

CSE201: Monsoon 2020, Section-A
Advanced Programming

Lecture 12: Collection Framework

Vivek Kumar

Computer Science and Engineering

IIT Delhi

vivekk@iiitd.ac.in

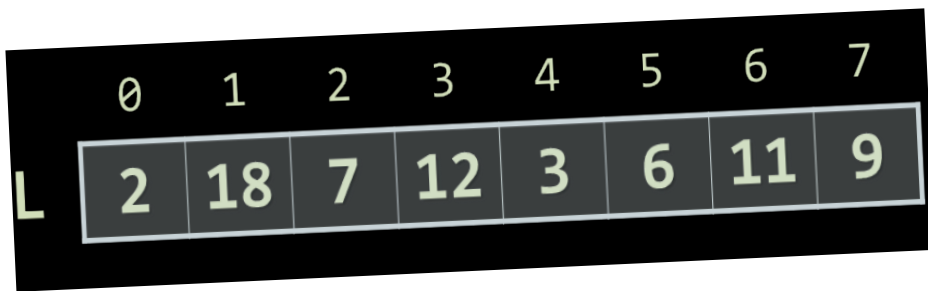
Today's Lecture

- Collections framework in Java

Note

- Today's slides will use all the concepts that you have learned so far in this course

How is your Experience using Arrays?



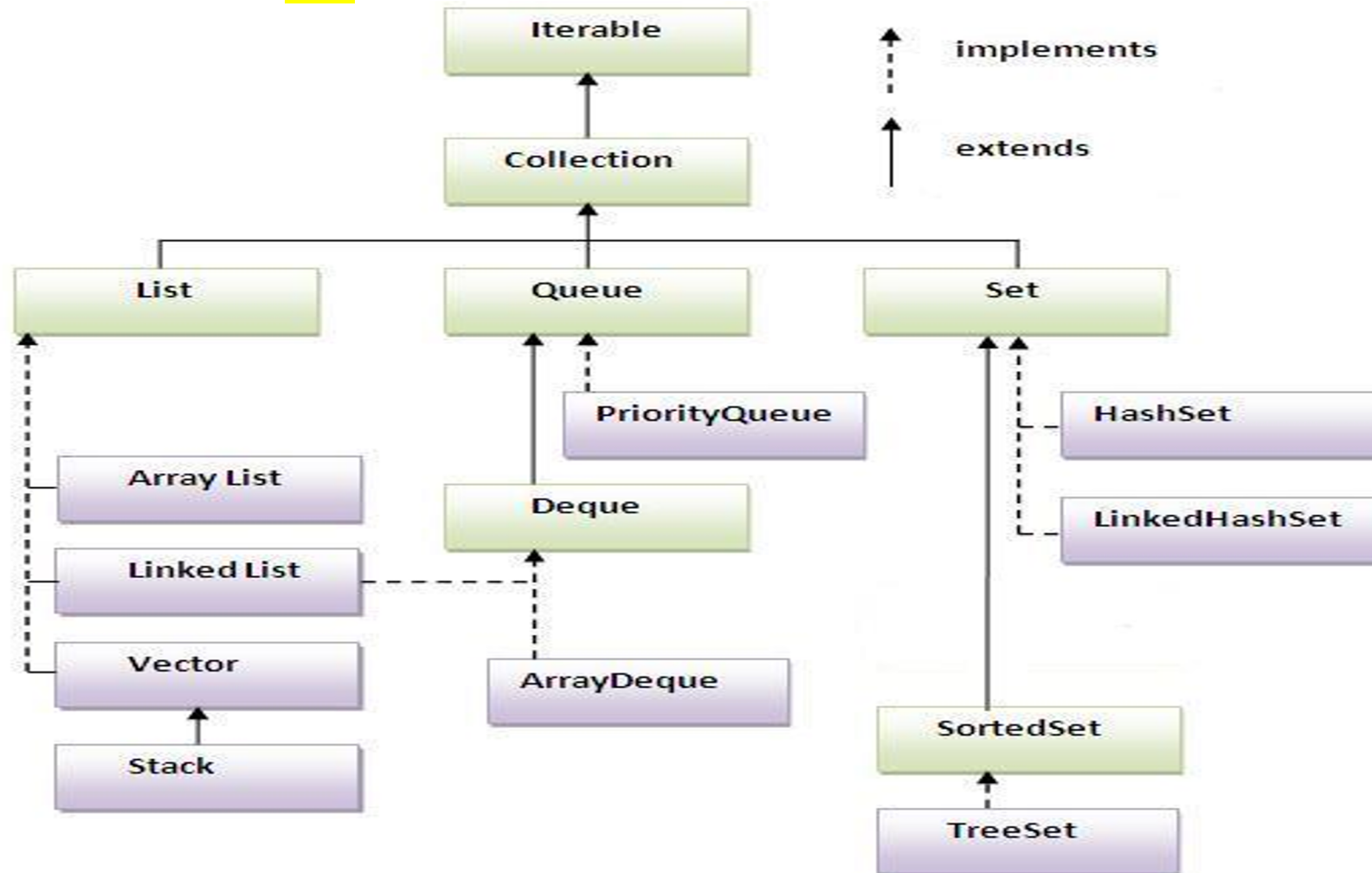
	0	1	2	3	4	5	6	7
L	2	18	7	12	3	6	11	9

- Has fixed size (length)
 - Can you do it programmatically?
 - Memory wastage?
- Deleting an element
 - Can you do it programmatically?
- Comparing two arrays
 - Can you use “==” or equals()?
- Can you assign one array to other?
 - `int a[], b[]; a=b`

Java Collection Framework

- Unified architecture for representing and manipulating collections
 - A collection (sometimes called a *container*) is simply an object that groups multiple elements into a single unit
 - Very useful
 - store, retrieve and manipulate data
 - transmit data from one method to another
- Collection framework contains three things
 - Interfaces
 - Implementations
 - Algorithms
- This group of collection classes/interfaces are referred to as Java Collection Framework (JCF)
 - The classes in JCF are found in package “java.util”.

Collection Hierarchy



Interface Can Extend Another Interface (1/2)

```
public interface Moveable {  
    public void move_left();  
    public void move_right();  
}
```

```
public interface Flyable {  
    public void fly_up();  
    public void fly_down();  
    public void move_left();  
    public void move_right();  
}
```

```
public class Car  
    implements Moveable {  
    ...  
    public void move_left() {  
        // move left  
    }  
    public void move_right() {  
        // move right  
    }  
    // more methods elided  
}
```

```
public class Airplane  
    implements Flyable {  
    ...  
    public void move_left() {  
        // move left  
    }  
    public void move_right() {  
        // move right  
    }  
    public void fly_up() {  
        // fly up  
    }  
    public void fly_down() {  
        // fly down  
    }  
}
```

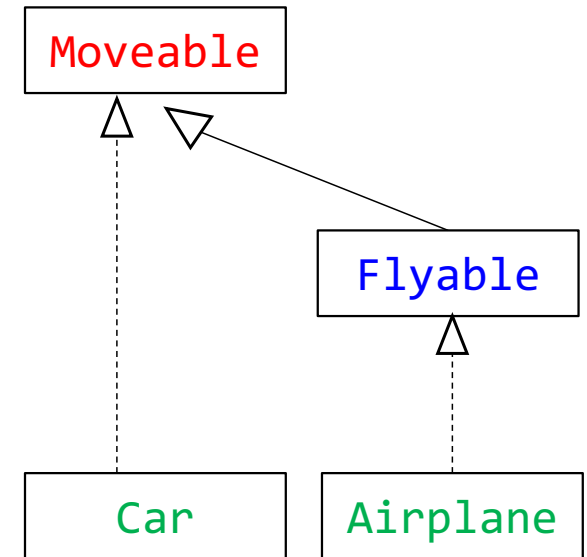
Interface Can Extend Another Interface (2/2)

```
public interface Moveable {  
    public void move_left();  
    public void move_right();  
}
```

```
public interface Flyable  
    extends Moveable {  
    public void fly_up();  
    public void fly_down();  
}
```

```
public class Car  
    implements Moveable {  
    ...  
    public void move_left() {  
        // move left  
    }  
    public void move_right() {  
        // move right  
    }  
    // more methods elided  
}
```

```
public class Airplane  
    implements Flyable {  
    ...  
    public void move_left() {  
        // move left  
    }  
    public void move_right() {  
        // move right  
    }  
    public void fly_up() {  
        // fly up  
    }  
    public void fly_down() {  
        // fly down  
    }  
}
```



Iterable Interface Source Code

```
package java.lang;  
public interface Iterable<E> {  
    Iterator<E> iterator();  
}
```

- Just one method in this interface
- Objects of all classes that implements this interface can be the target of **foreach** statement
- Iterators allow iterating over the entire collection. It also allows element removal from collection during iteration

Iterator Interface

- Defines three fundamental methods
 - `Object next()`
 - `boolean hasNext()`
 - `void remove()`
- These three methods provide access to the contents of the collection
- An Iterator knows position within collection
- Each call to `next()` “reads” an element from the collection
 - Then you can use it or remove it

Iterator Position

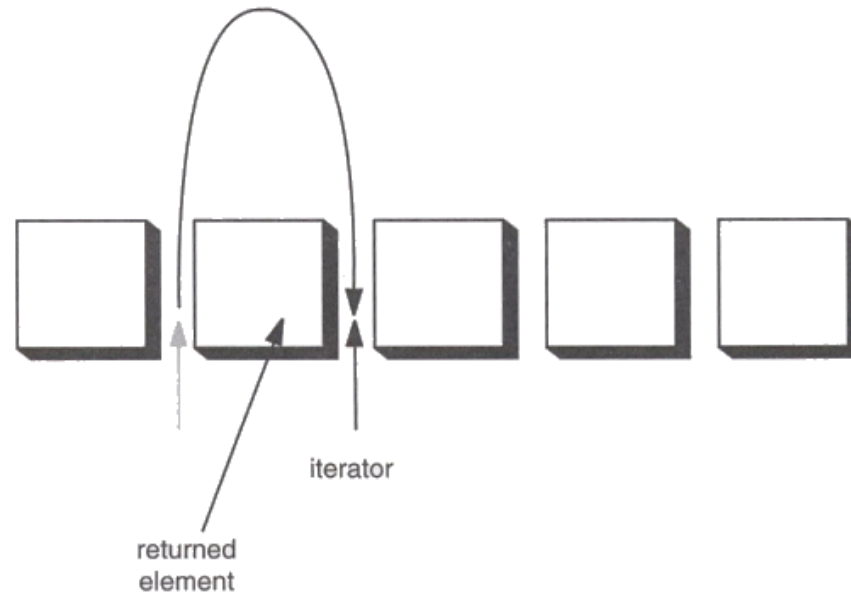


Figure 2-3: Advancing an iterator

Collection Interface Source Code

```
package java.util;
public interface Collection<E> extends Iterable<E>
{
    int size();
    boolean isEmpty();
    contains(Object o);
    boolean add(E e);
    boolean remove(Object o);
    equals(Object o);
    .....
    .....
}
```

- Defines fundamental methods that are enough to define the basic behavior of a collection
- Inherit the method from Iterable interface

Example - SimpleCollection

```
public class SimpleCollection {  
    public static void main(String[] args) {  
        Collection c = new ArrayList();  
        for (int i=0; i < 10; i++) {  
            c.add(i);  
        }  
        Iterator iter = c.iterator();  
        while (iter.hasNext())  
            System.out.println(iter.next());  
    }  
}
```

List Interface

- Recall in Java, arrays have fixed length
 - Cannot add / remove / insert elements
- Lists are like resizable arrays
 - Allow add / remove / insert of elements
- List **interface** is defined through the **ArrayList<E>** class
 - Where **E** is the type of the list, e.g. **String** or **Integer**

List Interface Source Code

```
package java.util;
public interface List<E> extends Collection<E> {
    E get(int index);
    E set(int index);
    void add(int index, E element);
    E remove(int index);
    ListIterator<E> listIterator();
    .....
    .....
}
```

- Observe that List interface has two different iterators
 - Iterator<E>
iterator();
 - ListIterator<E>
listIterator();

ListIterator Interface

- Extends the Iterator interface
- Defines three fundamental methods
 - `void add(Object o)` - before current position
 - `boolean hasPrevious()`
 - `Object previous()`
- The addition of these three methods defines the basic behavior of an ordered list
- Iterator v/s ListIterator
 - Unlike Iterator, a ListIterator knows position within list (obtain indexes)
 - Iterator allows traversal only in forward direction but ListIterator allows list traversal in both forward and backward directions
 - ListIterator can be used to traverse a List only

List Implementations

- ArrayList

- low cost random access (at an index)
- high cost insert and delete
 - Why?
- array that resizes if need be

- LinkedList

- sequential access but high cost random access (at an index)
- low cost insert and delete

ArrayList overview

- Constant time positional access (it's an array)
- One tuning parameter, the initial capacity to constructor
- Constructors
 - `ArrayList()`
 - Build an empty ArrayList (of initial size 10)
 - `ArrayList(Collection c)`
 - Build an ArrayList initialized with the elements of the collection c
 - `ArrayList(int initialCapacity)`
 - Build with the specified initial capacity

ArrayList Methods

- The indexed get and set methods of the List interface are appropriate to use since ArrayLists are backed by an array
 - `Object get(int index)`
 - `Object set(int index, Object element)`
 - May throw `IndexOutOfBoundsException`
- Indexed add and remove are provided, but can be costly if used frequently
 - `void add(int index, Object element)`
 - `Object remove(int index)`
 - May throw `IndexOutOfBoundsException`
- May want to resize in one shot if adding many elements
 - `void ensureCapacity(int minCapacity)`
- ArrayList allows adding duplicate elements

How ArrayList Store Objects in Heap?

```
public boolean add(E e) {  
    ensureCapacity(size+1);  
    elementData[size++] = e;  
    return true;  
}
```

- Resizable array implementation

```
// Increase the capacity if necessary to ensure that it can  
// hold atleast the minCapacity number of elements  
public void ensureCapacity(int minCapacity) {  
    .....  
    int oldCapacity = elementData.length;  
    if(minCapacity > oldCapacity) {  
        .....  
        int newCapacity = .....  
        elementData = Arrays.copyOf(elementData, newCapacity);  
    }  
}
```

LinkedList Overview (1/2)

- Stores each element in a node
- Each node stores a link to the next and previous nodes
 - Doubly linked list
 - Recall, ListIterator allows traversal both ways
- Insertion and removal are inexpensive
 - just update the links in the surrounding nodes
- Linear traversal is inexpensive
- Random access is expensive
 - Start from beginning or end and traverse each node while counting

LinkedList Overview (2/2)

- Constructors

- `LinkedList()`
 - Build an empty `LinkedList`
- `LinkedList(Collection c)`
 - Construct a list containing the elements of the specified collection, in the order they are returned by the collection's iterator
- Does it make sense to have a constructor with parameter as follows?
`LinkedList(int initialCapacity)`

LinkedList Methods

- ListIterator knows about position
 - use `add()` to add at a position
 - USE `remove()` to remove at a position
- Few other methods
 - `void addFirst(Object o), void addLast(Object o)`
 - `Object getFirst(), Object getLast()`
 - `Object removeFirst(), Object removeLast()`

Example: LinkedList

```
import java.util.*;
public class Book {
    private String name;
    private int pages;
    public Book(int p, String s) { ..... }
    @Override
    public String toString() { ..... }
    public static void main(String[] args) {
        List<Book> list = new LinkedList<Book>();

        list.add(new Book(100, "ABC"));
        list.add(new Book(200, "DEF"));
        list.add(new Book(300, "GHI"));

        for(Book b:list) {
            System.out.println(b);
        }
    }
}
```


Sets

- Sets keep unique elements only
 - Like lists but no duplicates
- HashSet<E>
 - Keeps a set of elements in a hash tables
 - Hash key?
 - The elements are randomly ordered by their hash code
- TreeSet<E>
 - Keeps a set of elements in a red-black ordered search tree
 - The elements are ordered incrementally

Set Interface

- Same methods as Collection
 - different contract - no duplicate entries
- Provides an Iterator to step through the elements in the Set
 - No guaranteed order in the basic Set interface

HashSet

- Find and add elements very quickly
 - uses hashing
- Hashing uses an array of linked lists
 - The **hashCode** () is used to index into the array
- No order imposed on elements

TreeSet

- Elements can be inserted in any order
 - The TreeSet stores them in order
- Default order is defined by natural order
 - Objects implement the Comparable interface
 - TreeSet uses `compareTo (Object o)` to sort

Example: TreeSet

```
import java.util.*;
public class Book implements Comparable<Book> {
    private String name;
    private int pages;
    public Book(int p, String s) { ..... }
    @Override
    public String toString() { ..... }
    public int compareTo(Book b) {
        if(this.page>b.getpage()) return 1;
        else if(this.page<b.getpage()) return -1;
        else return 0;
    }
    public static void main(String[] args) {
        Set<Book> set = new TreeSet<Book>();

        set.add(new Book(100, "ABC"));
        set.add(new Book(200, "DEF"));
        for(Book b:set) { // you can also use iterator
            System.out.println(b);
        }
    }
}
```

- The elements in TreeSet must be of Comparable type
- You need to add compareTo in user defined classes

Maps

- Maps keep unique <key, value> pairs
- HashMap<K, V>
 - Keeps a map of elements in a hash table
 - The elements are randomly ordered using their hash code
- TreeMap<K, V>
 - Keep a set of elements in a red-black ordered search tree
 - The elements are ordered incrementally by their key

Map Interface

- Stores unique key/value pairs
- Maps from the key to the value
- Keys are unique
 - a single key only appears once in the Map
 - a key can map to only one value
- Values do not have to be unique

Example: HashMap

```
import java.util.*;
public class Book {
    private String name;
    private int pages;
    public Book(int p, String s) { ..... }
    @Override
    public String toString() { ..... }
    public static void main(String[] args) {
        Map<Integer, Book> map = new HashMap<Integer, Book>();

        map.add(1, new Book(100, "ABC"));
        map.add(2, new Book(200, "DEF"));
        for(Map.Entry e:map.entrySet()) {
            System.out.println(e.getKey() + ":" + e.getValue());
        }
    }
}
```


Next Lecture

- I/O Streams
- Assignment-3 on Friday (Tomorrow at 9am)