

תוכנה 1

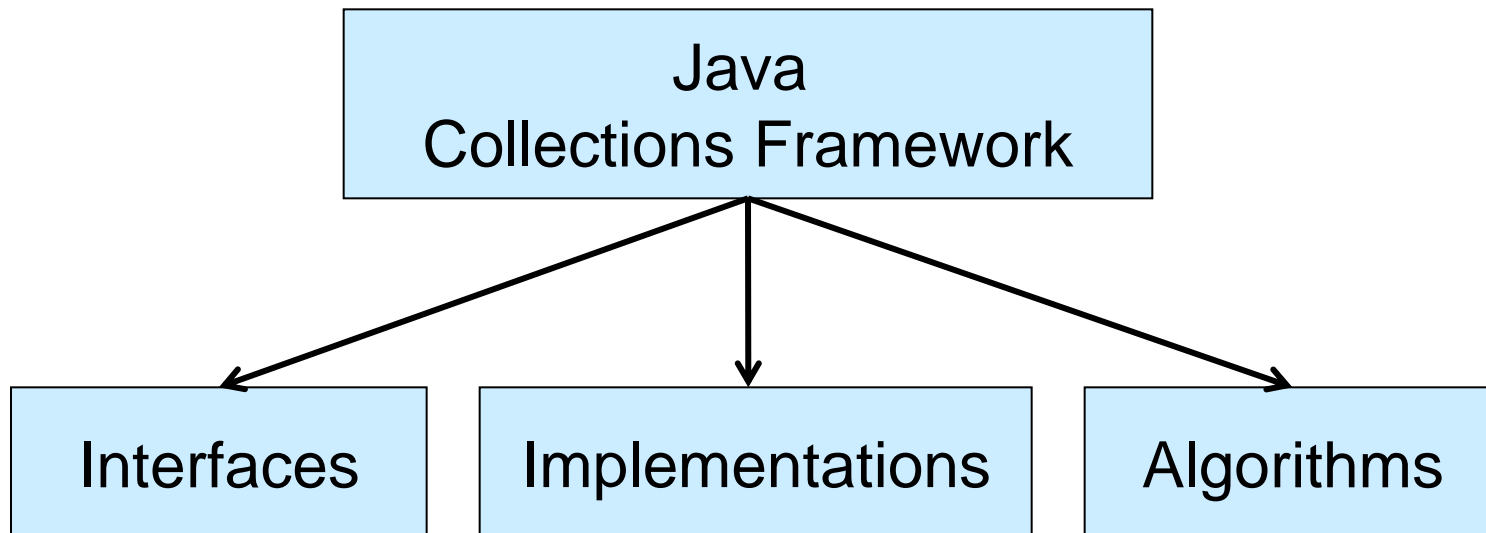
תרגול 8 – מבני נתונים גנריים

Java Collections Framework

- **Collection:**
a group of elements



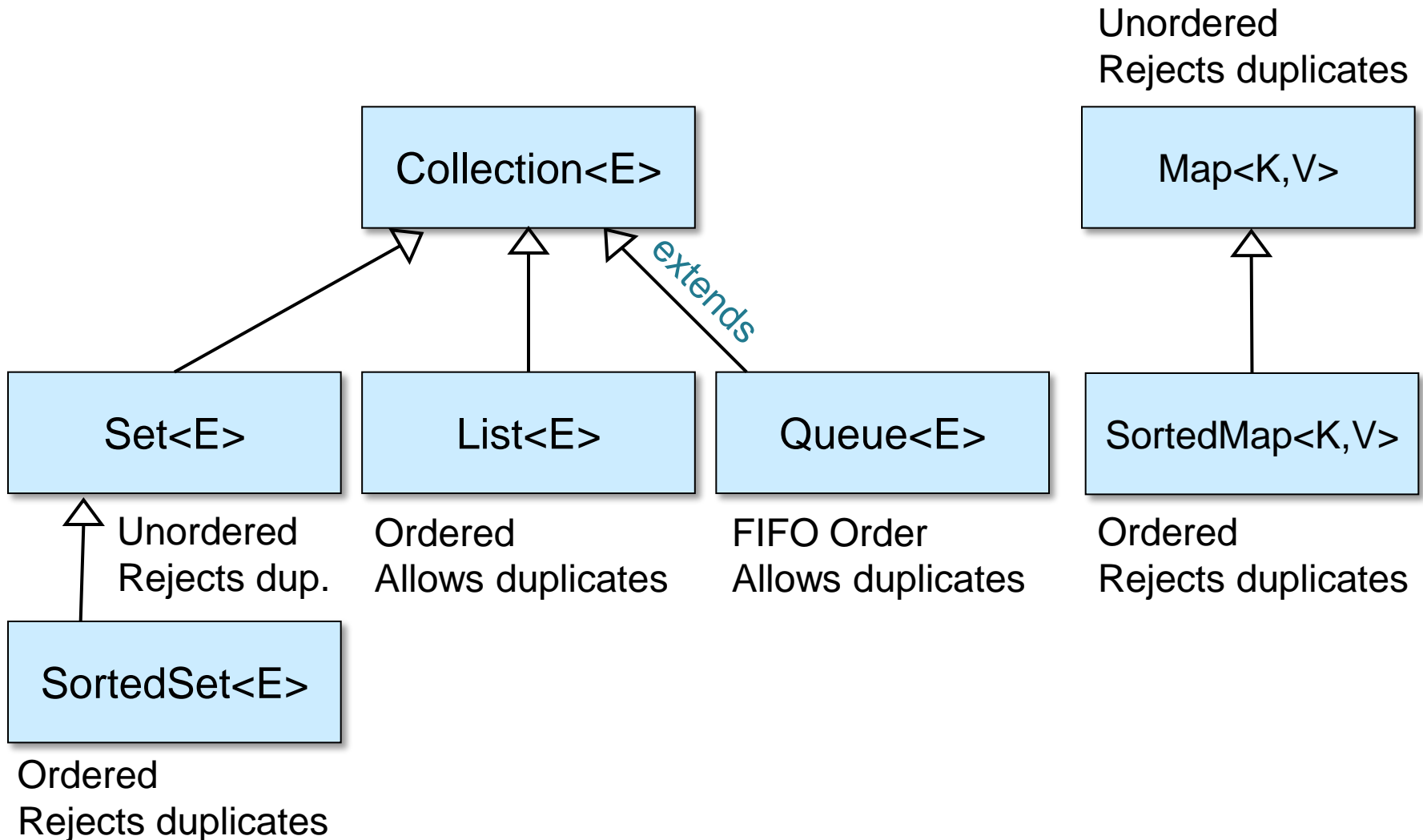
- Interface Based Design:



Online Resources

- Java 8 API Specification of the Collections Framework:
<https://docs.oracle.com/javase/8/docs/technotes/guides/collections/reference.html>
- Oracle Tutorial:
<http://docs.oracle.com/javase/tutorial/collections/>

Collection Interfaces



A Simple Example

```
Collection<String> stringCollection = ...  
Collection<Integer> integerCollection = ...  
  
stringCollection.add("Hello");  
integerCollection.add(5);  
integerCollection.add(new Integer(6));  
  
stringCollection.add(7);  
integerCollection.add("world");  
stringCollection = integerCollection;
```

A Simple Example

```
Collection<String> stringCollection = ...
```

```
Collection<Integer> integerCollection = ...
```

```
stringColle
```

```
integerColl
```

```
integerCollection.add(new Integer(6));
```

```
stringCollection.add(7);
```

```
integerCollection.add("world");
```

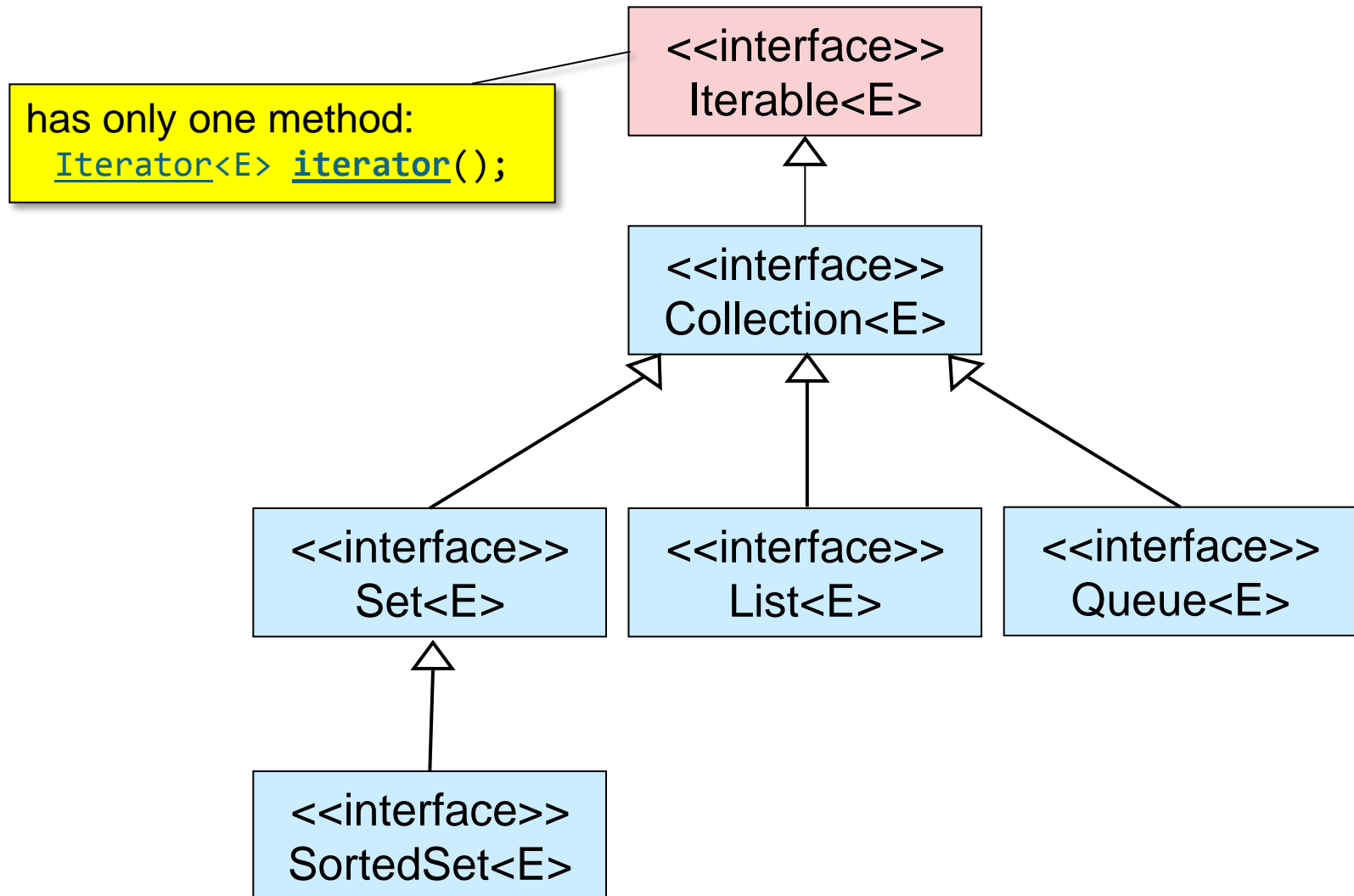
```
stringCollection = integerCollection;
```

- מצביעים ל Collection של מחרוזות ושל מספרים
- Collection אינו מחזיק טיפוסים פרימיטיביים, לכן נשתמש ב
Float ,Double ,Integer וכדומה
- נראה בהמשך אילו מחלקות מממשות מנשק זה

A Simple Example

```
Collection<String> stringCollection = ...  
Collection<Integer> integerCollection = ...  
  
stringCollection.add("Hello");  
integerCollection.add(5);  
integerCollection.add(new Integer(6));  
  
stringCollection.add(7);  
integerCollection.add("world");  
stringCollection = integerCollection;
```

Collection extends Iterable



The Iterator Interface

- Provide a way to access the elements of a collection sequentially without exposing the underlying representation
- Methods:
 - `hasNext()` - Returns true if there are more elements
 - `next()` - Returns the next element
 - `remove()` - Removes the last element returned by the iterator (optional operation)

Command and Query!

Iterating over a Collection

■ Explicitly using an Iterator

```
for (Iterator<String> iter = stringCollection.iterator();  
     iter.hasNext(); ) {  
    System.out.println(iter.next());  
}
```

■ Using foreach syntax

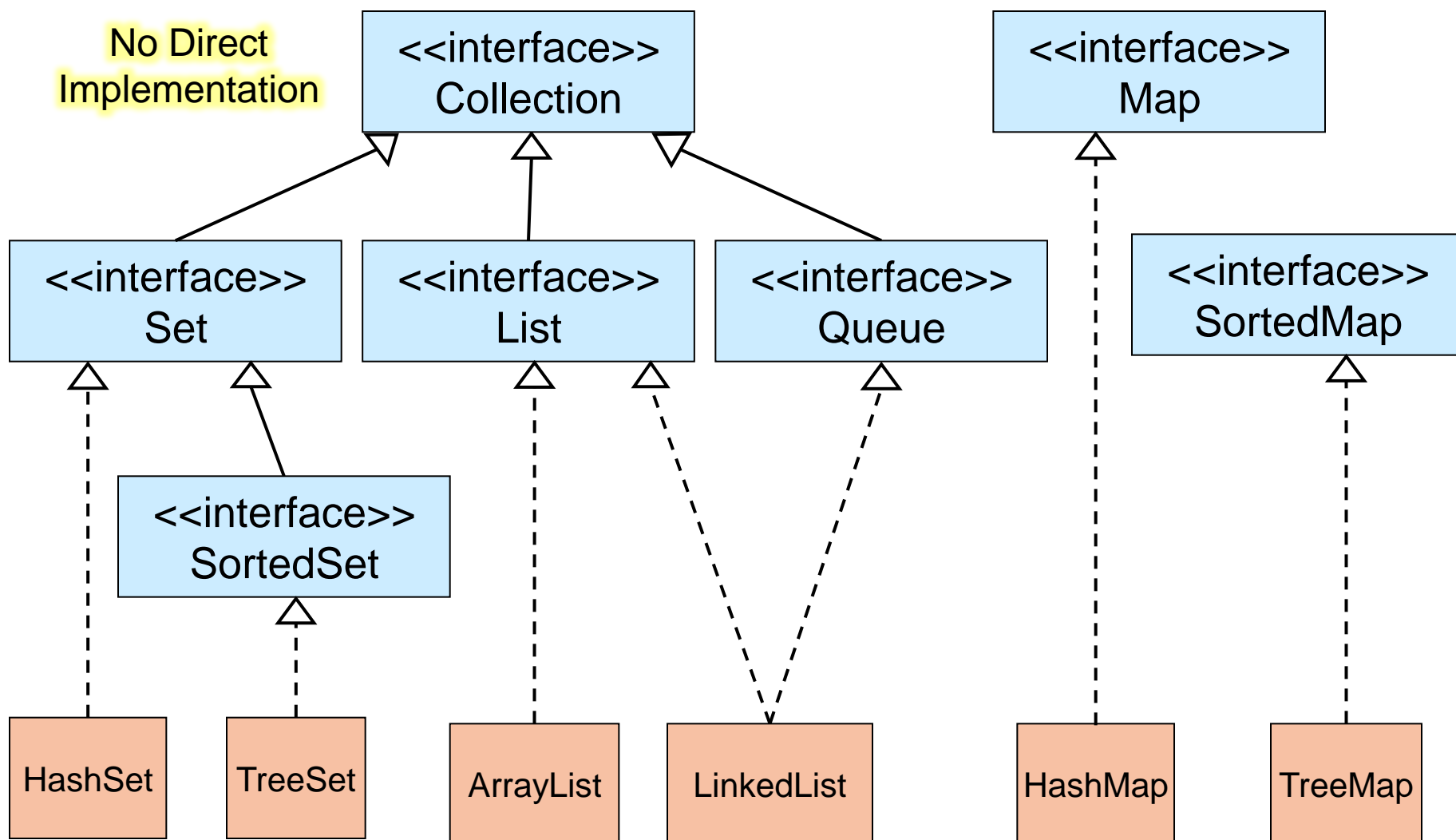
```
for (String str : stringCollection) {  
    System.out.println(str);  
}
```

General Purpose Implementations

- Class Name Convention: <Data structure> <Interface>

General Purpose Implementations		Data Structures			
		Hash Table	Resizable Array	Balanced Tree	Linked
Interfaces	Set	HashSet		TreeSet (SortedSet)	LinkedHashSet
	Queue		ArrayDeque		LinkedList
	List		ArrayList		LinkedList
	Map	HashMap		TreeMap (SortedMap)	LinkedHashMap

Adding Implementations to the Picture



Interface

List Example

```
List<Integer> list = new ArrayList<Integer>();  
list.add(3);  
list.add(1);  
list.add(new Integer(1));  
list.add(new Integer(6));  
list.remove(list.size()-1);  
System.out.println(list);
```

Implementation

List holds
Integer
references
(auto-boxing)

List allows
duplicates

Invokes
List.toString()

Output:
[3, 1, 1]

Insertion
order is kept

remove() can get
index or *reference*
as argument

Set Example

```
Set<Integer> set = new HashSet<Integer>();  
set.add(3);  
set.add(1);  
set.add(new Integer(1));  
set.add(new Integer(6));  
set.remove(6);  
System.out.println(set);
```

A set does not allow duplicates. It **does not** contain:

- two references to the same object
- two references to null
- references to two objects a and b such that a.equals(b)

remove() can get only *reference* as argument

Output: [1, 3] or [3, 1]

Insertion order is not guaranteed (unlike LinkedHashSet)

Queue Example

```
Queue<Integer> queue = new LinkedList<Integer>();  
queue.add(3);  
queue.add(1);  
queue.add(new Integer(1));  
queue.add(new Integer(6));  
queue.remove();  
System.out.println(queue);
```

Output: [1, 1, 6]

FIFO order

Elements are added
at the end of the
queue

remove() may
have no argument –
head is removed

Map Example

```
Map<String, String> map = new HashMap<String,  
    String>();
```

```
map.put("Dan", "03-9516743");
```

```
map.put("Rita", "09-5076452");
```

```
map.put("Leo", "08-5530098");
```

```
map.put("Rita", "06-8201124");
```

```
System.out.println(map);
```

Output:

```
{Leo=08-5530098, Dan=03-9516743, Rita=06-8201124}
```

No key duplicates

Unordered

Keys (names)	Values (phone numbers)
Dan	03-9516743
Rita	06-8201124
Leo	08-5530098

LinkedHashMap Example

```
Map<String, String> map = new LinkedHashMap<String, String>();  
map.put("Dan", "03-9516743");  
map.put("Rita", "09-5076452");  
map.put("Leo", "08-5530098");  
map.put("Rita", "06-8201124");  
System.out.println(map);
```

Insertion order (first time
key insertion)

Output:

```
{Dan=03-9516743, Rita=06-8201124, Leo=08-5530098}
```

Keys (names)	Values (phone numbers)
Dan	03-9516743
Rita	06-8201124
Leo	08-5530098

SortedMap Example

```
SortedMap <String,String> map = new TreeMap<String,String> ();  
map.put("Dan", "03-9516743");  
map.put("Rita", "09-5076452");  
map.put("Leo", "08-5530098");  
map.put("Rita", "06-8201124");  
System.out.println(map);
```

lexicographic order

Output:

```
{Dan=03-9516743, Leo=08-5530098, Rita=06-8201124}
```

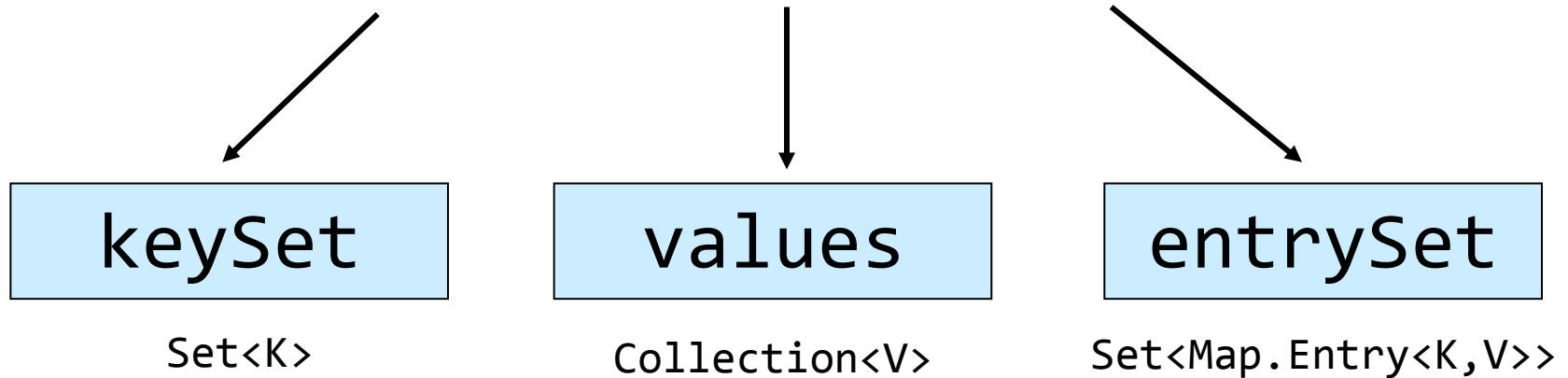
Keys (names)	Values (phone numbers)
Dan	03-9516743
Leo	08-5530098
Rita	06-8201124

Map Collection Views



A Map is not Iterable!

Three views of a `Map<K, V>` as a collection



The Set of key-value pairs
(implement `Map.Entry`)

Iterating Over the Keys of a Map

```
Map<String,String> map = new HashMap<String,String> ();  
map.put("Dan", "03-9516743");  
map.put("Rita", "09-5076452");  
map.put("Leo", "08-5530098");  
map.put("Rita", "06-8201124");
```

```
for (String key : map.keySet()) {  
    System.out.println(key);  
}
```

Output: Leo
 Dan
 Rita



Iterating Over the Key-Value Pairs of a Map

```
Map<String,String> map = new HashMap<String,String> ();
map.put("Dan", "03-9516743");
map.put("Rita", "09-5076452");
map.put("Leo", "08-5530098");
map.put("Rita", "06-8201124");

for (Map.Entry<String,String> entry: map.entrySet()) {
    System.out.println(entry.getKey() + ": " +
        entry.getValue());
}
```

Output: Leo: 08-5530098
 Dan: 03-9516743
 Rita: 06-8201124

Collection Algorithms

- Defined in the Collections class
- Main algorithms:
 - sort
 - binarySearch
 - reverse
 - shuffle
 - min
 - max

Sorting and Comparing

- Sort a List `l` by `Collections.sort(l);`
- If the list consists of String objects it will be sorted in lexicographic order. Why?

- String implements the interface Comparable<String>:

```
public interface Comparable<T> {  
    public int compareTo(T other);  
}
```

- Returns

┌	a negative value	if this < other
	zero	if this.equals(other)
	a positive value	if this > other
- Error when sorting a list whose elements
 - do not implement Comparable or
 - are not *mutually comparable*.
- User defined comparator
 - `Collections.sort(List, Comparator);`

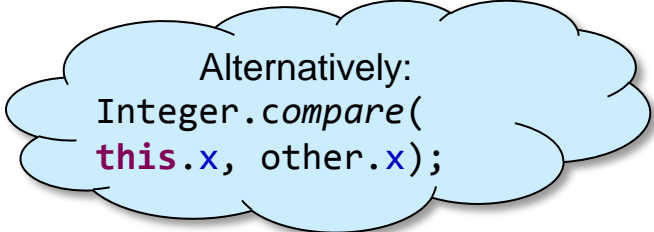
Comparable and Comparator Example

- Write the class Point that represents a point in the plane
- How to sort List<Point>?
 - By the x value? By the y value?
Distance from the origin?...
- Two options --
 - Make Point implement Comparable<Point>, and use Collections.sort
 - Write a class that implements Comparator<Point>, and pass it as an argument to Collections.sort.
 - **Don't: write a sorting algorithm yourselves!**
- **Recommended Tutorial:**
<http://docs.oracle.com/javase/tutorial/collections/interfaces/order.html>

```
public class Point {  
    private int x;  
    private int y;  
    ...  
}
```


Implementing Comparable

```
public class Point implements Comparable<Point>{  
    ...  
    public int compareTo(Point other) {  
        //comparison by the x axis  
        return this.x - other.x;  
    }  
}
```



Alternatively:
`Integer.compare(
 this.x, other.x);`


- The program:

```
List<Point> pointList = new LinkedList<Point>();  
pointList.add(new Point(1, 3));  
pointList.add(new Point(0, 6));  
Collections.sort(pointList);  
System.out.println(pointList);
```

- Output: [(0,6), (1,3)]

Writing a Comparator

```
public class YAxisPointComparator implements Comparator<Point> {  
    public int compare(Point p1, Point p2) {  
        return p1.getY() - p2.getY();  
    }  
}
```



- The program:

```
List<Point> pointList = new LinkedList<Point>();  
pointList.add(new Point(1, 3));  
pointList.add(new Point(0, 6));  
Collections.sort(pointList, new YAxisPointComparator());  
System.out.println(pointList);
```

- The output: [(1,3), (0,6)]
- Useful for sorting existing classes (e.g., String)

Best Practice <with generics>

- Specify an element type only when a collection is instantiated:

```
Set<String> s = new HashSet<String>();
```

Interface

Implementation

Works, but...

```
public void foo(HashSet<String> s){...}
```

```
public void foo(Set<String> s) {...}
```

```
s.add() invokes HashSet.add()
```

Better!

polymorphism

Diamond Notation

```
Set<String> s = new HashSet<String>();
```

```
→ Set<String> s = new HashSet<>();
```

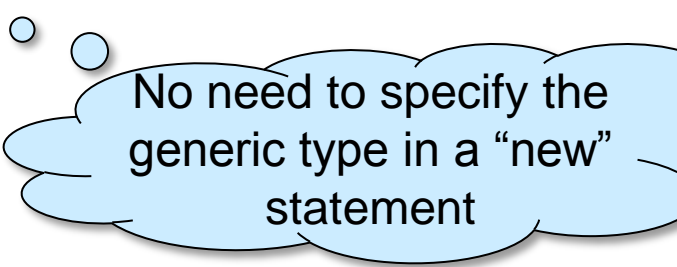
```
Map<String, List<String>> myMap =  
    new HashMap<String, List<String>>();
```

```
→ Map<String, List<String>> myMap = new HashMap<>();
```

Not the same as:

```
Map<String, List<String>> myMap = new HashMap();
```

(Compilation warning)



No need to specify the generic type in a “new” statement