Object-Oriented Programming with Java

Recitation No. 2

Oranit Dror

The String Class

- Represents a character string (e.g. "Hi")
- Explicit constructor:

```
String quote = "Hello World";
```

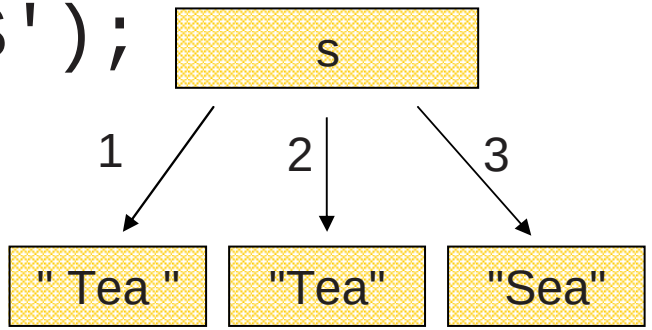
string literal

- All string literals are `String` instances
- `Object` has a `toString()` method
- For more details: JDK Documentation

String Immutability

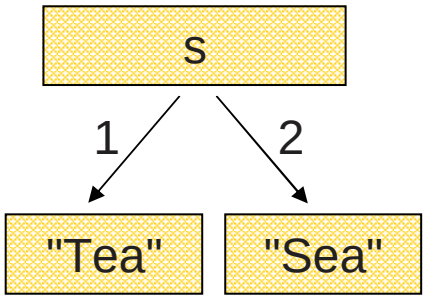
- Strings are constants

```
String s = " Tea ";
s = s.trim();
s = s.replace('T', 'S');
```



- A string reference may be set:

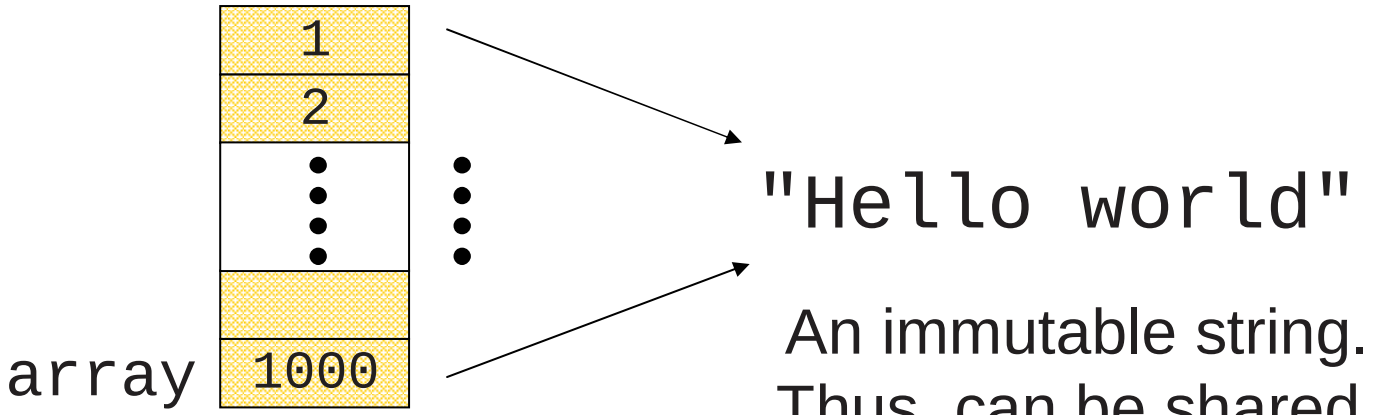
```
String s = "Tea";
s = "Sea";
```



String Interning

- Avoids duplicate strings

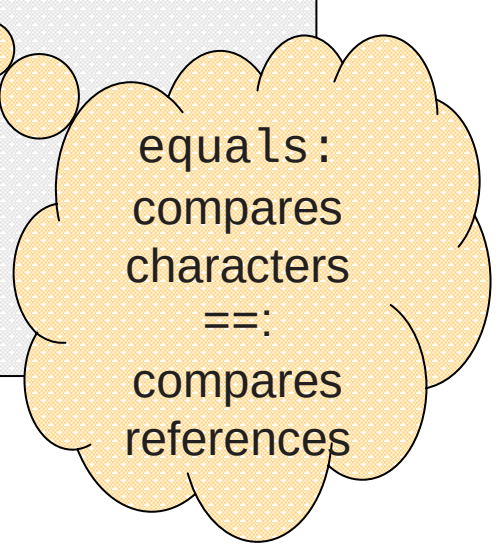
```
String[] array = new String[1000];
for (int i=0 ; i<1000 ; i++) {
  array[i] = "Hello world";
}
```



String Interning (cont.)

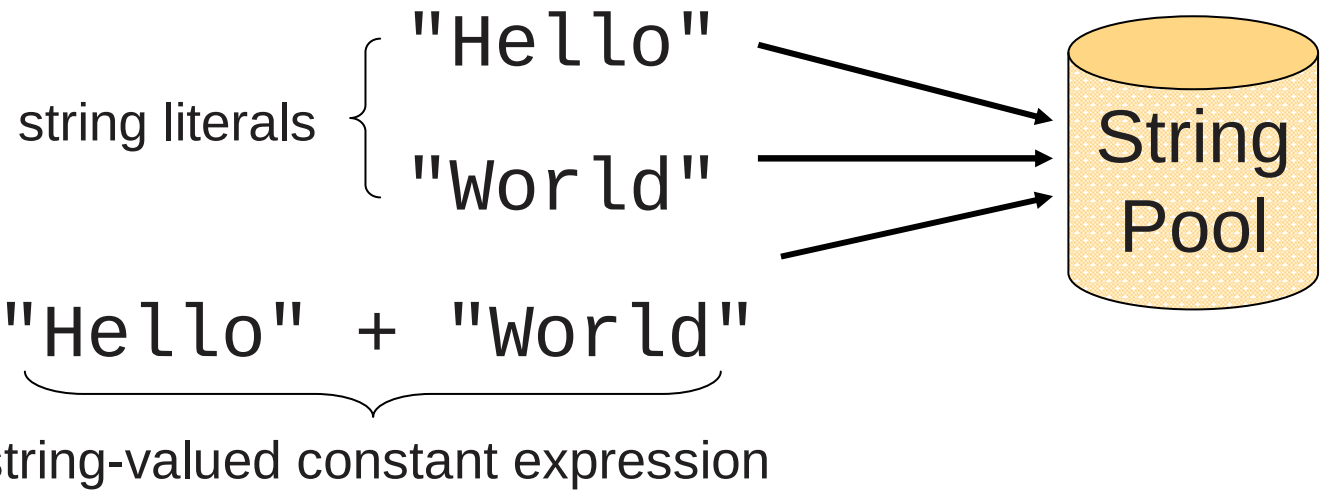
- The `String` class has a static private **pool** of internal strings.
- `myString.intern()` implementation:

```
if ∃s ∈ pool : myString.equals(s) == true
        return s;
else
        add myString to the pool
        return myString;
```



String Interning (cont.)

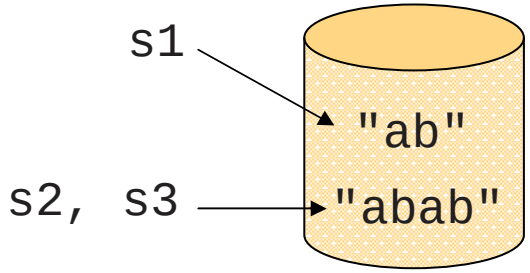
- All string literals and string-valued constant expressions are interned.



String Interning (cont.)

If:

```
String s1 = "ab";
String s2 = "ab" + "ab";
String s3 = "aba" + "b";
String s4 = s1 + s1;
String s5 = s1 + s1;
String s6 = s1 + "ab";
```



Then:

```
s4.equals(s2) is true
(s4 == s2) is false
(s4 == s5) is false
(s2 == s3) is true
(s2 == s6) is false
(s4.intern() == s2) is true
(s4.intern() == s5.intern()) is true
```

s4 → "abab"

s5 → "abab"

s6 → "abab"

String Constructors

- Use explicit constructor:

```
String s = "Hello";
```

(string literals are interned)

Instead of:

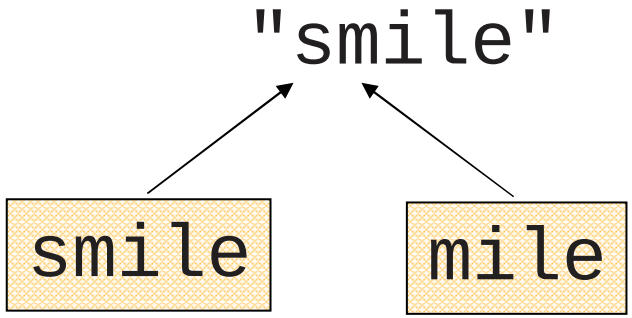
```
String s = new String("Hello");
```

(causes extra memory allocation)

Substrings

- Substrings are created without copying
- `String.substring()` is very efficient

```
String smile = "smile";
String mile = "smile".substring(1,4);
```



The StringBuffer Class

- Represents a mutable character string
- Main methods: `append()` & `insert()`
  - accept data of any type
  - If: `sb = new StringBuffer("123")`

    Then: `sb.append(4)` is equivalent to `sb.insert(sb.length(), 4).`

    Both yields `"1234"`

The Concatenation Operator (+)

- String conversion and concatenation:
  - `"Hello " + "World"` is `"Hello World"`
  - `"19" + 8 + 9` is `"1989"`
- Conversion by `toString()`
- Concatenation by `StringBuffer`
- `String x = "19" + 8 + 9;`

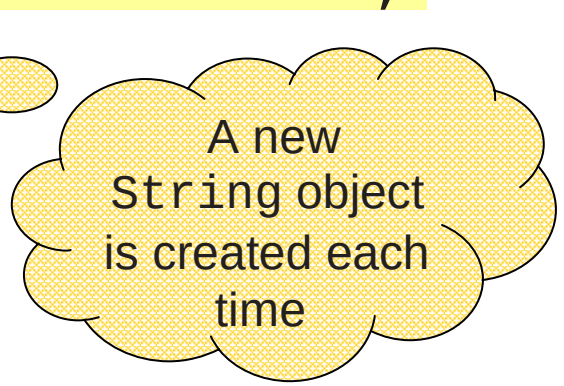
  is compiled to the equivalent of:

```
String x = new StringBuffer().append("19").
append(8).append(9).toString();
```

StringBuffer vs. String

- Inefficient version using String:

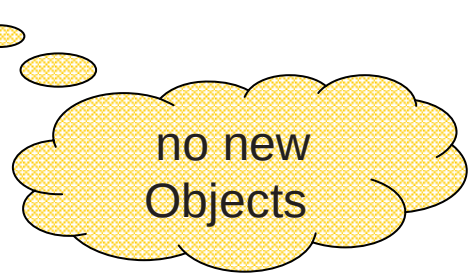
```
public static String
duplicate(String s, int times) {
    String result = s;
    for (int i=1; i<times; i++) {
        result = result + s;
    }
    return result;
}
```



StringBuffer vs. String (cont.)

- More efficient version with StringBuffer:

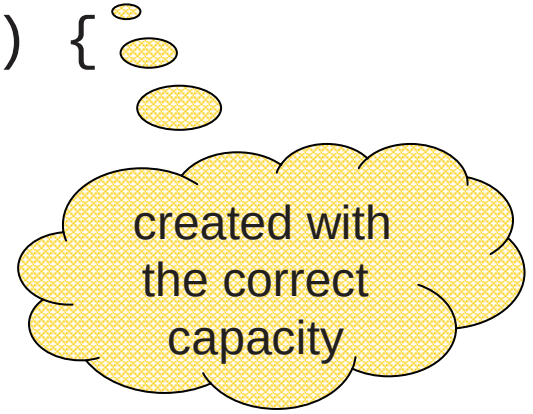
```
public static String
duplicate(String s, int times) {
    StringBuffer result = new StringBuffer(s);
    for (int i=1; i<times; i++) {
        result.append(s);
    }
    return result.toString();
}
```



StringBuffer vs. String (cont.)

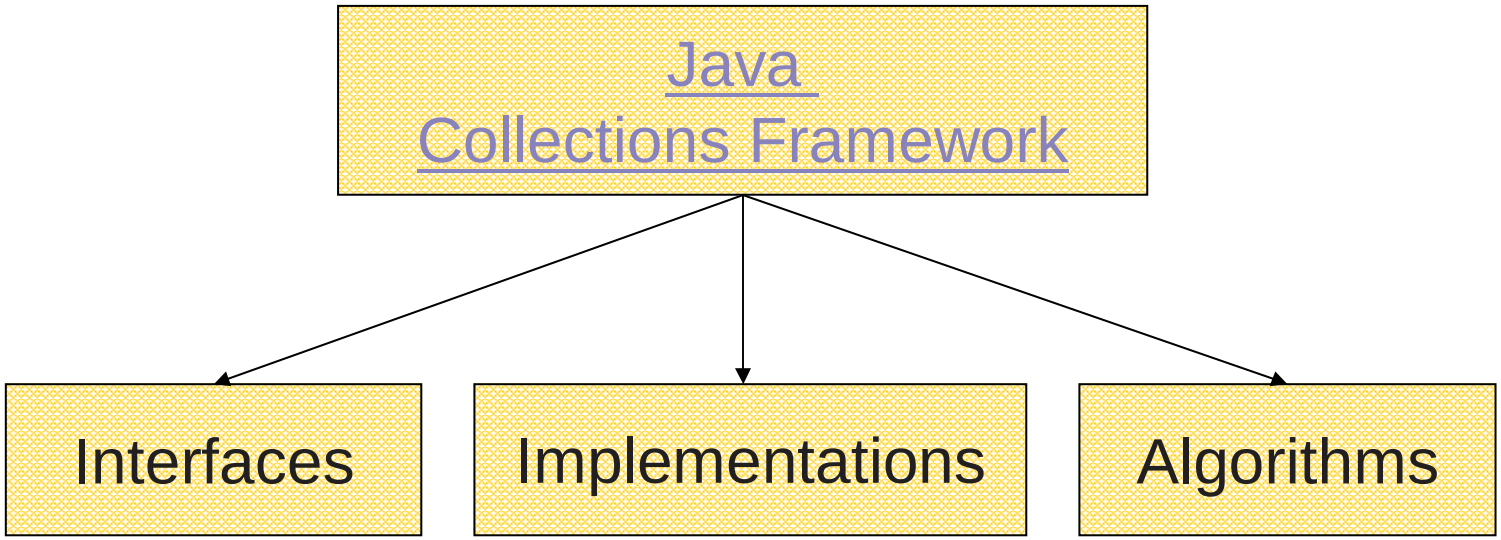
- Much more efficient version:

```
public static String
duplicate(String s, int times) {
    StringBuffer result = new
          StringBuffer(s.length() * times);
    for (int i=0; i<times; i++) {
          result.append(s);
    }
    return result.toString();
}
```

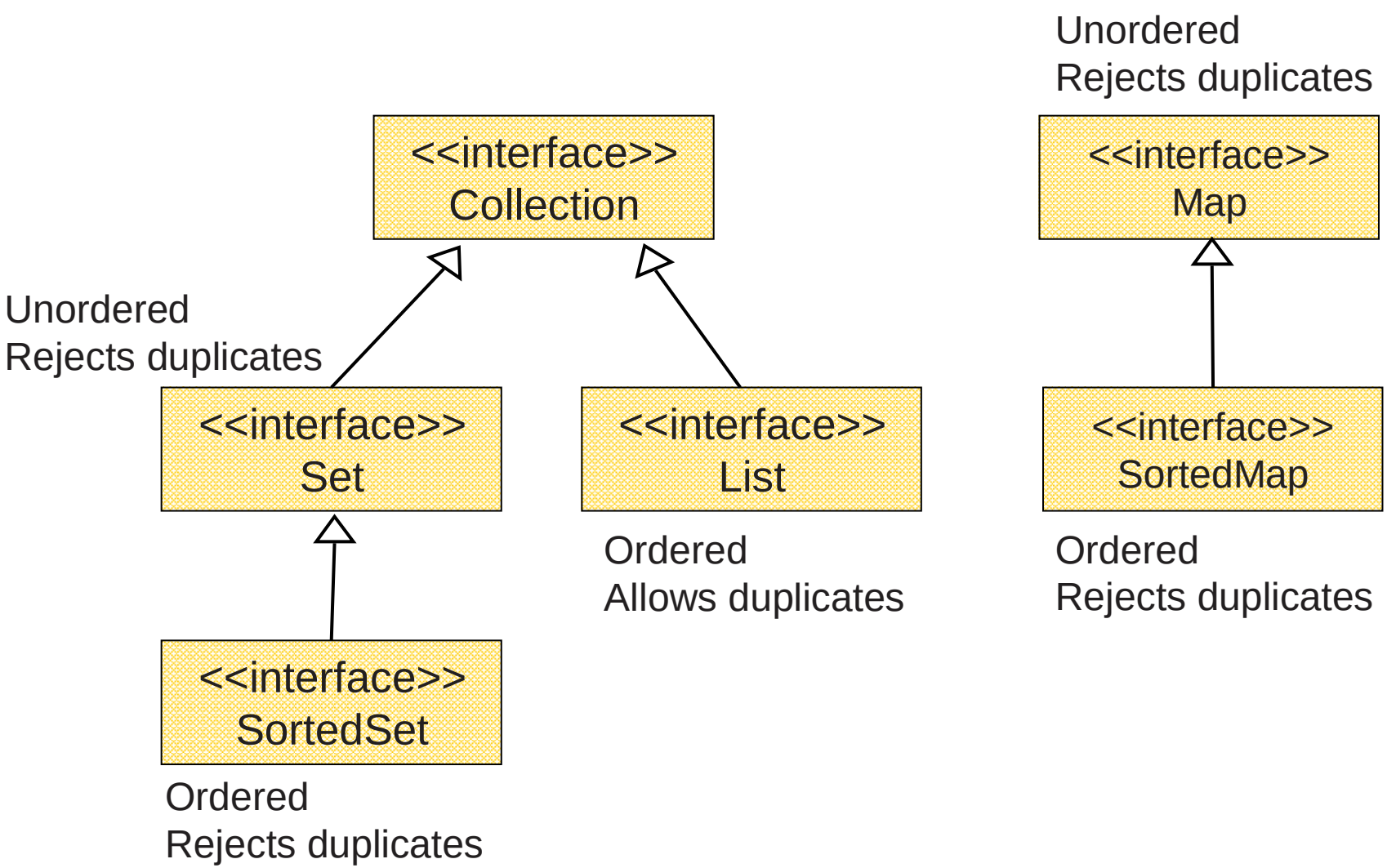


Java Collections Framework

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



Collection Interfaces



The `Collection` Interface

- Hold any `Object` references (use casting)
- Doesn't hold primitives (use wrapper classes)
- Assume `c` is a `Collection`, then:

```
c.add("Hello World");
c.add(new Integer(1));
```

- In Java5 type-safe collections are defined
- Iterating over `Collection` elements:

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

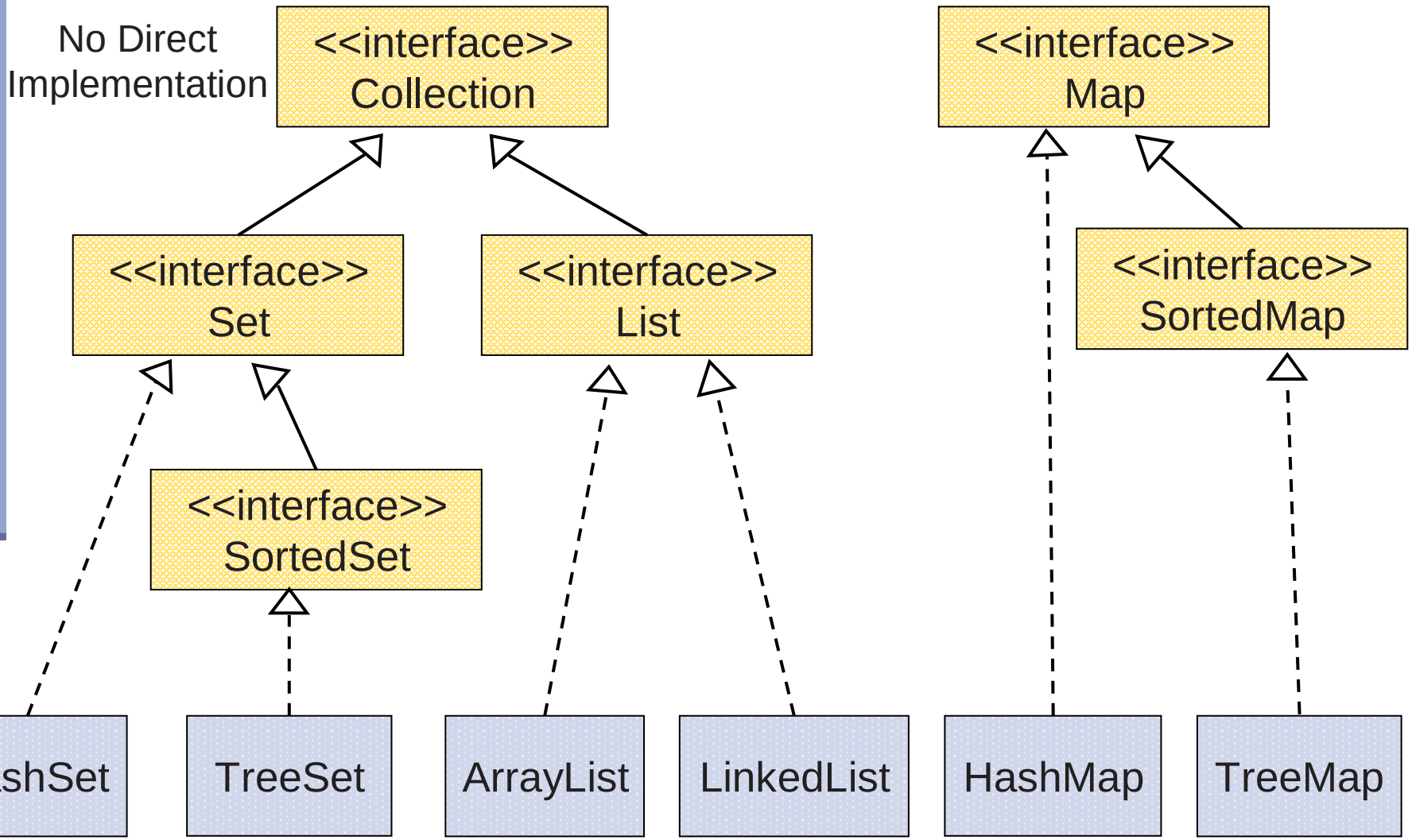
Collection Implementations
General Purpose Implementations

- Class Name Convention:

  <Data structure> <Interface>

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

General Purpose Implementations (cont.)



Best Practice

- Specify an implementation only when a collection is constructed:
  - `Set s= new HashSet();`

    Interface — Implementation
  - `public void foo(HashSet s) {…}` ✗

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

List Example

```
List list = new ArrayList();
list.add("A");
list.add("B");
list.add(new Integer(1));
list.add(new Integer(1));
System.out.println(list);
```

Interface

Implementation

`List` holds `Object` references (casting)

Invokes `List.toString()`

List allows duplicates

Insertion order is kept

Output: `[A, B, 1, 1]`

Set Example

```
Set set = new HashSet();
set.add("A");
set.add("B");
set.add(new Integer(1));
set.add(new Integer(1));
System.out.println(set);
```

Set rejects duplicates (tested by equals)

Output: [A, 1, B]

Insertion order is not guaranteed

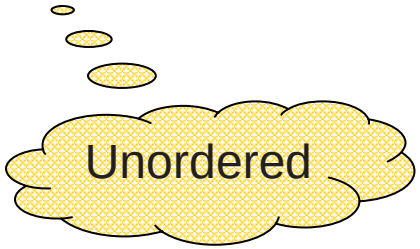
Map Example

```
Map map = new HashMap();
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:

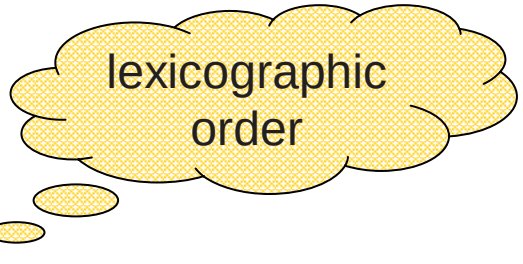
```
{Leo=08-5530098, Dan=03-9516743, Rita=06-8201124}
```

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



SortedMap Example

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



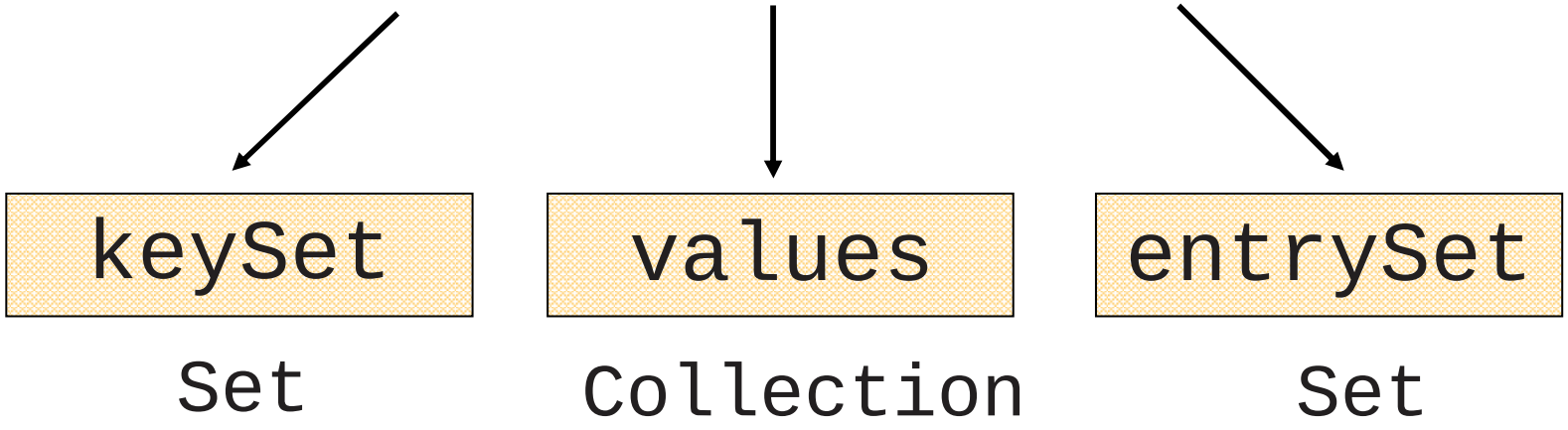
Output:

```
{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`)

Map Collection Views (cont.)

- provide the *only* means to iterate over a `Map`
- Iterating over the keys in a `Map`:

```
for (Iterator iter=map.keySet().iterator();
     iter.hasNext(); ) {
    System.out.println(iter.next());
}
```

- Iterating over the key-value pairs in a `Map`:

```
for (Iterator iter=map.entrySet().iterator();
     iter.hasNext(); ) {
    Map.Entry e = (Map.Entry) iter.next();
    System.out.println(e.getKey() + ": " +
                       e.getValue());
}
```

Historical Collection Implementations

- The primary legacy classes are:
  - Vector
  - Hashtable
- In new code use:
  - ArrayList instead of Vector
  - HashMap instead of Hashtable

Collection Algorithms

- Defined in the Collections class
- Main algorithms:
  - `sort(List list)`
  - `binarySearch(List list, Object key)`
  - `reverse(List list)`
  - `shuffle(List list)`
  - `min(Collection collection)`
  - `max(Collection collection)`

Sorting

```
import java.util.*;

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

import the package of `List`, `Collections` and `Arrays`

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:

```
public interface Comparable {
   public int compareTo(Object o);
}
```

- Exception when sorting a list whose elements
  - do not implement `Comparable` or
  - are not *mutually comparable*.

Sorting (cont.)

- To sort a list whose elements are not mutually comparable use:

```
Collections.sort(List list,
                 Comparator c)
```

where:

```
public interface Comparator {
 int compare(Object o1, Object o2);
 }
```