Java & JEE Training

Day 17 – Collections – Lists, Sets

MindsMapped Consulting

Quick Recap...

- How to define a class completely
  - Define a POJO
    - Define private data members
    - Define public getters and setters
  - Override
    - toString() for more meaningful logging and printing
    - equals() for search in a collection
    - hashCode() for providing hashing function
    - Comparison logic
      - java.lang.Comparable interface - DEFAULT
      - java.util.Comparator interface – A Comparator class for every comparison logic

Agenda...

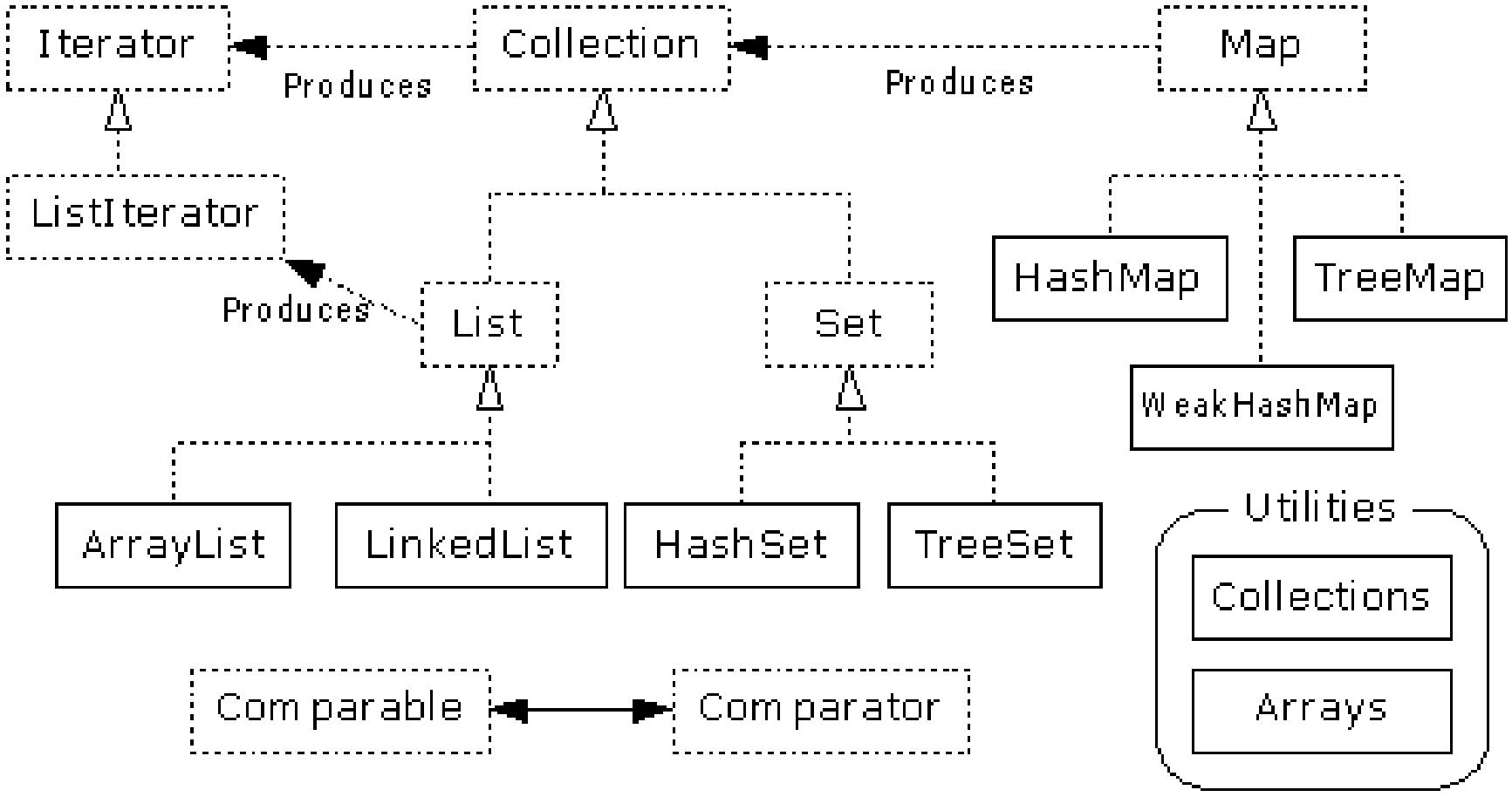
- List – ArrayList, LinkedList
- Set – HashSet, LinkedHashSet, TreeSet

Java & JEE Training

Collections

MindsMapped Consulting

Collections Framework Diagram



Collection Interface

- Defines fundamental methods
  - `int size();`
  - `boolean isEmpty();`
  - `boolean contains(Object element);`
  - `boolean add(Object element);    // Optional`
  - `boolean remove(Object element); // Optional`
  - `Iterator iterator();`
- These methods are enough to define the basic behavior of a collection
- Provides an Iterator to step through the elements in the Collection

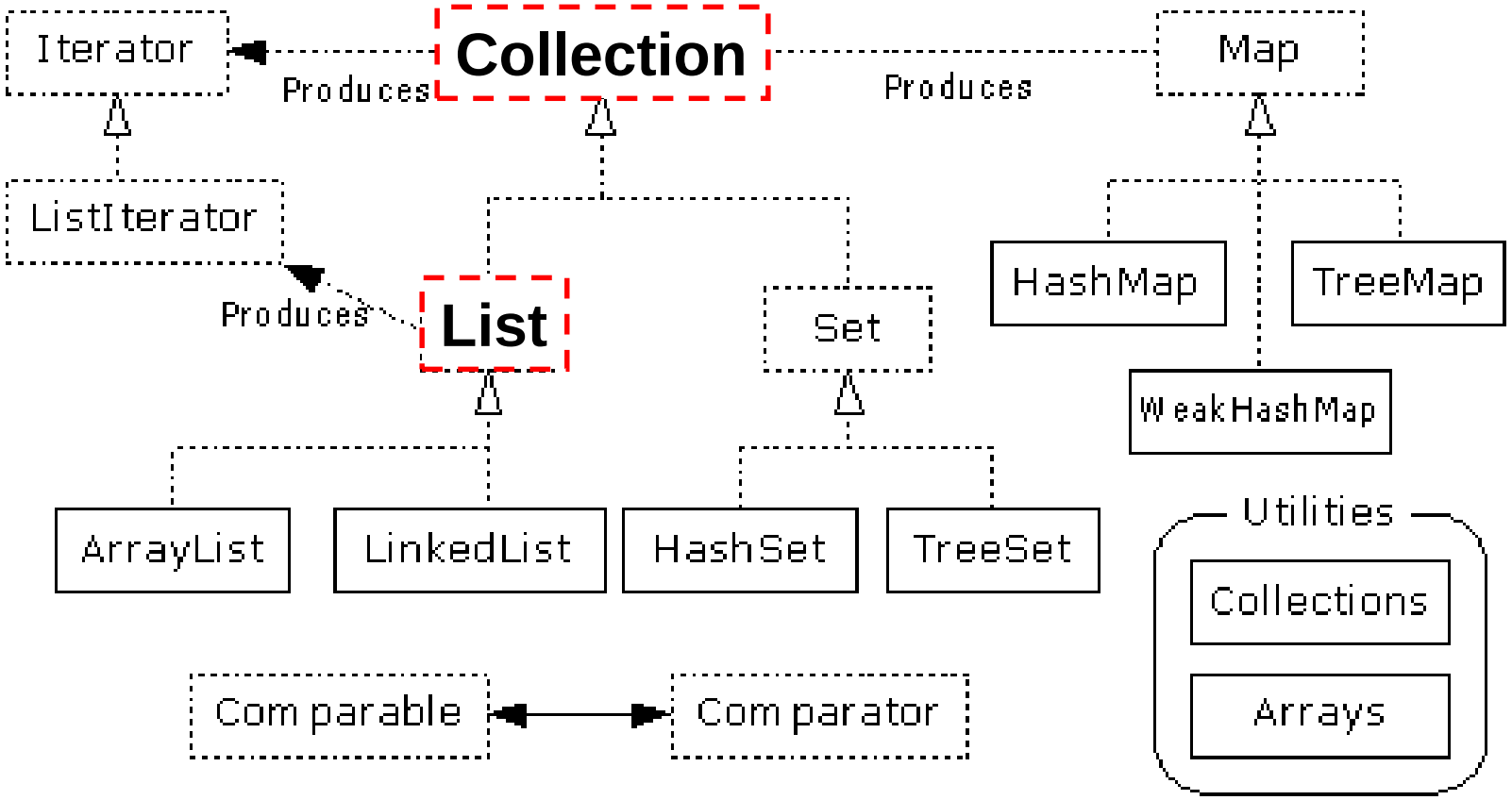
Iterator Interface

- 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

Example - SimpleCollection

```
public class SimpleCollection  {
 public static void main(String[] args) {
    Collection c;
    c = new ArrayList();
    System.out.println(c.getClass().getName());
    for (int i=1; i <= 10; i++) {
           c.add(i + " * " + i + " = "+i*i);
    }
    Iterator iter = c.iterator();
    while (iter.hasNext())
           System.out.println(iter.next());
  }
}
```

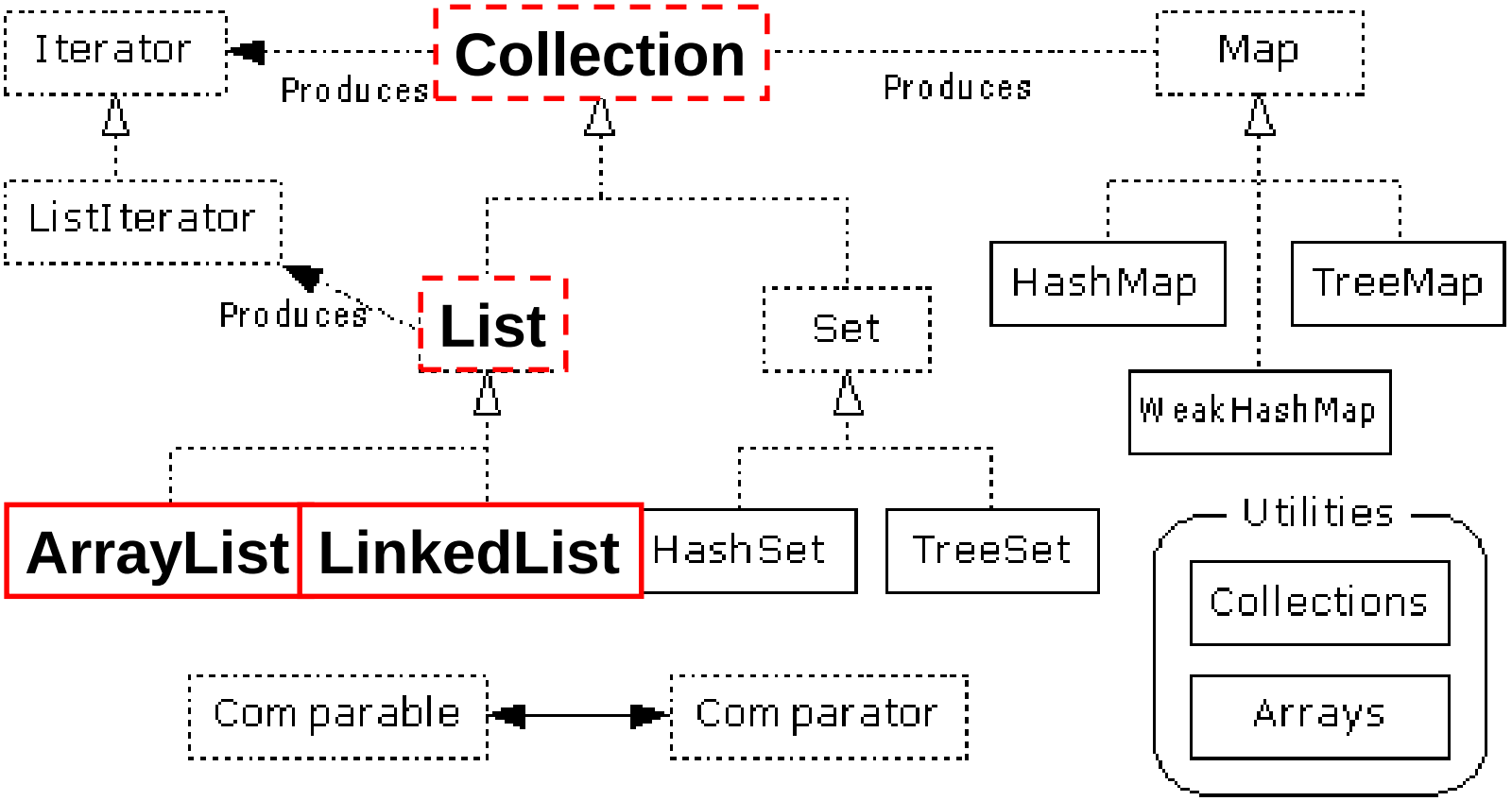
List Interface Context



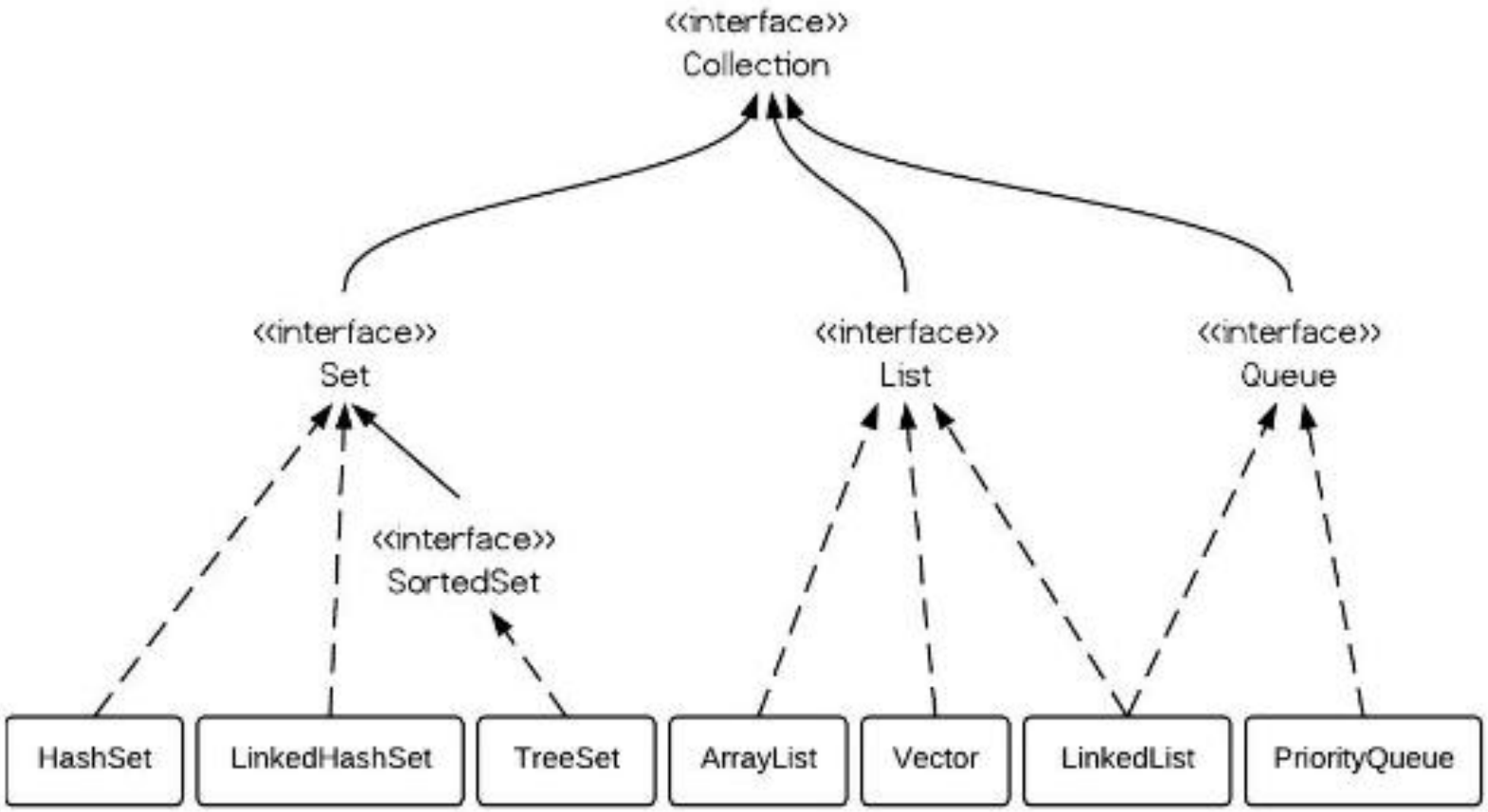
ListIterator Interface

- 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
- A ListIterator knows position within list

ArrayList and LinkedList Context



List as part of Collection



List Implementations

- ArrayList
  - low cost random access
  - high cost insert and delete
  - array that resizes if need be
- LinkedList
  - sequential access
  - low cost insert and delete
  - high cost random access
- Vector
  - Similar to ArrayList, but thread-safe

ArrayList overview

- Constant time positional access (it's an array)
- One tuning parameter, the initial capacity

```
public ArrayList(int initialCapacity) {
    super();
    if (initialCapacity < 0)
            throw new IllegalArgumentException(
                    "Illegal Capacity:
    "+initialCapacity);
    this.elementData = new Object[initialCapacity];
}
```

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)`
- Indexed add and remove are provided, but can be costly if used frequently
  - `void add(int index, Object element)`
  - `Object remove(int index)`
- May want to resize in one shot if adding many elements
  - `void ensureCapacity(int minCapacity)`

LinkedList overview

- Stores each element in a node
- Each node stores a link to the next and previous nodes
- 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 entries

```
private static class Entry {
  Object element;
  Entry next;
  Entry previous;

  Entry(Object element, Entry next, Entry previous) {
      this.element = element;
      this.next = next;
      this.previous = previous;
  }
}

private Entry header = new Entry(null, null, null);

public LinkedList() {
  header.next = header.previous = header;
}
```

LinkedList methods

- The list is sequential, so access it that way
  - **ListIterator listIterator()**
- ListIterator knows about position
  - use **add()** from ListIterator to add at a position
  - use **remove()** from ListIterator to remove at a position
- LinkedList knows a few things too
  - **void addFirst(Object o), void addLast(Object o)**
  - **Object getFirst(), Object getLast()**
  - **Object removeFirst(), Object removeLast()**

ArrayList vs. LinkedList vs. Vector

From the hierarchy diagram, they all implement List interface. They are very similar to use. Their main difference is their implementation which causes different performance for different operations. ArrayList is implemented as a resizable array. As more elements are added to ArrayList, its size is increased dynamically. It's elements can be accessed directly by using the get and set methods, since ArrayList is essentially an array. LinkedList is implemented as a double linked list. Its performance on add and remove is better than Arraylist, but worse on get and set methods. Vector is similar with ArrayList, but it is synchronized. ArrayList is a better choice if your program is thread-safe. Vector and ArrayList require space as more elements are added. Vector each time doubles its array size, while ArrayList grow 50% of its size each time. LinkedList, however, also implements Queue interface which adds more methods than ArrayList and Vector, such as offer(), peek(), poll(), etc. Note: The default initial capacity of an ArrayList is pretty small. It is a good habit to construct the ArrayList with a higher initial capacity. This can avoid the resizing cost.

Example: ArrayList

```
ArrayList al = new ArrayList();
al.add(3);
al.add(2);
al.add(1);
al.add(4);
al.add(5);
al.add(6);
al.add(6);

Iterator iter1 = al.iterator();
while(iter1.hasNext()){
   System.out.println(iter1.next());
}
```

Example: ArrayList (Using Generics + Iterating using Iterator and for-each loop)

```
import java.util.*;
class TestCollection1{
 public static void main(String args[]){
  ArrayList<String> list=new ArrayList<String>();//Creating arraylist
  list.add("Ravi");//Adding object in arraylist
  list.add("Vijay");
  list.add("Ravi");
  list.add("Ajay");
  //Traversing list through Iterator
  Iterator itr=list.iterator();
  while(itr.hasNext()){
   System.out.println(itr.next());
  }
  //Traversing using for-each loop
  for(String obj:list)
    System.out.println(obj);
 }
}
```

User-defined class objects in Java ArrayList

```
class Student{
  int rollno;
  String name;
  int age;
  Student(int rollno,String name,int age){
   this.rollno=rollno;
   this.name=name;
   this.age=age;
  }
}
```

```
import java.util.*;
public class TestCollection3{
 public static void main(String args[]){
  //Creating user-defined class objects
  Student s1=new Student(101,"Sonoo",23);
  Student s2=new Student(102,"Ravi",21);
  Student s2=new Student(103,"Hanumat",25);
  //creating arraylist
  ArrayList<Student> al=new ArrayList<Student>();
  al.add(s1);//adding Student class object
  al.add(s2);
  al.add(s3);
  //Getting Iterator
  Iterator itr=al.iterator();
  //traversing elements of ArrayList object
  while(itr.hasNext()){
    Student st=(Student)itr.next();
    System.out.println(st.rollno+" "+st.name+"
"+st.age);
  }
 }
}
```

ArrayList Constructors

Constructor	Description
ArrayList()	It is used to build an empty array list.
ArrayList(Collection c)	It is used to build an array list that is initialized with the elements of the collection c.
ArrayList(int capacity)	It is used to build an array list that has the specified initial capacity.

ArrayList Methods

Method	Description
void add(int index, Object element)	It is used to insert the specified element at the specified position index in a list.
boolean addAll(Collection c)	It is used to append all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator.
void clear()	It is used to remove all of the elements from this list.
int lastIndexOf(Object o)	It is used to return the index in this list of the last occurrence of the specified element, or -1 if the list does not contain this element.
Object[] toArray()	It is used to return an array containing all of the elements in this list in the correct order.
Object[] toArray(Object[] a)	It is used to return an array containing all of the elements in this list in the correct order.
boolean add(Object o)	It is used to append the specified element to the end of a list.
boolean addAll(int index, Collection c)	It is used to insert all of the elements in the specified collection into this list, starting at the specified position.
Object clone()	It is used to return a shallow copy of an ArrayList.
int indexOf(Object o)	It is used to return the index in this list of the first occurrence of the specified element, or -1 if the List does not contain this element.
void trimToSize()	It is used to trim the capacity of this ArrayList instance to be the list's current size.

Example of addAll(Collection c) method

```
import java.util.*;
class TestCollection4{
 public static void main(String args[]){
  ArrayList<String> al=new ArrayList<String>();
  al.add("Ravi");
  al.add("Vijay");
  al.add("Ajay");
  ArrayList<String> al2=new ArrayList<String>();
  al2.add("Sonoo");
  al2.add("Hanumat");
  al.addAll(al2);//adding second list in first list
  Iterator itr=al.iterator();
  while(itr.hasNext()){
   System.out.println(itr.next());
  }
 }
}
```

Example of removeAll() method

```
import java.util.*;
class TestCollection5{
 public static void main(String args[]){
  ArrayList<String> al=new ArrayList<String>();
  al.add("Ravi");
  al.add("Vijay");
  al.add("Ajay");
  ArrayList<String> al2=new ArrayList<String>();
  al2.add("Ravi");
  al2.add("Hanumat");
  al.removeAll(al2);
  System.out.println("iterating the elements after removing the elements of
al2...");
  Iterator itr=al.iterator();
  while(itr.hasNext()){
   System.out.println(itr.next());
  }

  }
}
```

Example of retainAll()

```
import java.util.*;
class TestCollection6{
 public static void main(String args[]){
  ArrayList<String> al=new ArrayList<String>();
  al.add("Ravi");
  al.add("Vijay");
  al.add("Ajay");
  ArrayList<String> al2=new ArrayList<String>();
  al2.add("Ravi");
  al2.add("Hanumat");
  al.retainAll(al2);
  System.out.println("iterating the elements after retaining the elements of al2...");
  Iterator itr=al.iterator();
  while(itr.hasNext()){
   System.out.println(itr.next());
  }
 }
}
```

Exercise...

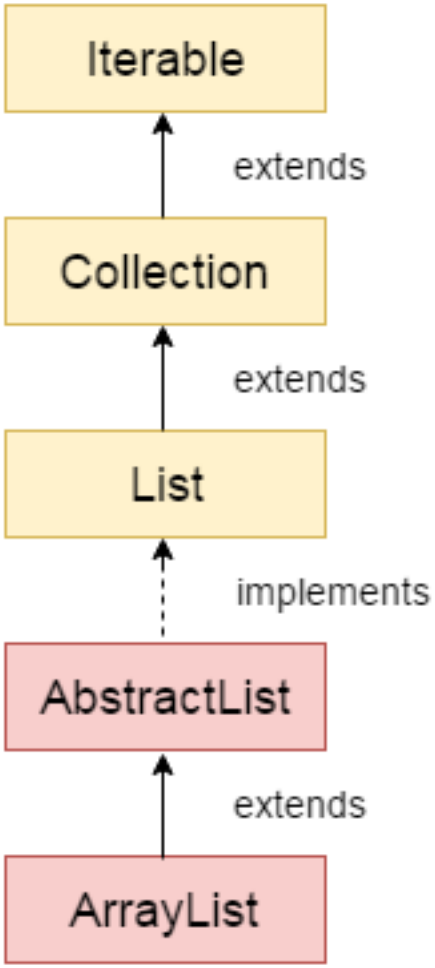
- Write a class “Book” that takes as members id, name, author, publisher and quantity.
- Write a program for creating an ArrayList of Books. Add 5 books to the list.
- Iterate through the list and print all the properties of the books.

Exercise… ListIterator

- What is a ListIterator?
- When we already have Iterator, why do we need ListIterator? What are the advantages?

Exercise: ArrayList Hierarchy

The ArrayList Hierarchy is shown. Go through the Java API on Oracle's site and draw the hierarchy for all collections discussed in class today.



LinkedList

- Uses doubly linked list to store the elements. It provides a linked-list data structure. It inherits the AbstractList class and implements List and Deque interfaces
- Java LinkedList class can contain duplicate elements.
- Java LinkedList class maintains insertion order.
- Java LinkedList class is non synchronized.
- In Java LinkedList class, manipulation is fast because no shifting needs to be occurred.
- Java LinkedList class can be used as list, stack or queue.

LinkedList constructors

Constructor	Description
LinkedList()	It is used to construct an empty list.
LinkedList(Collection c)	It is used to construct a list containing the elements of the specified collection, in the order they are returned by the collection's iterator.

LinkedList Methods

Method	Description
void add(int index, Object element)	It is used to insert the specified element at the specified position index in a list.
void addFirst(Object o)	It is used to insert the given element at the beginning of a list.
void addLast(Object o)	It is used to append the given element to the end of a list.
int size()	It is used to return the number of elements in a list
boolean add(Object o)	It is used to append the specified element to the end of a list.
boolean contains(Object o)	It is used to return true if the list contains a specified element.
boolean remove(Object o)	It is used to remove the first occurence of the specified element in a list.
Object getFirst()	It is used to return the first element in a list.
Object getLast()	It is used to return the last element in a list.
int indexOf(Object o)	It is used to return the index in a list of the first occurrence of the specified element, or -1 if the list does not contain any element.
int lastIndexOf(Object o)	It is used to return the index in a list of the last occurrence of the specified element, or -1 if the list does not contain any element.

Example: LinkedList

```
LinkedList ll = new LinkedList();
ll.add(3);
ll.add(2);
ll.add(1);
ll.add(4);
ll.add(5);
ll.add(6);
ll.add(6);

Iterator iter2 = al.iterator();
while(iter2.hasNext()){
   System.out.println(iter2.next());
}
```

Exercise: LinkedList

Implement all the program examples written for ArrayList with LinkedList.

Performance: ArrayList vs LinkedList

```
ArrayList arrayList = new ArrayList();
LinkedList linkedList = new LinkedList();

// ArrayList add
long startTime = System.nanoTime();

for (int i = 0; i < 100000; i++) {
arrayList.add(i);
}
long endTime = System.nanoTime();
long duration = endTime - startTime;
System.out.println("ArrayList add:  " + duration);

// LinkedList add
startTime = System.nanoTime();

for (int i = 0; i < 100000; i++) {
linkedList.add(i);
}
endTime = System.nanoTime();
duration = endTime - startTime;
System.out.println("LinkedList add: " + duration);

// ArrayList get
startTime = System.nanoTime();

for (int i = 0; i < 10000; i++) {
arrayList.get(i);
}
endTime = System.nanoTime();
duration = endTime - startTime;
System.out.println("ArrayList get:  " + duration);
```

```
// LinkedList get
startTime = System.nanoTime();

for (int i = 0; i < 10000; i++) {
linkedList.get(i);
}
endTime = System.nanoTime();
duration = endTime - startTime;
System.out.println("LinkedList get: " + duration);


// ArrayList remove
startTime = System.nanoTime();

for (int i = 9999; i >=0; i--) {
arrayList.remove(i);
}
endTime = System.nanoTime();
duration = endTime - startTime;
System.out.println("ArrayList remove:  " + duration);

// LinkedList remove
startTime = System.nanoTime();

for (int i = 9999; i >=0; i--) {
linkedList.remove(i);
}
endTime = System.nanoTime();
duration = endTime - startTime;
System.out.println("LinkedList remove: " + duration);
```

ArrayList vs LinkedList

	ArrayList	LinkedList
Implementation	Resizable Array	Douby-LinkedList
ReverseIterator	No	Yes , descendingIterator()
Initial Capacity	10	Constructs empty list
get(int) operation	Fast	Slow in comparision
add(int) operation	Slow in comparision	Fast
Memory Overhead	No	Yes

List Interface in Java

- **public interface** List<E> **extends** Collection<E>

Method	Description
void add(int index,Object element)	It is used to insert element into the invoking list at the index passed in the index.
boolean addAll(int index,Collection c)	It is used to insert all elements of c into the invoking list at the index passed in the index.
object get(int index)	It is used to return the object stored at the specified index within the invoking collection.
object set(int index,Object element)	It is used to assign element to the location specified by index within the invoking list.
object remove(int index)	It is used to remove the element at position index from the invoking list and return the deleted element.
ListIterator listIterator()	It is used to return an iterator to the start of the invoking list.
ListIterator listIterator(int index)	It is used to return an iterator to the invoking list that begins at the specified index.

Java ListIterator Interface

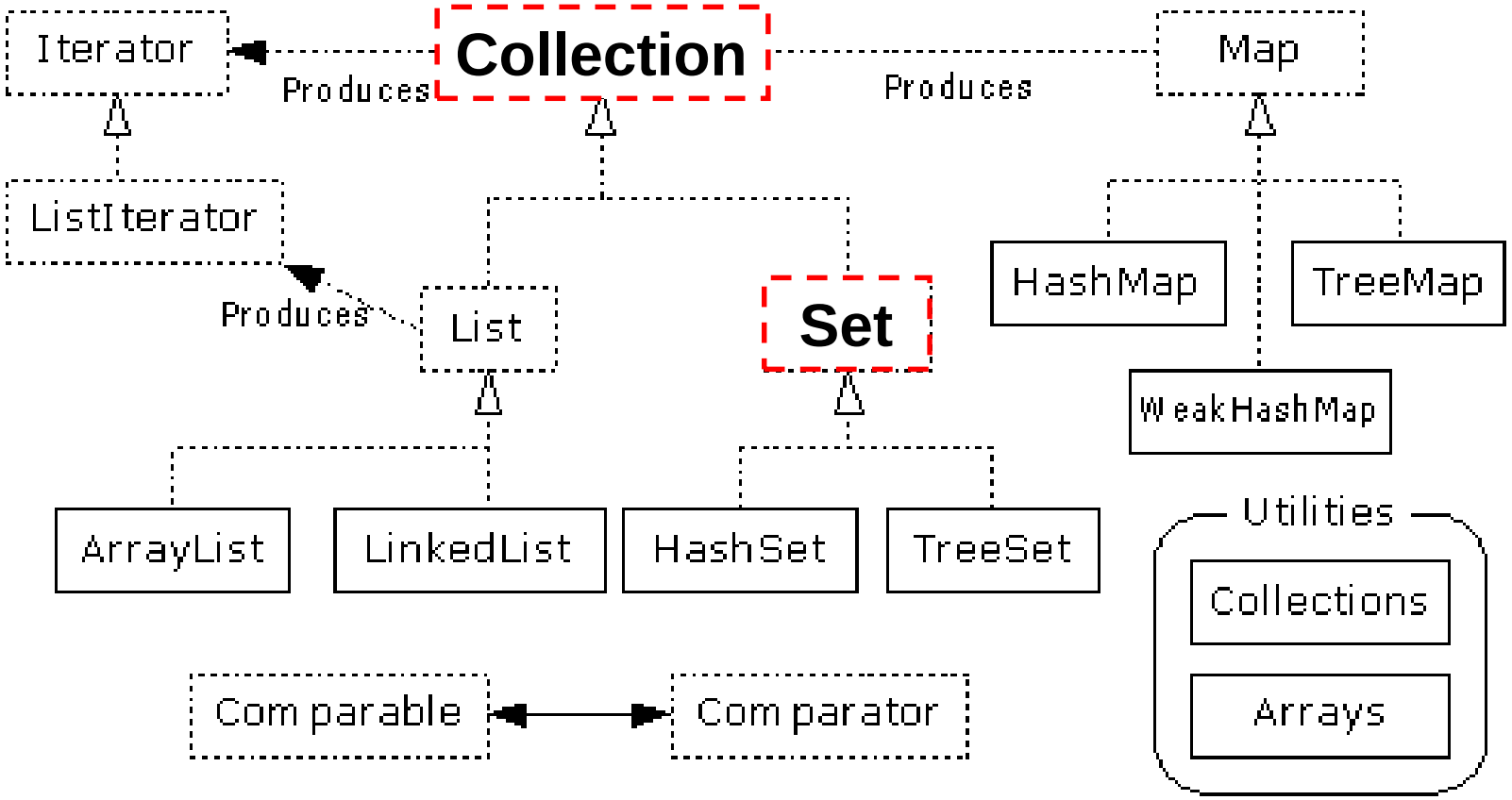
- **public interface** ListIterator<E> **extends** Iterator<E>
- Can be used to traverse through the list in forward and backwards/reverse direction

Method	Description
boolean hasNext()	This method return true if the list iterator has more elements when traversing the list in the forward direction.
Object next()	This method return the next element in the list and advances the cursor position.
boolean hasPrevious()	This method return true if this list iterator has more elements when traversing the list in the reverse direction.
Object previous()	This method return the previous element in the list and moves the cursor position backwards.

Java ListIterator example

```
import java.util.*;
public class TestCollection8{
public static void main(String args[]){

ArrayList<String> al=new ArrayList<String>();
al.add("Amit");
al.add("Vijay");
al.add("Kumar");
al.add(1,"Sachin");

System.out.println("element at 2nd position: "+al.get(2));

ListIterator<String> itr=al.listIterator();

System.out.println("traversing elements in forward direction...");
while(itr.hasNext()){
System.out.println(itr.next());
 }

System.out.println("traversing elements in backward direction...");
while(itr.hasPrevious()){
System.out.println(itr.previous());
 }
}
}
```

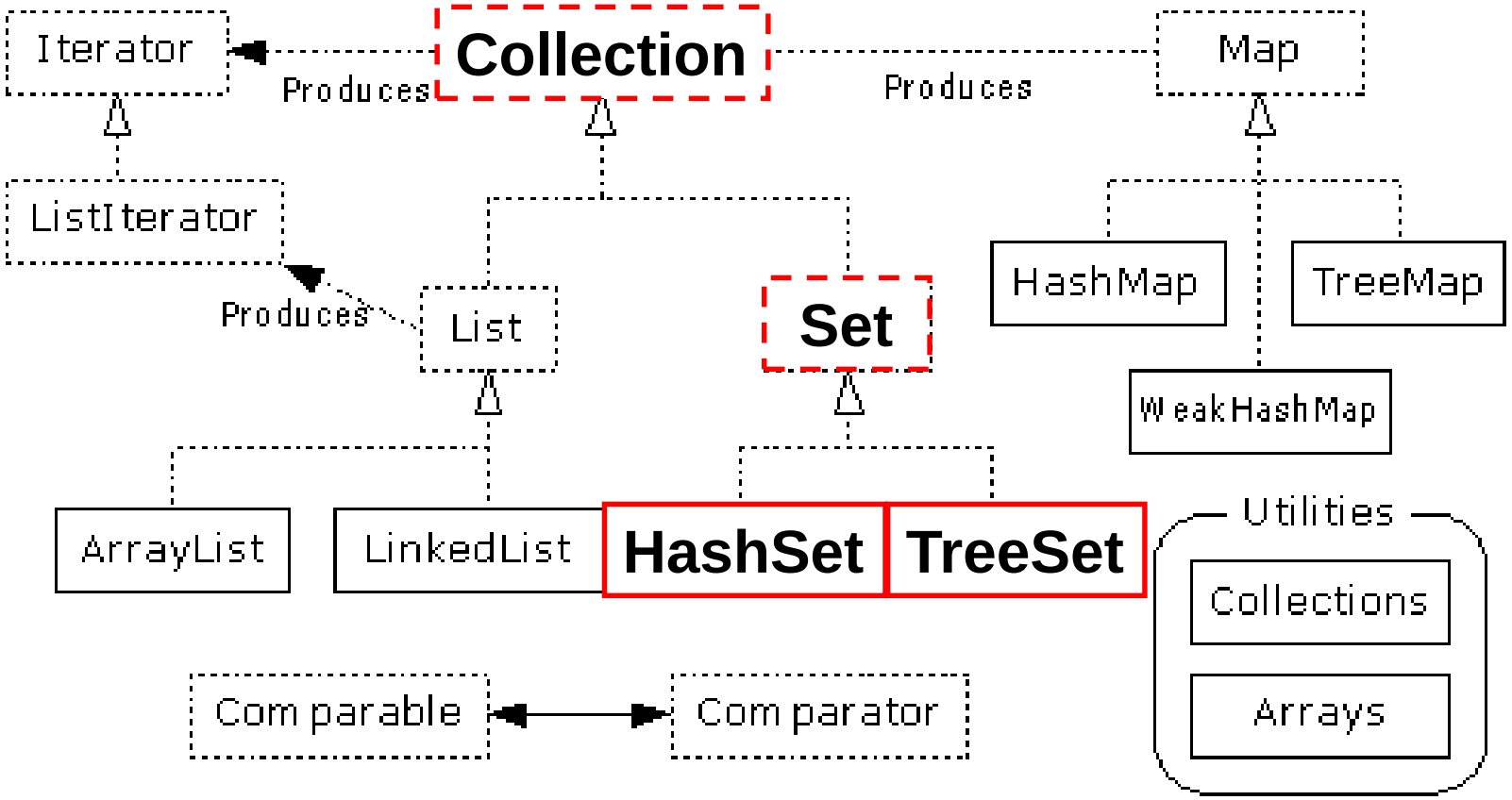
Set Interface Context



Set Interface

- Same methods as Collection
  - different contract - no duplicate entries
- Defines two fundamental methods
  - `boolean add(Object o)` - reject duplicates
  - `Iterator iterator()`
- Provides an Iterator to step through the elements in the Set
  - No guaranteed order in the basic Set interface
  - There is a SortedSet interface that extends Set

HashSet and TreeSet Context



List vs Set

- List can contain duplicate elements whereas Set contains unique elements only.

HashSet

- Find and add elements very quickly
  - uses hashing implementation in HashMap
- Hashing uses an array of linked lists
  - The **`hashCode()`** is used to index into the array
  - Then **`equals()`** is used to determine if element is in the (short) list of elements at that index
- No order imposed on elements
- The **`hashCode()`** method and the **`equals()`** method must be compatible
  - if two objects are equal, they must have the same **`hashCode()`** value

HashSet constructors

Constructor	Description
HashSet()	It is used to construct a default HashSet.
HashSet(Collection c)	It is used to initialize the hash set by using the elements of the collection c.
HashSet(int capacity)	It is used to initialize the capacity of the hash set to the given integer value capacity. The capacity grows automatically as elements are added to the HashSet.

HashSet methods

Method	Description
void clear()	It is used to remove all of the elements from this set.
boolean contains(Object o)	It is used to return true if this set contains the specified element.
boolean add(Object o)	It is used to adds the specified element to this set if it is not already present.
boolean isEmpty()	It is used to return true if this set contains no elements.
boolean remove(Object o)	It is used to remove the specified element from this set if it is present.
Object clone()	It is used to return a shallow copy of this HashSet instance: the elements themselves are not cloned.
Iterator iterator()	It is used to return an iterator over the elements in this set.
int size()	It is used to return the number of elements in this set.

HashSet Example

```
import java.util.*;
class TestCollection9{
 public static void main(String args[]){
  //Creating HashSet and adding elements
  HashSet<String> set=new HashSet<String>();
  set.add("Ravi");
  set.add("Vijay");
  set.add("Ravi");
  set.add("Ajay");
  //Traversing elements
  Iterator<String> itr=set.iterator();
  while(itr.hasNext()){
   System.out.println(itr.next());
  }
 }
}
```

HashSet Example : Book

```
import java.util.*;
class Book {
int id;
String name,author,publisher;
int quantity;
public Book(int id, String name, String author,
String publisher, int quantity) {
    this.id = id;
    this.name = name;
    this.author = author;
    this.publisher = publisher;
    this.quantity = quantity;
}
}
```

```
public class HashSetExample {
public static void main(String[] args) {
    HashSet<Book> set=new HashSet<Book>();
    //Creating Books
    Book b1=new Book(101,"Let us
C","Yashwant Kanetkar","BPB",8);
    Book b2=new Book(102,"Data
Communications & Networking","Forouzan","Mc
Graw Hill",4);
    Book b3=new Book(103,"Operating
System","Galvin","Wiley",6);
    //Adding Books to HashSet
    set.add(b1);
    set.add(b2);
    set.add(b3);
    //Traversing HashSet
    for(Book b:set){
    System.out.println(b.id+" "+b.name+"
"+b.author+" "+b.publisher+" "+b.quantity);
    }
}
}
```

Exercise...

- What is LinkedHashSet?
- When would you use LinkedHashSet?

TreeSet

- Elements can be inserted in any order
- The TreeSet stores them in order
- An iterator always presents them in order
- Default order is defined by natural order
  - objects implement the Comparable interface
  - TreeSet uses `compareTo(Object o)` to sort
- Can use a different Comparator
  - provide Comparator to the TreeSet constructor

TreeSet constructor

Constructor	Description
TreeSet()	It is used to construct an empty tree set that will be sorted in an ascending order according to the natural order of the tree set.
TreeSet(Collection c)	It is used to build a new tree set that contains the elements of the collection c.
TreeSet(Comparator comp)	It is used to construct an empty tree set that will be sorted according to given comparator.
TreeSet(SortedSet ss)	It is used to build a TreeSet that contains the elements of the given SortedSet.

TreeSet Methods

Method	Description
boolean addAll(Collection c)	It is used to add all of the elements in the specified collection to this set.
boolean contains(Object o)	It is used to return true if this set contains the specified element.
boolean isEmpty()	It is used to return true if this set contains no elements.
boolean remove(Object o)	It is used to remove the specified element from this set if it is present.
void add(Object o)	It is used to add the specified element to this set if it is not already present.
void clear()	It is used to remove all of the elements from this set.
Object clone()	It is used to return a shallow copy of this TreeSet instance.
Object first()	It is used to return the first (lowest) element currently in this sorted set.
Object last()	It is used to return the last (highest) element currently in this sorted set.
int size()	It is used to return the number of elements in this set.

TreeSet Example

```
import java.util.*;
class TestCollection11{
 public static void main(String args[]){
  //Creating and adding elements
  TreeSet<String> al=new TreeSet<String>();
  al.add("Ravi");
  al.add("Vijay");
  al.add("Ravi");
  al.add("Ajay");
  //Traversing elements
  Iterator<String> itr=al.iterator();
  while(itr.hasNext()){
   System.out.println(itr.next());
  }
 }
}
```

TreeSet Example: Book

```
import java.util.*;
class Book implements Comparable<Book>{
int id;
String name,author,publisher;
int quantity;
public Book(int id, String name, String author,
String publisher, int quantity) {
    this.id = id;
    this.name = name;
    this.author = author;
    this.publisher = publisher;
    this.quantity = quantity;
}
public int compareTo(Book b) {
    if(id>b.id){
        return 1;
    }else if(id<b.id){
        return -1;
    }else{
    return 0;
    }
}
}
```

```
public class TreeSetExample {
public static void main(String[] args) {
    Set<Book> set=new TreeSet<Book>();
    //Creating Books
    Book b1=new Book(121,"Let us C","Yashwant
Kanetkar","BPB",8);
    Book b2=new Book(233,"Operating
System","Galvin","Wiley",6);
    Book b3=new Book(101,"Data Communications &
Networking","Forouzan","Mc Graw Hill",4);
    //Adding Books to TreeSet
    set.add(b1);
    set.add(b2);
    set.add(b3);
    //Traversing TreeSet
    for(Book b:set){
    System.out.println(b.id+" "+b.name+"
"+b.author+" "+b.publisher+" "+b.quantity);
    }
}
}
```