

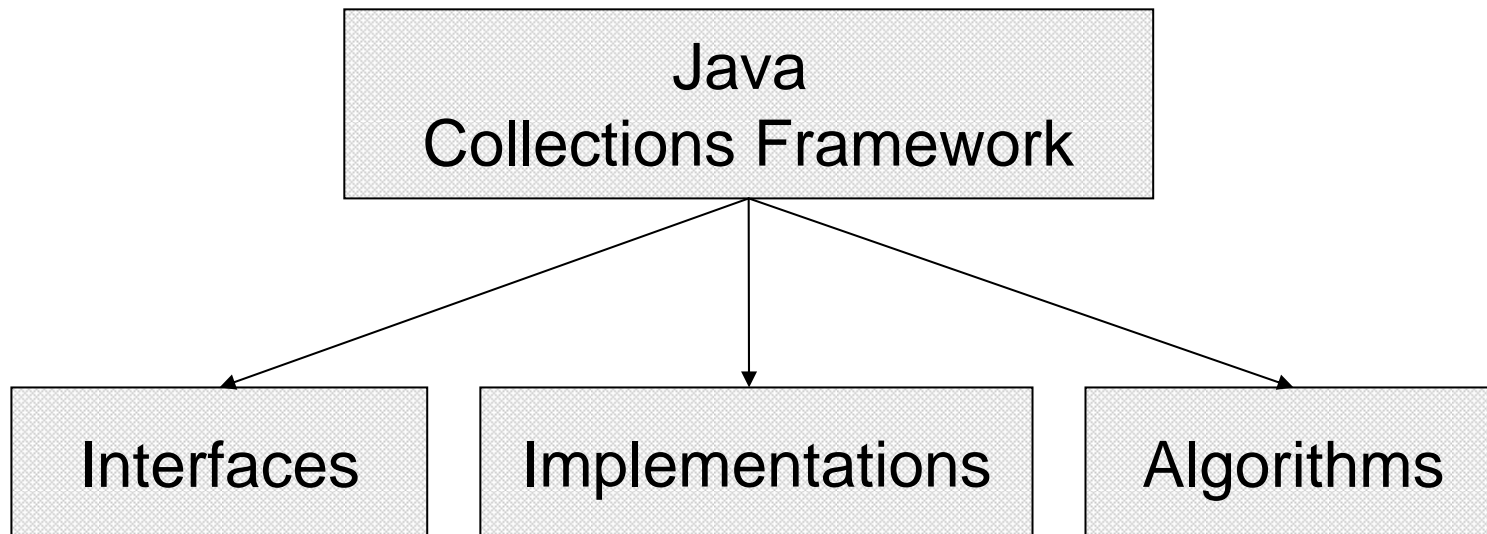


Software 1 with Java

Recitation No. 6
(Collections)

Java Collections Framework

- **Collection:** a group of elements
- Interface Based Design:



Online Resources

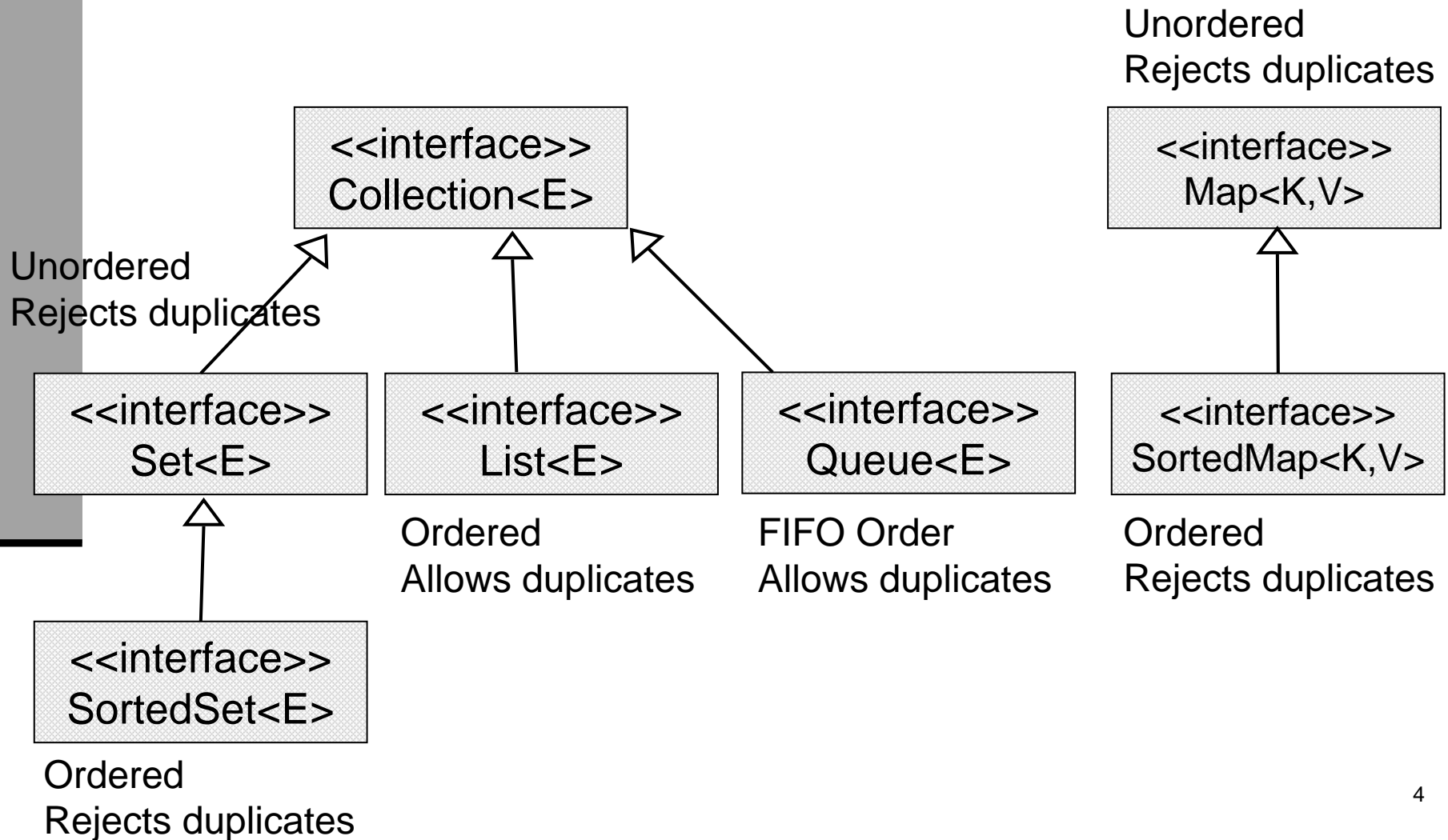
- Java 5 API Specification:

<http://java.sun.com/j2se/1.5.0/docs/api/index.html>

- Sun Tutorial:

<http://java.sun.com/docs/books/tutorial/collections/>

Collection Interfaces

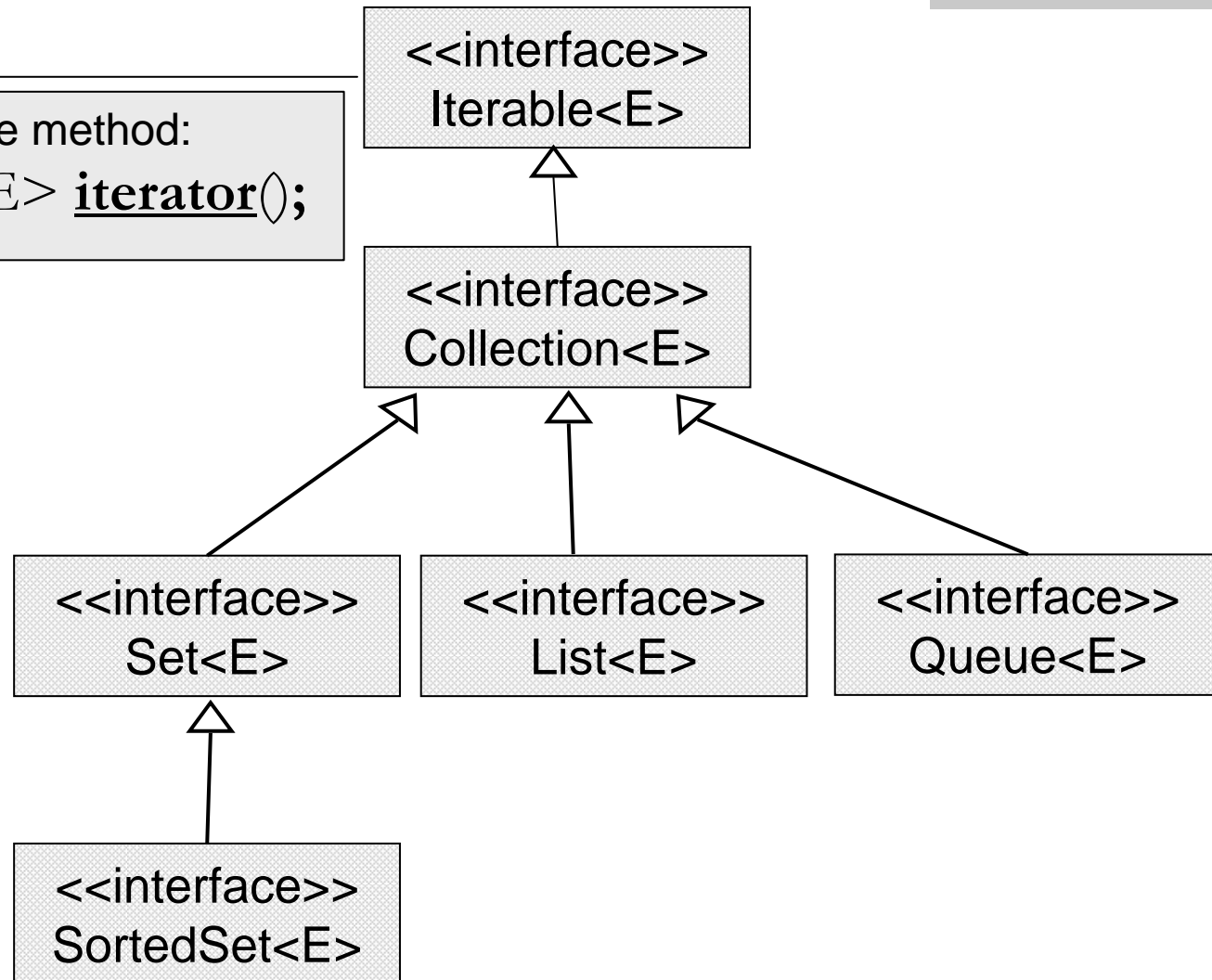


The Collection Interface

- Doesn't hold primitives
 - Use wrapper classes
- Before Java5:
 - Not type safe
 - Need to use casting
- Since Java5: the type of the elements in the collection can be specified (using generics)
 - → collections are type-safe

Collection extends Iterable

has only one method:
Iterator<E> iterator();



The Iterator Interface

- Provide a way to access the elements of a collection sequentially without exposing its 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)

Iterating over a Collection

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


Collection Implementations

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

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

Best Practice <with generics>

- Specify an implementation only when a collection is constructed:

- `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

How it was done before Java 5

- Specify an implementation only when a collection is constructed:

- `Set s = new HashSet ();`
Interface Implementation

Works, but...

- `public void foo(HashSet s) {...}`
- `public void foo(Set s) {...}`
- `s.add()` invokes `HashSet.add()`

Better!

polymorphism

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()

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

Output:
[3, 1, 1]

Insertion
order is kept

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]

Insertion order is
not guaranteed

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);
```

Elements are added
to the tail of the
queue

remove() may
have no argument –
head is removed

Output: [1, 1, 6]

FIFO order

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);
```

No duplicates

Unordered

Output:

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

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

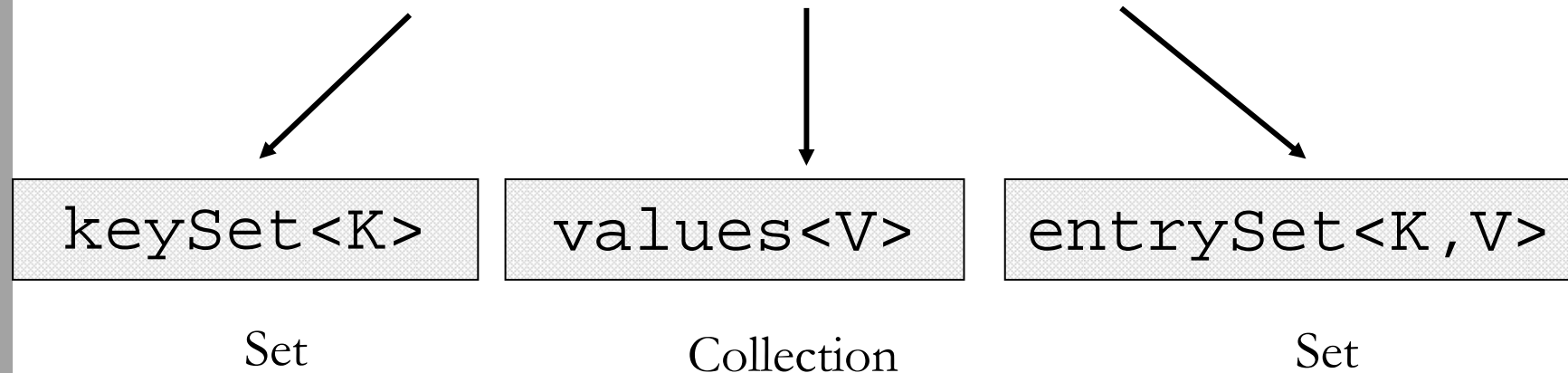
Output:

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

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

Map Collection Views

Three views of a `Map` 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 (Iterator<String> iter= map.keySet().iterator(); iter.hasNext(); ) {
    System.out.println(iter.next());
}
```

Output:

- Leo
- Dan
- Rita

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 (Iterator<Map.Entry<String,String>> iter= map.entrySet().iterator();
     iter.hasNext(); ) {

    Map.Entry<String,String> entry = iter.next();
    System.out.println(entry.getKey() + ": " + entry.getValue());
}
```

Output:

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

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

```
import java.util.*;
```

import the package of
List, Collections
and Arrays

```
public class Sort {  
    public static void main(String args[]) {  
        List<String> list = Arrays.asList(args);  
        Collections.sort(list);  
        System.out.println(list);  
    }  
}
```

returns a List-view of
its array argument.

Arguments: A C D B

Output: [A, B, C, D]

lexicographic
order

Sorting (cont.)

- 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 `Comparable<String>`:

```
public interface Comparable<T> {  
    public int compareTo(T o);  
}
```
- Exception when sorting a list whose elements
 - do not implement `Comparable` or
 - are not *mutually comparable*.