex();
previousIndex(); set(E e); add(E e);
```

List Iterator example

```
List<String> aList = new ArrayList<String>();
aList.add("1");
aList.add("2");
aList.add("3");
aList.add("4");
aList.add("5");
ListIterator<String> itr = aList.listIterator();

System.out.println(itr.previousIndex());
System.out.println(itr.nextIndex());
itr.next();
System.out.println(itr.previousIndex());
System.out.println(itr.nextIndex());
```

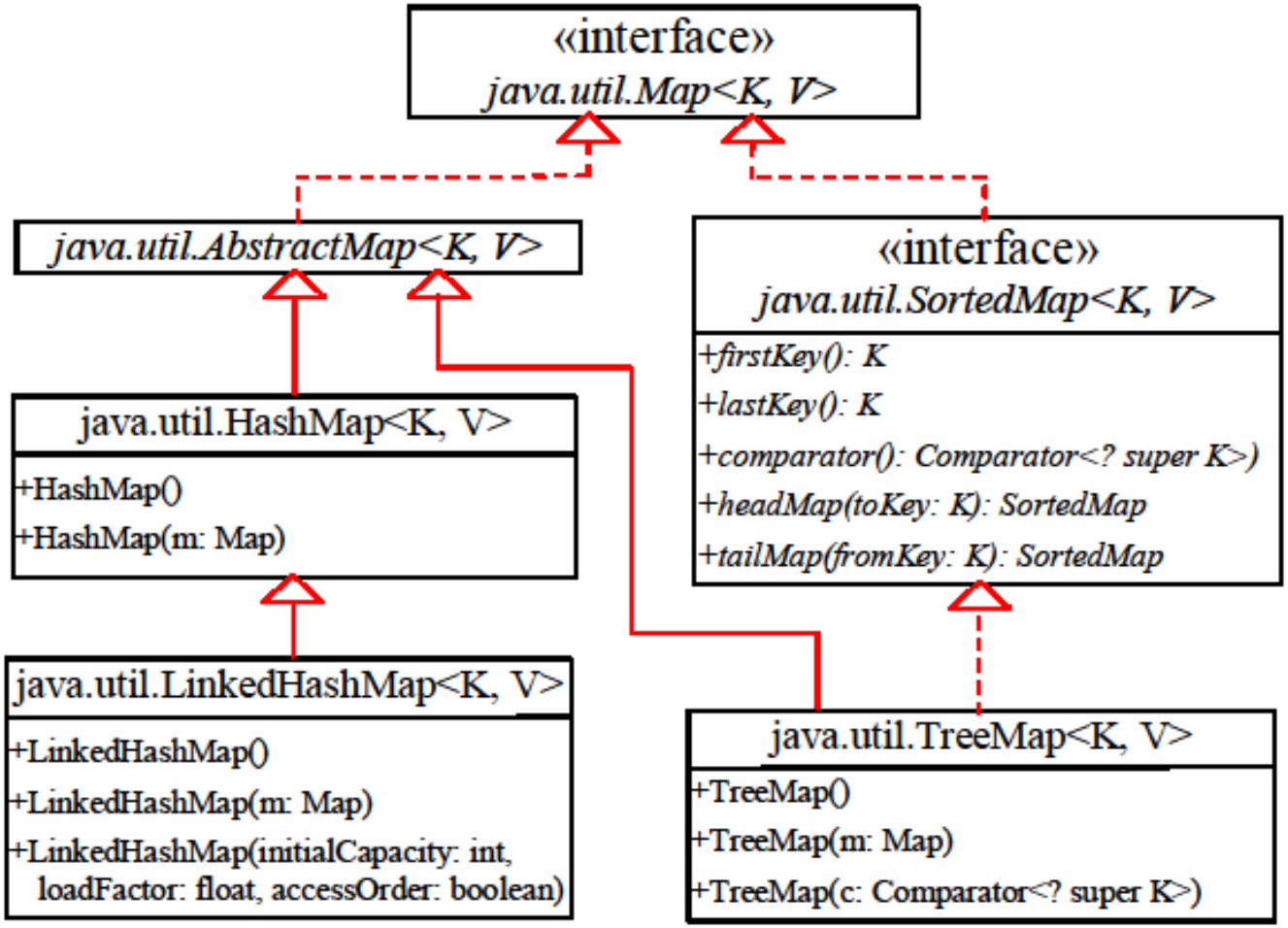
Output:

-1
0
0
1

Map interface

- A Map is an object that maps keys to values.
- A Map cannot contain duplicate keys: Each key can map to at most one value.
- It models the mathematical *function* abstraction.

Concrete Map Classes



4

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

Map Iteration

- No direct iteration over Maps
  - Maps do not provide an iterator() method as do Lists and Sets
- Three indirect ways: iteration over Sets, Collections, or key-value pairs
  - Get a Set of keys by keySet();
  - Get a Collection of values by values();
  - Get a Set of key-value pairs entrySet();

Map Iterator example

```
import java.util.*;
public class IterateValuesOfHashMapExample {
   public static void main(String[] args) {
        HashMap<String, String> hMap = new HashMap<String,
   String>();
        hMap.put("1","One");
        hMap.put("2","Two");
        hMap.put("3","Three");
        Collection<String> c = hMap.values();
        Iterator<String> itr = c.iterator();
        while(itr.hasNext())
            System.out.println(itr.next());
   }
}
```

Output: One
Two
Three

HashMap & 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.
- With the TreeMap implementation, elements added to the map must be sortable

Example: HashMap & TreeMap

```
public class TestMap {
   public static void main(String[] args) {
        // Create a HashMap
        Map<String, String> hashMap = new HashMap<String,
   String>();
        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, String> treeMap = new TreeMap<String,
   String> (hashMap);
        System.out.println("\nDisplay entries in ascending
   order of key");
        System.out.println(treeMap);
   }
}
```

Example: HashMap and 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}
```

Summary

- Three types of Java collections
  - Set
  - List
  - Map
- Iterator
- Comparable (Comparator)
  - HashSet and TreeSet
  - HashMap and TreeMap