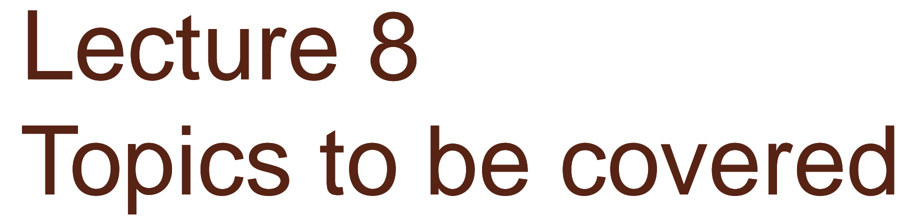
Course Name:
Advanced Java



- Collections

Introduction

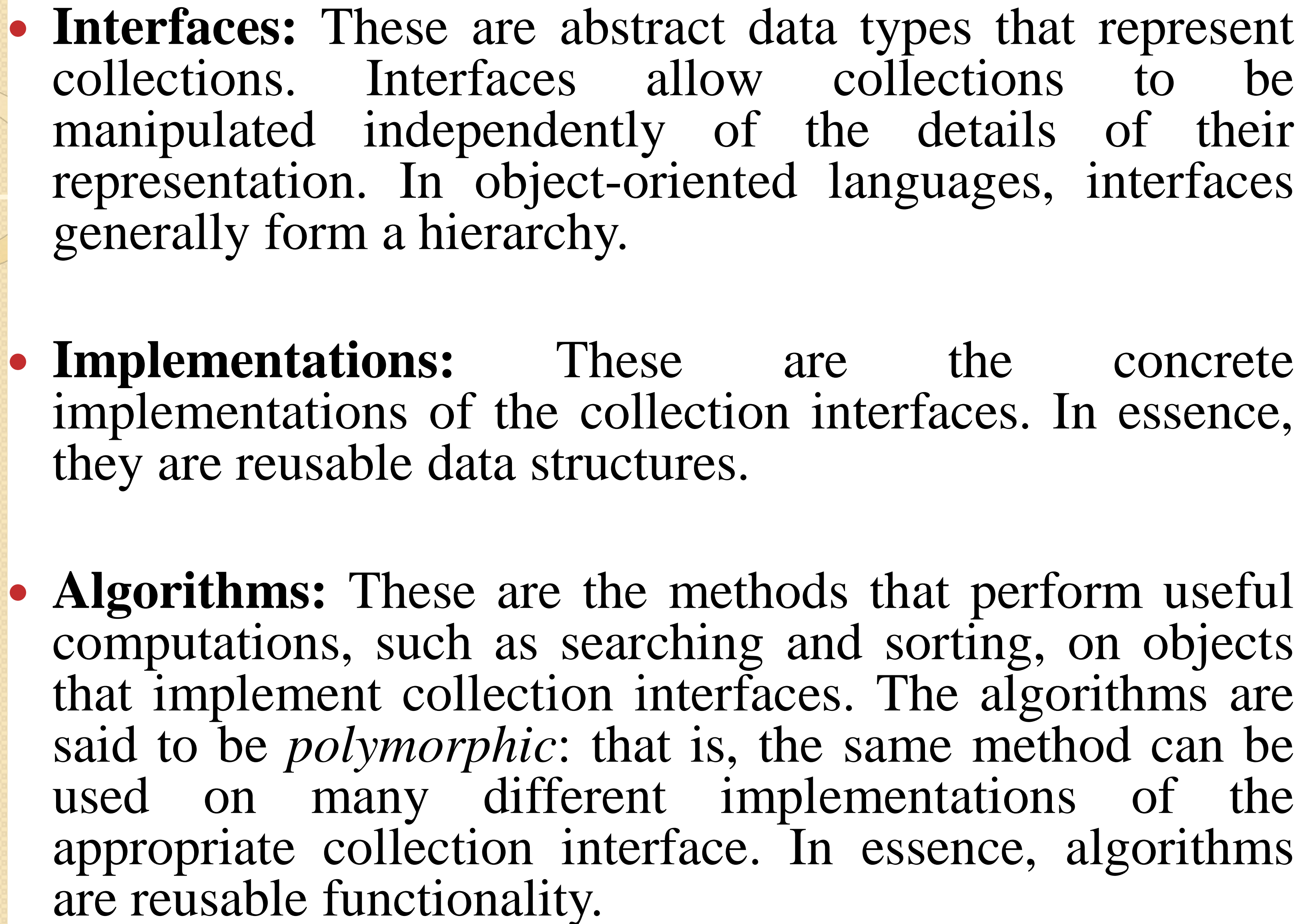
- A collection, sometimes called a container, is simply an object that groups multiple elements into a single unit.
- Collections are used to store, retrieve, manipulate, and communicate aggregate data.
- Typically, they represent data items that form a natural group, such as a poker hand (a collection of cards), a mail folder (a collection of letters), or a telephone directory (a mapping of names to phone numbers).
- Collections are contained in the java.util package.
- These are designed to provide high-performance to the processes.

Collection Framework

- Java language defines a collections framework as "a unified architecture for representing and manipulating collections, allowing them to be manipulated independent of the details of their representation."

All collections frameworks contain the following:

1. Interfaces
2. Implementations
3. Algorithms



- **Interfaces:** These are abstract data types that represent collections. Interfaces allow collections to be manipulated independently of the details of their representation. In object-oriented languages, interfaces generally form a hierarchy.

- **Implementations:** These are the concrete implementations of the collection interfaces. In essence, they are reusable data structures.

- **Algorithms:** These are the methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces. The algorithms are said to be *polymorphic*: that is, the same method can be used on many different implementations of the appropriate collection interface. In essence, algorithms are reusable functionality.

Java arrays have limitations.

- They cannot dynamically shrink and grow.
- Implementing efficient, complex data structures from scratch would be difficult.

The Java Collections Framework is a set of classes and interfaces implementing complex collection data structures.

- A *collection* is an object that represents a group of objects.

The Java Collections Framework provides many benefits:

- Reduces programming effort (already there)
- Increases performance (tested and optimized)
- Part of the core API (available, easy to learn)
- Promotes software reuse (standard interface)
- Easy to design APIs based on generic collections

Overview of Collection

- A *collection* is a group of data manipulate as a single object. Corresponds to a *bag.*
- Like C++'s Standard Template Library (STL)
- Can grow as necessary.
- Contain only **Objects** (reference types).

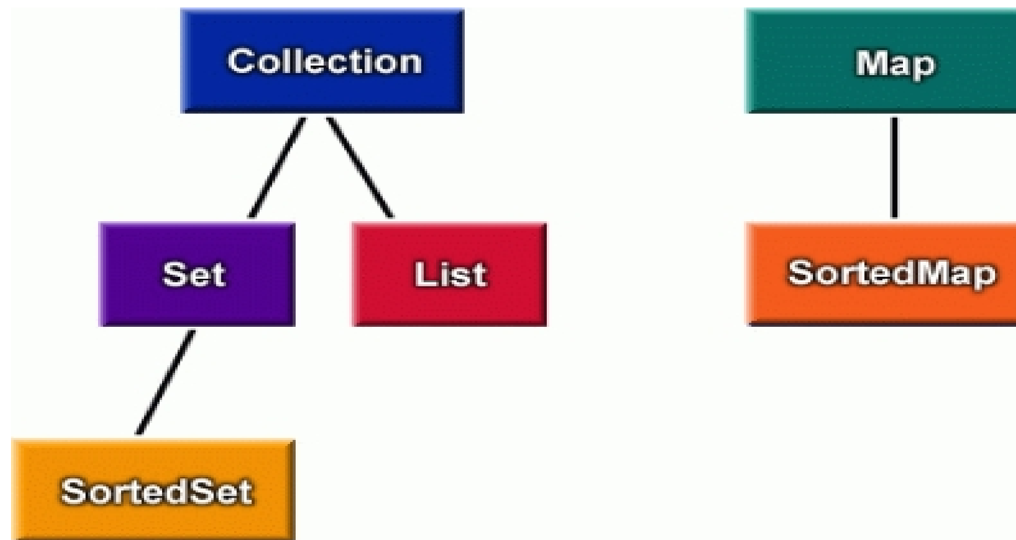
Collection Interfaces

- Collections are primarily defined through a set of interfaces.

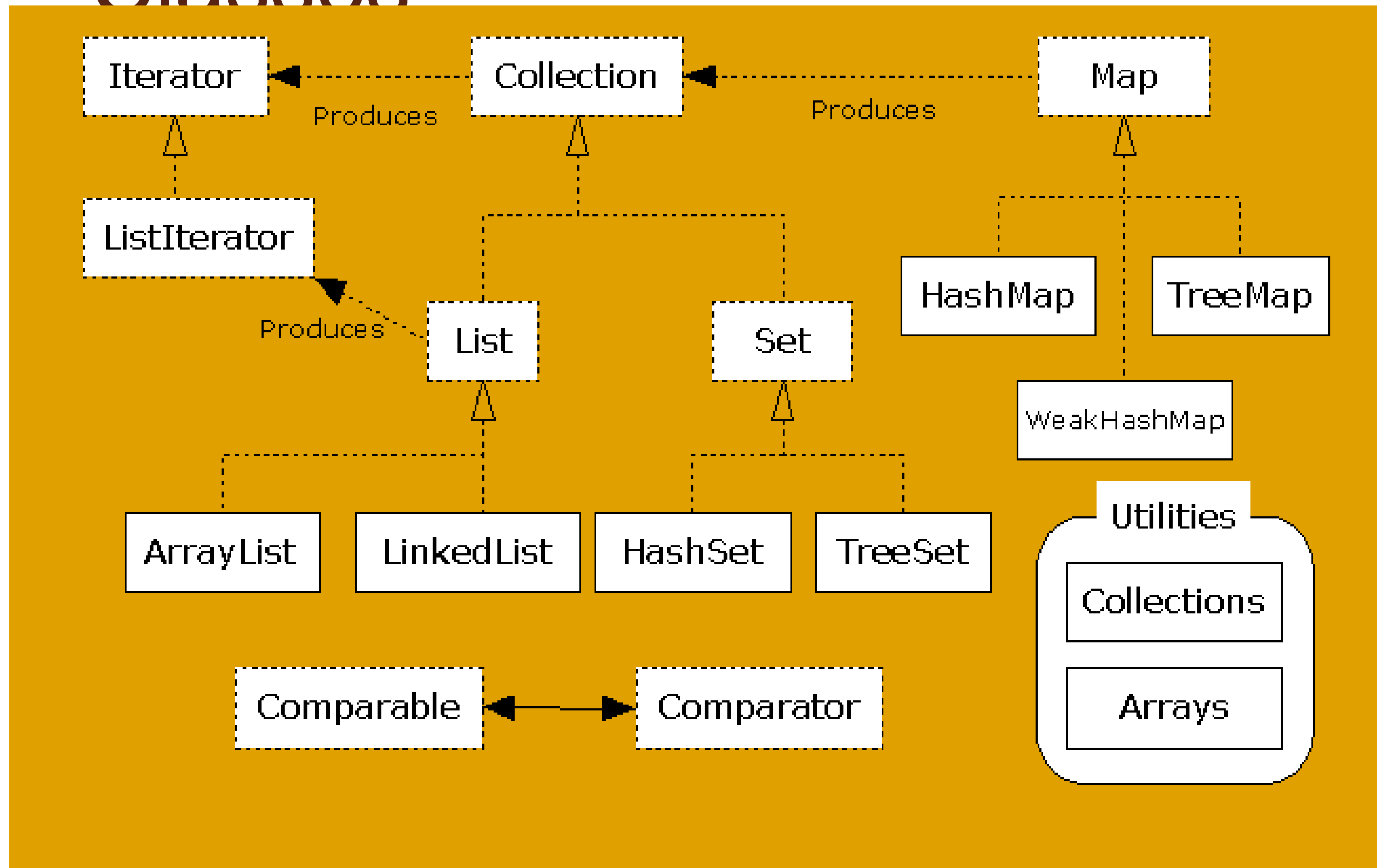
  Supported by a set of classes that implement the interfaces

- Interfaces are used of flexibility reasons

  . It is easy to change or replace the underlying collection class with another (more efficient) class that implements the same interface.



Collection Interfaces and Classes



The Set Interface

- Corresponds to the mathematical definition of a set (no duplicates are allowed).
- Compared to the **Collection** interface

  1. Interface is identical.

  2. Every constructor must create a collection without duplicates.

  3. The operation **add** cannot add an element already in the set.

  4. The method call **set1.equals(set2)** works at follows

  set1 $\subseteq$ set2, and set2 $\subseteq$ set1

Set Idioms

- set1 U set2

  set1.addAll(set2)
- set1 ∩ set2

  set1.retainAll(set2)
- set1 - set2

  set1.removeAll(set2)

HashSet and TreeSet Classes

- **HashSet** and **TreeSet** implement the interface **Set**.
- **HashSet**
    1. Implemented using a hash table.
    2. No ordering of elements.
    3. **add**, **remove**, and **contains** methods
- **TreeSet**
    1. Implemented using a tree structure.
    2. Guarantees ordering of elements.
    3. **add**, **remove**, and **contains** methods

HashSet, Example

```
import java.util.*;
public class FindDups {
    public static void main(String args[]){
            Set s = new HashSet();
            for (int i = 0; i < args.length; i++)
            {
                        if (!s.add(args[i]))
                        System.out.println("Duplicate detected: "
    +args[i]);
            }
            System.out.println(s.size() +" distinct words detected: "
    +s);
    }
}
```

The List Interface

- The **List** interface corresponds to an order group of elements.

  Duplicates are allowed.

- Extensions compared to the **Collection** interface

  1. Access to elements

  add (int, Object), get(int), remove(int), set(int, Object)

- Search for elements

  indexOf(Object), lastIndexOf(Object)

- **add(Object)**adds at the end of the list.
- **remove(Object)**removes at the start of the list.
- **list1.equals(list2)**the ordering of the elements is taken into consideration.
- Extra requirements to the method **hashCode**.

  list1.equals(list2) implies that

  list1.hashCode()==list2.hashCode()

ArrayList and LinkedList Classes

- The classes **ArrayList** and **LinkedList** implement the

  List interface.
- **ArrayList** is an array based implementation where elements can be accessed directly via the **get** and **set** methods.

  1. Default choice for simple sequence.
- **LinkedList** is based on a double linked list

  1.Gives better performance on **add** and **remove** compared to **ArrayList**.

  2.Gives poorer performance on **get** and **set** methods compared to **ArrayList**.

ArrayList, Example

```
import java.util.*;
public class Shuffle {
   public static void main(String args[])
   {
         List l = new ArrayList();
         for (int i = 0; i < args.length; i++)
               l.add(args[i]);
         Collections.shuffle(l, new Random());
         System.out.println(l);
   }
}
```

LinkedList, Example

```
import java.util.*;
public class MyStack {
    private LinkedList list = new LinkedList();
    public void push(Object o)
    {
            list.addFirst(o);
    }
    public Object top(){
            return list.getFirst();
    }
    public Object pop(){
            return list.removeFirst();
    }
    public static void main(String args[]) {
            Car myCar;
            MyStack s = new MyStack();
            s.push (new Car());
            myCar = (Car)s.pop();
    }
}
```

HashMap and TreeMap Classes

- The **HashMap** and **HashTree** classes implement the **Map** interface.
- **HashMap**

  1.The implementation is based on a hash table.

  2.No ordering on (key, value) pairs.
- **TreeMap**

  1.The implementation is based on *red-black tree structure*.

  2.(key, value) pairs are ordered on the key.

HashMap, Example

```
import java.util.*;
public class Freq {
    private static final Integer ONE = new Integer(1);
    public static void main(String args[]) {
        Map m = new HashMap();
        // Initialize frequency table from command line
        for (int i=0; i < args.length; i++) {
            Integer freq = (Integer) m.get(args[i]);
            m.put(args[i], (freq==null ? ONE :new
                                    Integer(freq.intValue() +
    1)));
        }
        System.out.println(m.size()+" distinct words detected:");
        System.out.println(m);
    }
}
```

Advantages

- Can hold different types of objects.
- Resizable

Disadvantages

- Must cast to correct type
- Cannot do compile-time type checking.