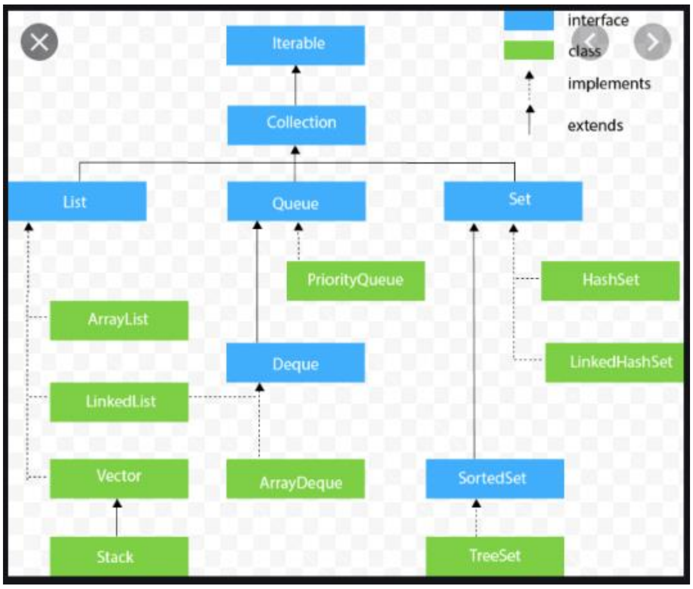
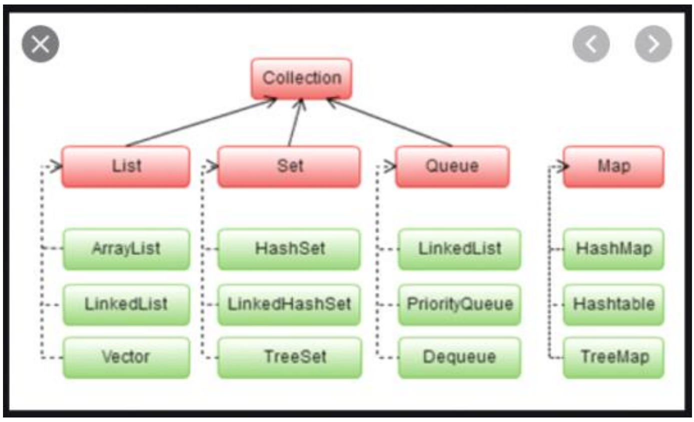
Collections

01 July 2020 09:02





Collection framework is used to store the group of objects
ArrayList extends AbstractList which implements List interface
If you print any collection it will print values present in the collection because it has overridden the toString() method
Where as Array cannot print

Array List:
================================

- **Array List accepts Duplicates**
- **Defauly capacity of AL is 10**
- **What is the data structure used by ArrayList is Array (Object Array)**
- **There is a variable present in Arraylist class and it creates the Array list with this capacity when we create the object or when we call the constructor**
- **Array List allows duplicates**
- **Array list is not synchronized.**
- **Array List is Ordered**

What happens inside add method ?
AL internally uses the Object array (Object[])
First it checks for the capacity
If is reached then it will increment the capacity by 50% on old capacity
Later it does the copy of the elements

If the capacity is not reached then directly it will add the elements

What internally happens when remove(2) is called

The argument that the remove method accepts is index.
First AL checks whether the index is present or not, if not present then throws Array index out of bound exception if not get the element present in the index and then it removes from that position and re arrange the AL and returns the removed element

How to convert Array to AL
Using ->iteration or Arrays.asList();
After performing the Arrays.asList we cannot add any new element because it is not creating the AL.
To solve this
List l= new ArrayList(Arrays.asList());
This statement will call the single arguemnet constructor of AL which accepts collection object

Vector
============
In AL increase capacity by 50 % of the existing capacity where Vector will increment the capacity by 100%
Vector is synchronized and AL is not synchronized.
Vector is also Ordered and allows duplicates

Custom ArrayList
=====================

1. **Create the class**
2. **Have the variables for default value, index**
3. **Define the array, because AL internally uses Array List**
4. **Create the Default constructor and initialize the array**
5. **Create Add method**
6. **In Add method check the capacity by comparing index and the default capacity**
7. **If index reached to default capacity then increment the size by 50% and do array copy**
8. **Else Add the element and increment the index.**
9. **Create another class and instead of using the AL use custom arraylist**
10. **Try to work on the remove functionality.**

HashMap
========

How does hashmap works internally

1. **Default hashmap is of size 16 and the loading factor is 0.75**
2. **What is loading factor ?**

   The Load factor is a measure that decides when to increase the HashMap capacity to maintain the get() and put() operation complexity of O(1). The default load factor of HashMap is 0.75f (75% of the map size)

   We insert the first element, the current load factor will be 1/16 = 0.0625. Check is 0.0625 > 0.75 ? The answer is No, therefore we don't increase the capacity.

Next we insert the second element, the current load factor will be 2/16 = 0.125. Check is 0.125 > 0.75 ? The answer is No, therefore we don't increase the capacity.

Similarly, for 3rd element, load factor = 3/16 = 0.1875 is not greater than 0.75, No change in the capacity.

4th element, load factor = 4/16 = 0.25 is not greater than 0.75, No change in the capacity.

5th element, load factor = 5/16 = 0.3125 is not greater than 0.75, No change in the capacity.

6th element, load factor = 6/16 = 0.375 is not greater than 0.75, No change in the capacity.

7th element, load factor = 7/16 = 0.4375 is not greater than 0.75, No change in the capacity.

8th element, load factor = 8/16 = 0.5 is not greater than 0.75, No change in the capacity.

9th element, load factor = 9/16 = 0.5625 is not greater than 0.75, No change in the capacity.

10th element, load factor = 10/16 = 0.625 is not greater than 0.75, No change in the capacity.

11th element, load factor = 11/16 = 0.6875 is not greater than 0.75, No change in the capacity.

12th element, load factor = 12/16 = 0.75 is equal to 0.75, still No change in the capacity.

13th element, load factor = 13/16 = 0.8125 is greater than 0.75, at the insertion of the 13th element we double the capacity.

Now the capacity is 32

3. **Whenever we tried to add element to hashmap that is map.put**
   Internal implementation of PUT method
4. **First find out hash of the key**
5. **Then divide it by the 16 buckets which gives the index**
6. **Now at this index store the element in the form of Node**
7. **Node will contain hash, key , value and address of next node**
8. **If it happens that index of 2 keys are same then it is referred as collision**
9. **In case collision hash map follows linked list approach**
10. **At index first node will contain the address next node**
11. **What happens when you try to retrieve the element-> first system finds the hash of the key then find out index by dividing the hash /16**
    Now it compares the hash at the index. If there are multiple nodes present then it checks hash of each and every node until it is matched
    Once it is matched it checks the key. If key also matches the retrives the value and returns it
12. **If there are multiple nodes a given index this process may introduce performance issue**
13. **To get rid of this we go for Tree in hash map from 1.8**
14. **Difference between hashmap and linked hashmap is**
    Linked hashmap is ordered -> it will give the element in the insertion order
    Whereas hash map doesn't guarantee the order
15. **Try putting user defined employee object and see if those are equal.**

Set
===========

1. **Set internally uses the Map structure**

2. **Where the map gets initialized in (hashset constructor)**
3. **Default capacity is 16 and loading factor 0.75**
4. **When the set is using the map internally, then what is the value for all the keys (OBJECT)--you can find final Object PRESENT=new Object();**
5. **Find why set doesn't allow duplicates, because it follows map structure and map doesn't allow duplicates**
6. **Question is why map doesn't allow duplicate keys**
7. **Hashset or hashmap doesn’t guarantee the order**
8. **Where linked hashmap and linked hash set guarantee the insertion order why because it uses double linked list approach and each node contain the address of next hence insertion order is preserved.**

Concurrent Hash Map
================================

1. **Map, Set and List whenever we try to iterate and trying add the element then these collection will throw the concurrent modification exception.**
   ```
   l.add(10);
   l.add(25);
   Iterator it=l.iterator();
   While(it.hasNext()){
   Sysout(it.next());
   l.add(45)
   }
   ```
2. **Here in the above program, whenever we add the element to collection internally java uses the modcount variable and it gets incremented whenever we try to perform the add or remove operation so when we try to call next() method (look at next() method in ArrayList or any other collection) it checks for the earlier mod count and expected mod count, if both are not matching it throws the concurrent modification exception**

To resolve this we go for concurrent Hash Map

1. **In Concurrent Hash Map lock acquires at segment level and it uses the Reentrant lock**
2. **While calling next(), it iterates over new collection not on the original collection hence concurrent modification exception is not thrown**

CopyOnWriteArrayList
===============================

1. **In order to have the concurrency in the list we go for Copy on write ArrayList**
2. **It has the lock in add and remove etc**
3. **It uses the reentrant lock**
4. **It doesn't throw concurrent modiofication exception because the iteration will happen on new array and not on the original array**
5. **Same currencny in set will be achieved with CopyOnWriteSet.**

Concurrency in Set can be achieved through CopyOnWriteArraySet
Concurrency in Map can be achieved through Concurrent Hash Map

Difference between Synchronized list vs Copy on write array list

1. **In Synchronized list the lock will acquire at entire list**
2. **Where as in copy on write array list the lock will acquire at segment or index level so that multiple threads can work at different indexes at a same time**

Linked List

14 July 2020 11:20

- **In linked list each node contains data and the address of next node**
- **Java provides the LinkedList class which implements the List interface**

  Like ArrayList class we can create the object for Linked list and add the elements. LinkedList class is present in java.util jar

  List<Integer> l= new LinkedList();
  l.add(14);
  l.add(17);
- **Difference between Array List and Linked List**
  - The major difference between Arraylist and Linked list regards to their structure. Arrays list are index based data structure where each element associated with an index. On the other hand, Linked list relies on references where each node consists of the data and the references to the previous and next element
  - Basically, an array is a set of similar data objects stored in sequential memory locations under a common heading or a variable name.
  - While a linked list is a data structure which contains a sequence of the elements where each element is linked to its next element. There are two fields in an element of linked list. One is Data field, and other is link field, Data field contains the actual value to be stored and processed. Furthermore, the link field holds the address of the next data item in the linked list

ArrayList	LinkedList
This class uses a dynamic array to store the elements in it. With the introduction of generics, this class supports the storage of all types of objects.	This class uses a doubly linked list to store the elements in it. Similar to the ArrayList, this class also supports the storage of all types of objects.
Manipulating ArrayList takes more time due to the internal implementation. Whenever we remove an element, internally, the array is traversed and the memory bits are shifted.	Manipulating LinkedList takes less time compared to ArrayList because, in a doubly-linked list, there is no concept of shifting the memory bits. The list is traversed and the reference link is changed.
This class implements a List interface. Therefore, this acts as a list.	This class implements both the List interface and the Deque interface. Therefore, it

	can act as a list and a deque.
This class works better when the application demands storing the data and accessing it.	This class works better when the application demands manipulation of the stored data.

- **If at all we have to create the user defined linked list or custom linked list then we should use the static class to define the data and the next node address like below**

  ```
  static class Node{
          private int data;
          Node next;//A is class A a
          Node(int data){
              this.data=data;
          }
      }
  ```

- **How to Add or insert element to custom linked list:**
  - In order to perform the insert first create the object to Node class by passing the value (it is like calling the single argument constructor) that take the head node and check if it is null , for the first time it will be null so assign the above created node object
  - For the next element onwards to insert the node, iterate the Node till you reach the null and then assign the element (i.e -> when you get node.next is null that means there is no next element since the next is null and it is not pointing to any other node)
  - (because in Linked list every node will contain the address of next node, when the node.next is null means there is no next node and this is the right place to insert the new node)
- **How to Count the Nodes ?**

  In order to count the number of nodes,

  Take a variable and initialize it to 0, and increment it until you get node.next is null. (because in Linked list every node will contain the address of next node, when the node.next is null means there is no next node)
- **How to get the element in Custom Linked List**

  Start the logic from first node, and compare the value or object sent in get method.

  By taking the next node repeat the logic until you get the value same.

When the node.next is null and still if we didn't find the element means, the object passed in the get method is not present in the Linked list

How to delete the Node

To delete the node, user has to provide which node to be deleted, i.e to the delete method user will mentioned which object he wants to delete

Iterate over the linked link till you reach Node.next ==null then find it, each time when this logic is performed, take the value of previous node and keep it in temp variable and when the user entered node matches with the node to be deleted then change the reference like.

Previous.next = n.next

Stack

15 July 2020 08:29

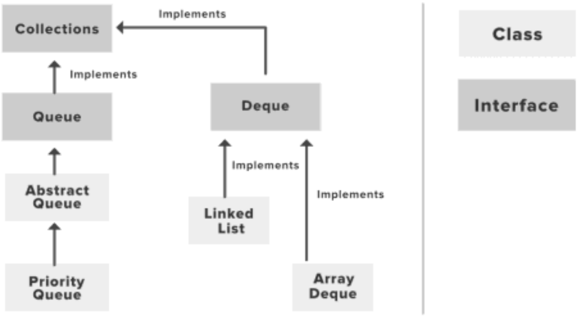
- Stack is a class and it is extending vector
- Methods available in stack are add, push, peek, remove, pop
- Why the add, remove methods are present, because stack class is extending vector class and hence vector becomes super class which has the add and remove methods hence these are available for stack class also
- Peek will give the top of the stack ->last element in stack
- Stack follows LIFO, or FILO
- Stack is a Synchronous data structure because it extends from vector. Since it is extending from vector, stack methods are also synchronous.
- Pop-> it will remove the top element from the stack and returns it
- Stack allows duplicated because internally it uses vector and the vector allows the duplicates
- If we perform peek and pop on empty stack then EmptyStackException will be thrown.
- To create the user defined stack
- First create the array and initialize the size
- If the user is trying to add elements more than the size declared then we should throw the over flow exception
- If the user is performing the pop and peek operation on the empty stack then we should throw underflow exception.
- ->every time when we add the element increment the top or index variable
- ->while poping or removing perform top-- or index--
- Default implementation of STACK uses Array data structure internally.
- We can also implement the STACK data structure using Linked List
- In order to implement the STACK data structure using object array, we have to follow the same process of creating the custom array list
- Stack Data Structure can be implemented using the Array or Linked List
- **In order to implement the STACK using the Array follow the below**.

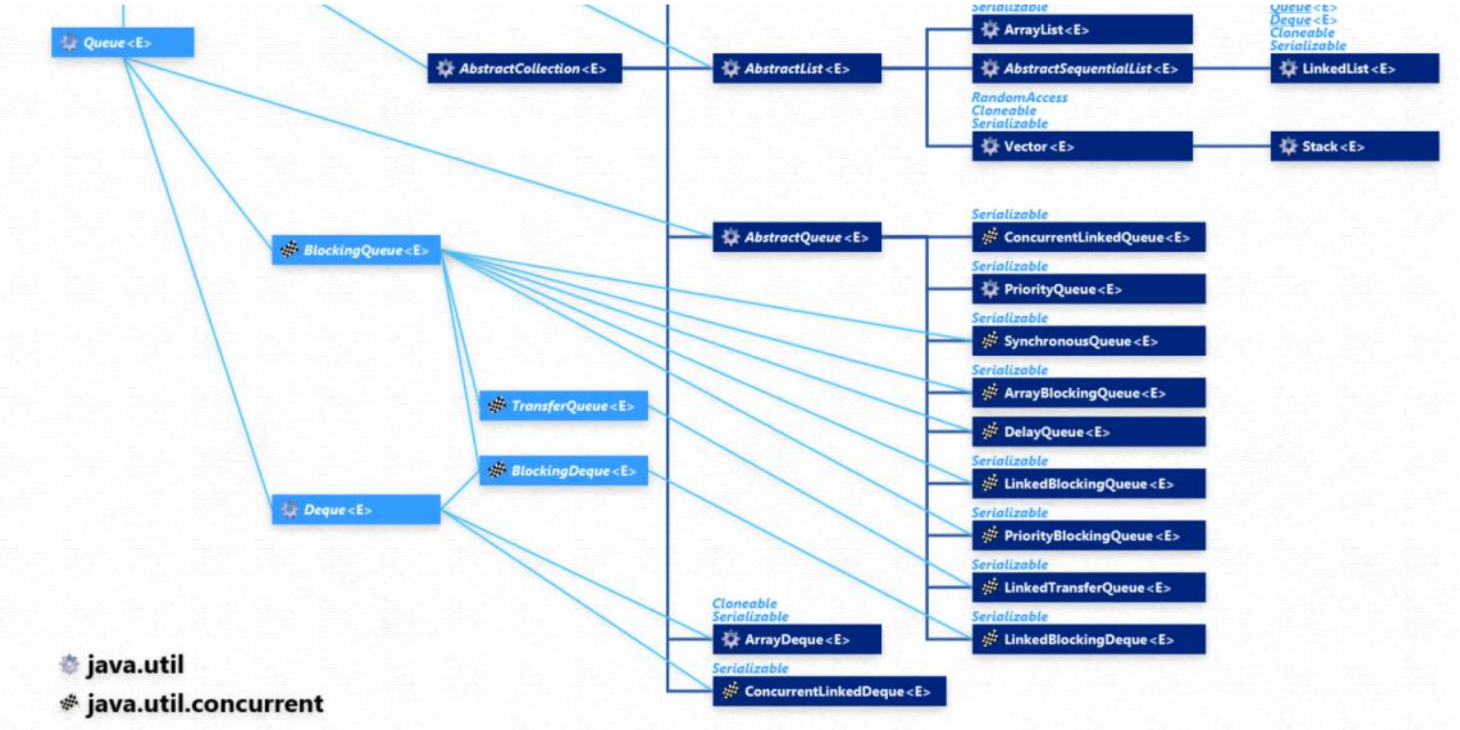
```
package com.ashokit.datastructures.collection.stack;

import java.util.Arrays;
import java.util.Stack;

public class CustomStack {

public static void main(String[] args) throws Exception {


            CustomStack s1= new CustomStack();
```

```
		s1.push(10);
		s1.push(20);
		s1.push(30);
		s1.push(40);
		s1.push(50);
		s1.push(60);
		System.out.println("********** Java CUSTOM STACK Details
		********************");
		System.out.println(s1);
		System.out.println(s1.peek());
		System.out.println(s1.pop());
		System.out.println(s1);
		System.out.println(s1.pop());
		System.out.println(s1);
		System.out.println(s1.pop());
		System.out.println(s1);
		Stack<Integer> s= new Stack<>();

		System.out.println("********** Java STACK Details ********************");
		s.add(10);
		s.add(20);
		s.add(30);
		s.add(40);
		s.add(50);
		s.add(60);
		s.add(70);
		s.add(80);
		s.add(90);
		s.add(90);
		s.add(100);

		System.out.println(s);
		System.out.println(s.peek());
		System.out.println(s.pop());
		System.out.println(s);
		System.out.println(s.pop());
		System.out.println(s);
		System.out.println(s.pop());
		System.out.println(s);
	}

	Object[] arr;
	int size;
	public CustomStack(){
		arr= new Object[10];
	}
	//
	public void push(Object obj) {
		checkCapacity();
		arr[size]=obj;//arr[size++]=obj;
		size=size+1;//5
	}

	private void checkCapacity() {
		if(size>=arr.length) {
			grow();
		}
	}
	private void grow() {
		int newSize=arr.length*2;
		Arrays.copyOf(arr, newSize);
	}

	public Object peek()throws Exception {
		if(size==0) {
			throw new Exception("Stack is Empty so cannot perform peek");
		}
		Object obj=arr[size-1];//array
		return obj;
	}
	//poll
	public Object pop()throws Exception {
		if(size==0) {
			throw new Exception("Stack is Empty so cannot perform POP");
		}
		Object obj=arr[size-1];//array
		//some other code to make
		arr[size-1]=null;
		size=size-1;
		return obj;
	}

	@Override
	public String toString() {
		StringBuffer s= new StringBuffer();
		s.append("[");
		for (int i = 0; i < arr.length; i++) {
			s.append(" "+arr[i]);
		}
		s.append(" ]");
		return s.toString();
	}
}
```

Queue

15 July 2020 08:29





➢ Queue Follows FIFO (First In First Out)

- ***offer()*** – Inserts a new element onto the *Queue*
- ***poll()*** – Removes an element from the front of the *Queue*
- ***peek()*** – Inspects the element at the front of the *Queue,* without removing it

- ➢ Classes that are implementing the Queue interface are PriorityQueue and LinkedList. These are not thread safe.
- ➢ Thread Safe Queue is PriorityBlockingQueue

➢ PriorityQueue

- Priority Queue class implements Queue
- Queue<Obj> queue = new PriorityQueue<Obj> ();
- In order to add an element in a queue, we can use the add() method. The insertion order is not retained in the PriorityQueue
- The elements are stored based on the priority order which is ascending by default.
- In order to perform the natural sorting order it uses the comparator internally.
- In order to remove an element from a queue, we can use the remove() method. If there are multiple such objects, then the first occurrence of the object is removed.
- Apart from that, poll() method is also used to remove the head and return it.

Dqueue (Double Ended Queue)

- *Deque* is short for **D**ouble-**E**nded ***Que***_ue_
- Deque provides methods to add, retrieve and peek at elements held at both the top and bottom
- In D Queue elements can be added to or removed from either the front (head) or back (tail).It is also often called a **head-tail linked list**

ArrayDqueue:

- **It is a special kind of a growable array that allows us to add or remove an element from both sides**
- An *ArrayDeque* implementation can be used as a *Stack* (Last-In-First-Out) or a *Queue*(First-In-First-Out)
- ArrayDeque class is likely to be faster than Stack when used as a stack.
- ArrayDeque class is likely to be faster than LinkedList when used as a queue

Operation	Method	Method throwing Exception
Insertion from Head	*offerFirst(e)*	*addFirst(e)*
Removal from Head	*pollFirst()*	*removeFirst()*
Retrieval from Head	*peekFirst()*	*getFirst()*
Insertion from Tail	*offerLast(e)*	*addLast(e)*
Removal from Tail	*pollLast()*	*removeLast()*
Retrieval from Tail	*peekLast()*	*getLast()*

Operation	**Method Name**
insert element at back	offerLast
insert element at front	offerFirst
remove last element	pollLast
remove first element	pollFirst
examine last element	peekLast
examine first element	peekFirst

Adding Elements

In order to add an element to the ArrayDeque, we can use the methods add(), addFirst(), addLast(), offer(), offerFirst(), offerLast() methods.

- add()
- addFirst()
- addLast()
- offer()
- offerFirst()
- offerLast()

Accessing the Elements

After adding the elements, if we wish to access the elements, we can use inbuilt methods like getFirst(), getLast(), etc.

- getFirst()
- getLast()
- peek()
- peekFirst()
- peekLast()

Removing Elements

In order to remove an element from a deque, there are various methods available. Since we can also remove from both the ends, the deque interface provides us with removeFirst(), removeLast() methods. Apart from that, this interface also provides us with the poll(), pop(), pollFirst(), pollLast() methods where pop() is used to remove and return the head of the deque. However, poll() is used because this offers the same functionality as pop() and doesn't return an exception when the deque is empty. These sets of operations are as listed below as follows:

- remove()
- removeFirst()
- removeLast()
- poll()
- pollFirst()
- pollLast()
- pop()

Iterating through the Deque

Since a deque can be iterated from both directions, the iterator method of the deque interface provides us two ways to iterate. One from the first and the other from the back. These sets of operations are listed below as follows:

- remove()
- iterator()
- descendingIterator()

ArrayBlockingQueue and LinkedBlockingQueue in Java Collection are the common implementations of the BlockingQueue interface.

- ArrayBlockingQueue is a bounded BlockingQueue backed by an array

  - bounded means the size of the Queue is finite and fixed. Once created, we cannot grow or shrink the size of the Queue. If we try to insert an element into a full Queue then it will result in the operation blocking. Similarly, if we try to take an element from an empty Queue, then also the operation will be blocked. ArrayBlockingQueue stores the elements in the Queue internally in the **FIFO (first-in-first-out)** order. The element at the head or front of the Queue is the oldest element of all the elements present in this queue. The element at the tail of this queue is the newest element of all the elements of this queue. The new elements are always inserted at the end or tail of the queue and the retrieval operations obtain elements at the head of the queue

LinkedBlockingQueue:

- LinkedBlockingQueue is a class in Java that implements the BlockingQueue interface. LinkedBlockingQueue is an **optionally-bounded** BlockingQueue backed by linked nodes. Here, optionally-bounded means the capacity given to LinkedBlockingQueue is bounded, otherwise, it will be unbounded. The capacity can be given as a parameter to the constructor of LinkedBlockingQueue. The capacity, if unspecified, is equal to **Integer.MAX_VALUE**

- inked queues typically have higher throughput than array-based queues but less predictable performance in most concurrent applications

Sr. No.	Key	ArrayBlockingQueue	LinkedBlockingQueue
1	Basic	It is backed by Array	It is backed by Linked list
2	Bounded	It is bounded array queue . Therefore Once created, the capacity cannot be changed	It is unbounded queue
3	Throughput	It has lower throughput than Linked queues queues	Linked queues have higher throughput than array-based queues
4.	Lock	It uses single-lock double condition algorithm	It has putLock for inserting element in the queue and takeLock for removing elements from the queue

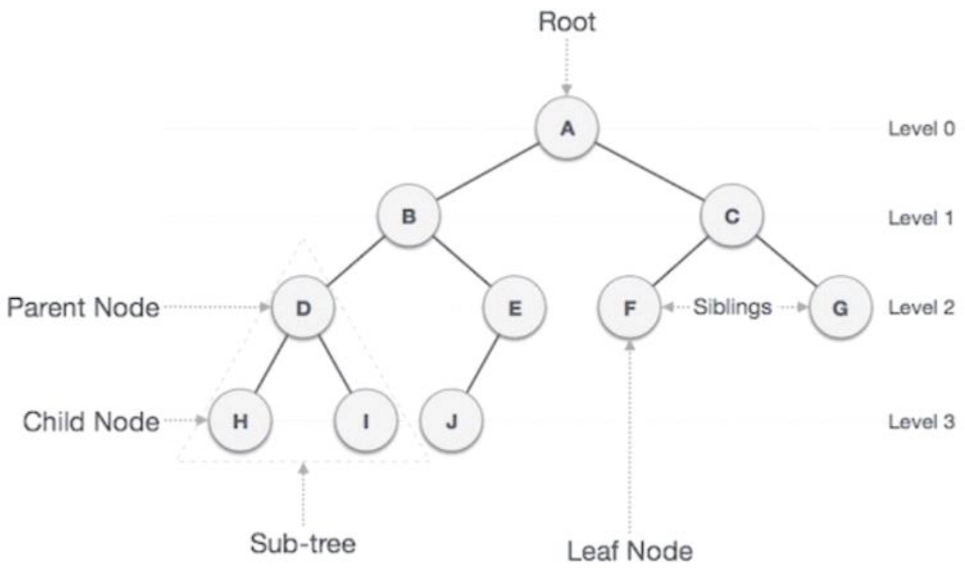
- LinkedBlockingQueue — an optionally bounded FIFO blocking queue backed by linked nodes
- ArrayBlockingQueue — a bounded FIFO blocking queue backed by an array
- PriorityBlockingQueue — an unbounded blocking priority queue backed by a heap
- DelayQueue — a time-based scheduling queue backed by a heap
- SynchronousQueue — a simple rendezvous mechanism that uses the BlockingQueue interface

Tree

16 July 2020 09:18

Java does not have a built in tree data structure

- **Trees** are non-linear data structures. They are often used to represent hierarchical data. For a real-world example, a hierarchical company structure uses a tree to organize
- In Java Tree, each node except the root node can have one parent and multiple children. Root node doesn't have a parent but has children. We will create a class *Node* that would represent each node of the tree. *Node* class has a data attribute which is defined as a generic type. *Node* class also has the reference to the *parent* and the List of *children*.



What is Tree ?

- Tree is a data structure in which root node will be present and it can have left and right child

Full binary Tree

- is a tree which has all the nodes (each root should have left and right nodes)
- If the tree height is h then 2^h nodes should be present at h height

Complete Binary Tree-

- Is a tree in which it should be full binary tree up to height n-1 and at nth level tree should be filled from Left

Construction of Tree:

How to find the Parent of Node: if we make the tree in array form

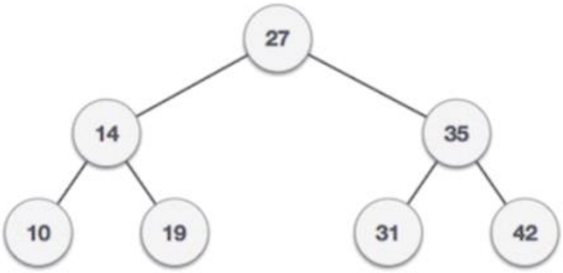
- Then i/2 will give the parent of tree provided if you array starts from 1 (I is the index starts from 1)
- **How to find right child and left child of Node**
- if we make the tree in array form
- Then (2*i)+1 will give the right child of a node and 2*i will provide the left child of a node

Retrieving the Nodes present at height k of Tree-

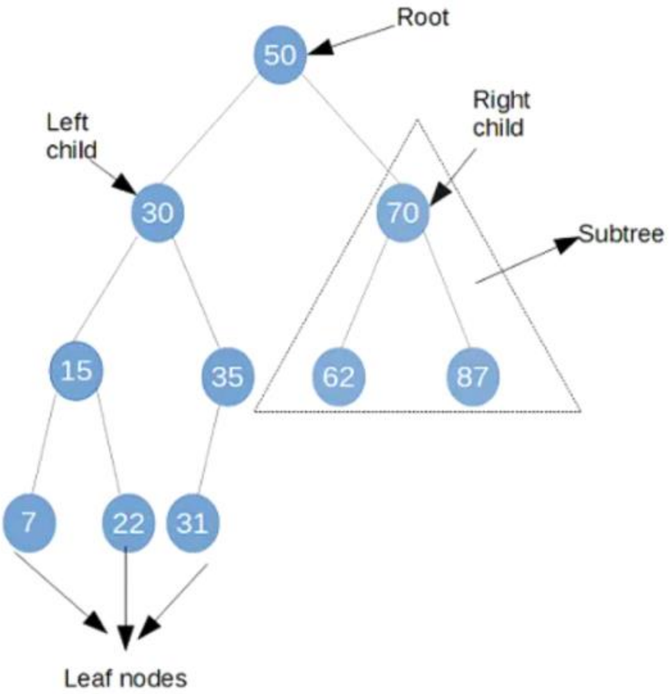
- Pass the Root node and the height of K and for every iteration or every recursion decrement the K value and when the K becomes zero get the data and display

Binary Search Tree Representation

- Binary Search tree exhibits a special behaviour. A node's left child must have a value less than its parent's value and the node's right child must have a value greater than its parent value.



Binary Search tree



- **Binary tree traversal**
  When you traverse a tree you visit each node in a specified order. The order that can be used for traversal are-
  - **Inorder traversal**
  - **Preorder traversal**
  - **Postorder traversal**

- **Binary tree Inorder traversal Java program**
  - Logic for Inorder traversal of binary search tree is as follows-
  - Recursively traverse the left subtree
  - Visit the root node
  - Recursively traverse the right subtree
  - Note that inorder traversal of the binary search tree visit the nodes in ascending order so inorder traversal is also used for tree sort.
    // For traversing in order

```
public void inOrder(Node node){
  if(node != null){
    inOrder(node.left);
    node.displayData();
    inOrder(node.right);
  }
}
```

Binary tree Preoder traversal Java program

- Logic for preorder traversal of binary search tree is as follows-
- Visit the root node
- Recursively traverse the left subtree.
- Recursively traverse the right subtree

```
// Preorder traversal
public void preOrder(Node node){
  if(node != null){
    node.displayData();
    preOrder(node.left);
    preOrder(node.right);
  }
}
```

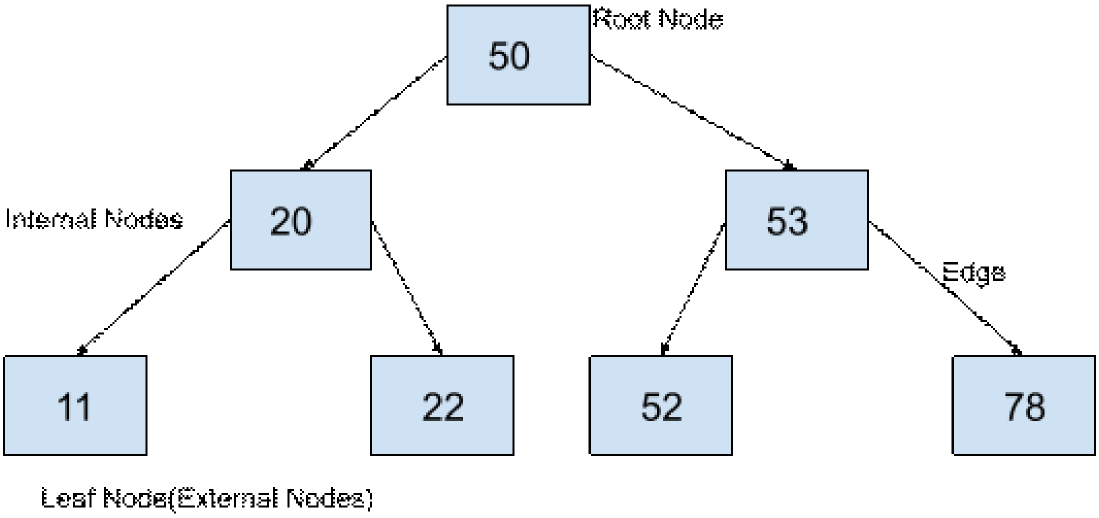
Binary tree Postorder traversal Java program

- Logic for postorder traversal of binary search tree is as follows-
- Recursively traverse the left subtree.
- Recursively traverse the right subtree
- Visit the root node
- // Postorder traversal

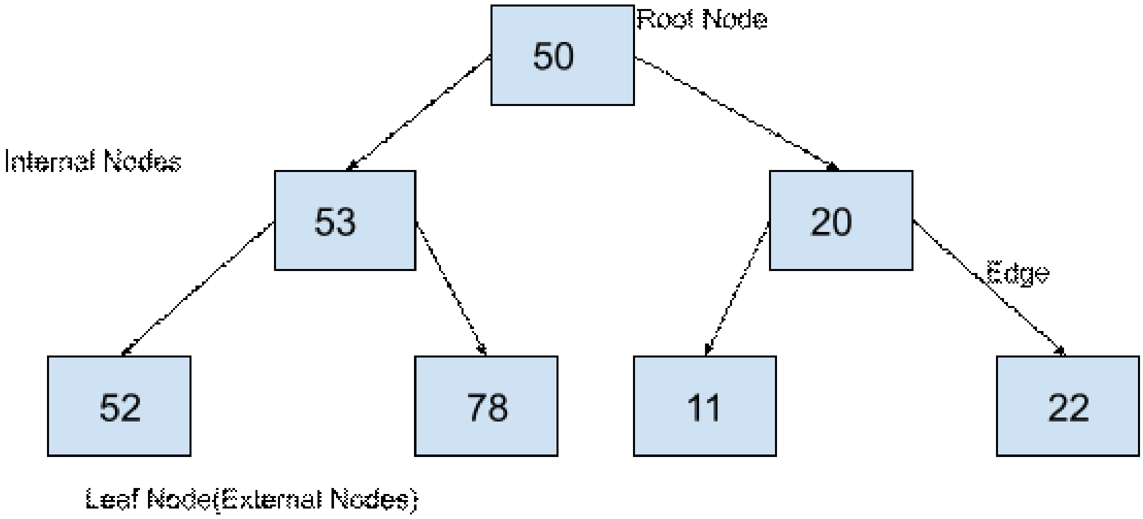
```
public void postOrder(Node node){
  if(node != null){
    postOrder(node.left);
    postOrder(node.right);
    node.displayData();
  }
}
```

What is the mirror image of a binary tree?

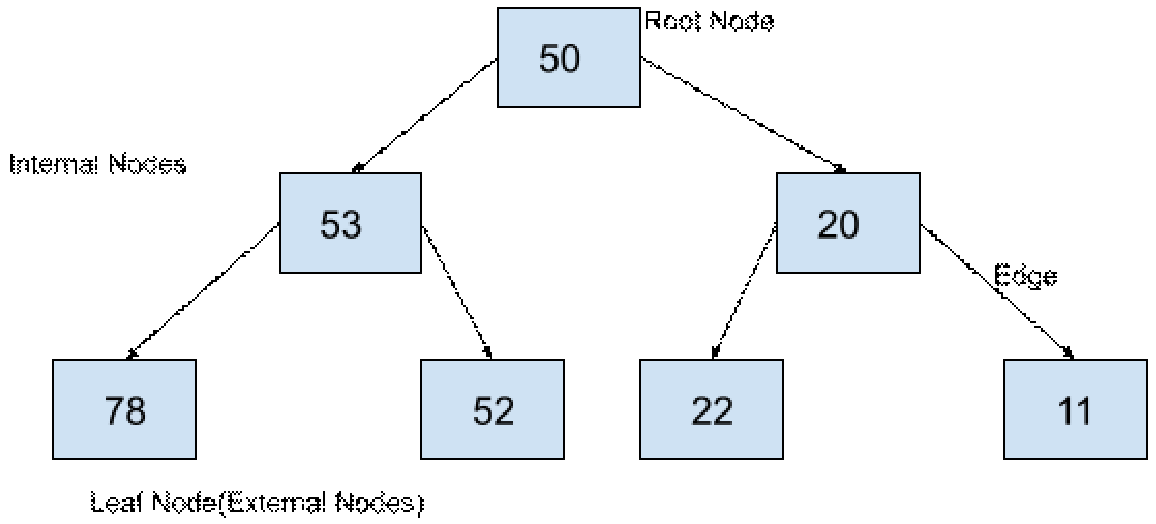
- Mirror image of a binary tree is another binary tree which can be created by swapping left child and right child at each node of a tree. So, to find the mirror image of a binary tree, we just have to swap the left child and right child of each node in the binary tree. Let us try to find the mirror image of the following tree.

➢ 

➢ At root , we will swap the left and right children of the binary tree. In this way, 20,11, and 22 will come into the right subtree of the binary tree and 53,52, and 78 will come to the left of the binary tree as follows.

➢ 

➢ Then we will move to the next level and swap the children of 53. Here, 78 will become the left child of 53 while 52 will become the right child of 53. Similarly we will swap the left child and right child of 20. In this way, 22 will become the left child of 20 while 11 will become the right child of 20. The output binary tree after swapping nodes at this level will be as follows.

- 

- Now we will move to the next level. At this level, all the nodes are leaf nodes and have no children. Due to this there will be no swapping at this level and the above image is the final output.

Algorithm to find mirror image of a binary tree

- As we have seen above, We can find the mirror image of a binary tree by swapping the left child and right child of each node. Let us try to formulate the algorithm in a systematic way.
- In the last example, At the second level, each node has only leaf nodes as their children. To find the mirror image at a node at this level, we have just swapped the left and right child at each node at this level. At the root node, we have swapped its both subtrees. As we are swapping each subtree (leaf nodes are also subtree), we can implement this algorithm using recursion.
- The algorithm for finding a mirror image of a binary tree can be formulated as follows.
  - ® Start from the root node.
  - ® Recursively find the mirror image of the left subtree.
  - ® Recursively find the mirror image of the right subtree.
  - ® Swap the left and right subtree.

Tree Set and Tree Map

- In order to add an element to the TreeSet, we can use the add() method. However, the insertion order is not retained in the TreeSet. Internally, for every element, the values are compared and sorted in ascending order. We need to keep a note that duplicate elements are not allowed and all the duplicate elements are ignored. And also, Null values are not accepted by the TreeSet.
- Tree set doesn't follow insertion order. It follows natural sorting order.
- Internally tree set uses comparator. If nothing is mentioned, it follows the comparator provided in that class
- For example. If I create the object below for Tree Set.
  TreeSet<Integer> t= new TreeSet<Integer>();
  During the above object creation, I haven't mentioned any comparator, but internally The wrapper class Integer is implementing the comparable interface so the logic of comparable interface in all the wrapper classes is natural sorting order. Hence the elements are arranged in natural sorting order.
  Tree Set internally uses Tree Map as data Structure.

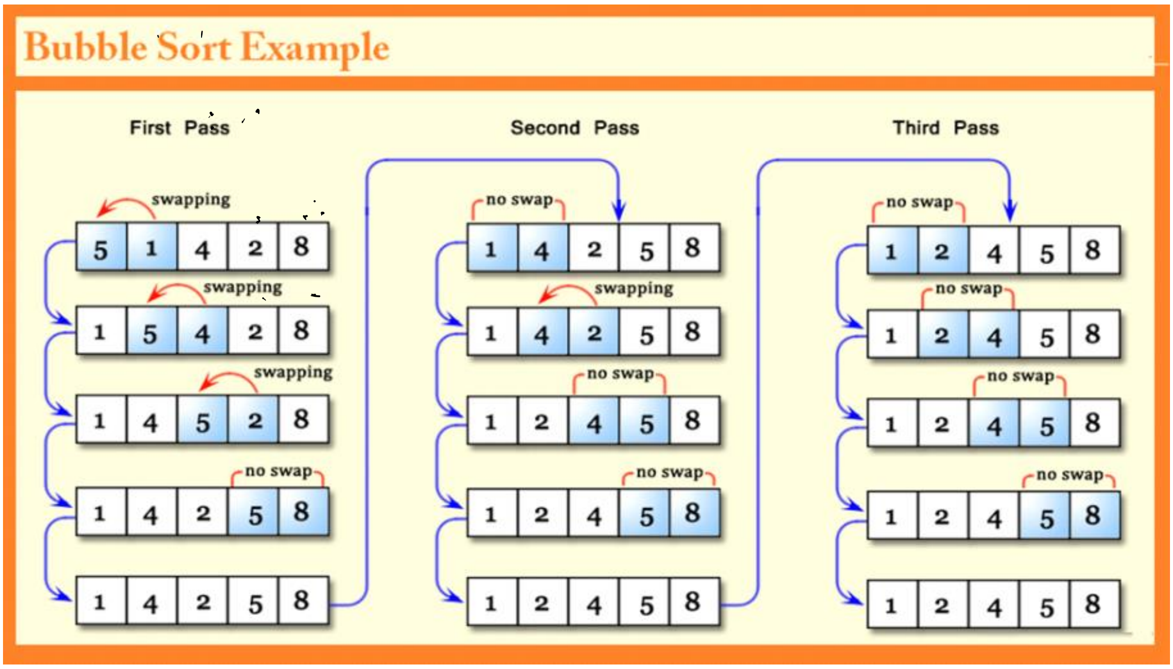
- Tree Map: tree map also arranges the keys in natural sorting order whereas hash map doesn’t follow any order.

- Tree map uses the comparator internally
- Tree Map follows the tree data structure internally to store the elements.

Bubble Sort

14 June 2020 11:42

1. **In bubble sort each element is compared with the next element and if it is greater then do the swap.**
2. **After the first pass the maximum element will be found at the last position.**
3. **After second pass the 2nd maximum element will be found at the last second position and so on**
4. **In bubble sort we are iterating over the array for n square times in worst scenario hence the time complexity is 0(n2).**
5. **Internal loop of bubble sort, uses n-i-1 because after completion of ith element max element is found at last position hence no need to compare to that element**
6. **In order to know whether the input array is sorted or not, use the Boolean variable and if the code doesn't enter the swapping logic, it means array is in sorted order, so break the loop.**



Insertion Sort

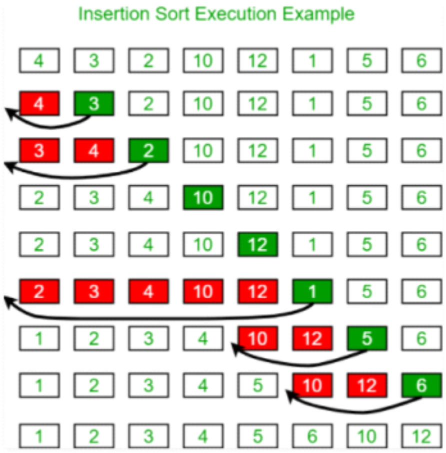
12 July 2020 09:30

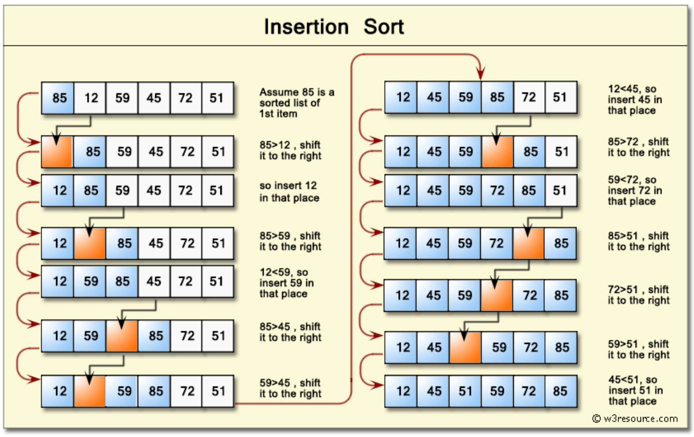
- **Insertion sort is a simple sorting algorithm that works similar to the way you sort playing cards in your hands**
- The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part.

- **Algorithm**
  - To sort an array of size n in ascending order:
  - 1: Iterate from arr[1] to arr[n] over the array.
  - 2: Compare the current element (key) to its predecessor.
  - 3: If the key element is smaller than its predecessor, compare it to the elements before. Move the greater elements one position up to make space for the swapped element.

Insertion sort makes use of sorted and unsorted array.
After each iteration left side array will be sorted and right side will be un sorted
For example if we are doing 3 rd iteration then the elements present in indexes 0,1,2 will be sorted this is called sorted array
And the 3rd element needs to be compared with values present at index 0,1,2 and then it determines the correct position for 3rd and element and insert the 3rd element. Hence it is called insertion sort
Since the element to be sorted should be compared with all the sorted list of left side we use while loop
Instead we can also use for loop

Time complixiety is 0(n2)



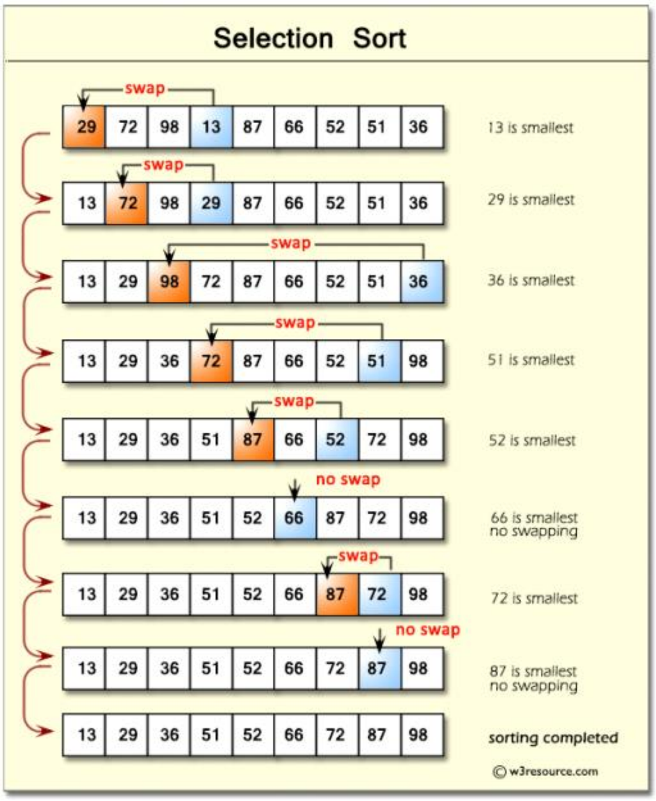

Insertion Sort
85 12 59 45 72 51
Assume 85 is a sorted list of 1st item
85 59 45 72 51
85>12 , shift it to the right
12 85 59 45 72 51
so insert 12 in that place
12 85 45 72 51
85>59 , shift it to the right
12 59 85 45 72 51
12<59, so insert 59 in that place
12 59 85 72 51
85>45 , shift it to the right
12 59 85 72 51
59>45 , shift it to the right
12 45 59 85 72 51
12<45, so insert 45 in that place
12 45 59 85 51
85>72 , shift it to the right
12 45 59 72 85 51
59<72, so insert 72 in that place
12 45 59 72 85
85>51 , shift it to the right
12 45 59 72 85
72>51 , shift it to the right
12 45 59 72 85
59>51 , shift it to the right
12 45 51 59 72 85
45<51, so insert 51 in that place
© w3resource.com

Selection Sort

12 July 2020 09:31

- In the selection sort, the intend is to find out at which index or position the minimum value of an array is present then replace it with Ith position
- In Selection sort, after first iteration the minimum values in the entire array will be at first position
- After second iteration the second minimum values will be in the second position and so on
- We make use of min_index variable and compare the elements with min_index and if there any value that is present < min index then make that position as min index
- Once the entire array is compared and sort the min index with the I value (outer for loop variable), to perform this SWAPPING, we use swap logic outside of the inner for loop.

- Time complixiety is (n2)



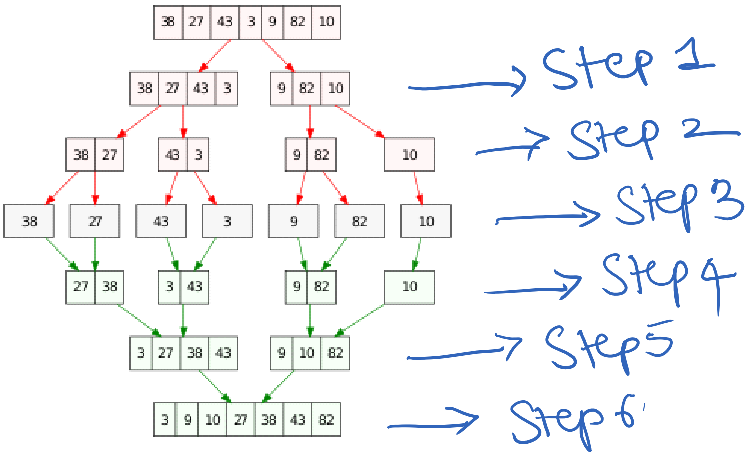
Merge Sort

14 July 2020 08:18

1. **Merge sort follows the divide and conquer algorithm.**
2. **The idea behind the merge sort is two way merging**
3. **In order to merge the 2 sorted lists, the pre requisite is 2 lists should be sorted and then using the below code we can merge two lists into 1 list**

```
int a[]= {9,10,15,16,19};
int b[]= {11,12,14,18,21,31,33};


int c[]=new int[a.length+b.length];
            int i=0,j=0,k=0;
            int m=a.length,n=b.length;

            while(i<m && j<n) {
                  if(a[i]<b[j]) {//9<11
                        c[k]=a[i];
                        i++;
                  }else {
                        c[k]=b[j];
                        j++;
                  }
                  k++;
            }

            while(i<m) {
                  c[k]=a[i];
                  i++;
                  k++;
            }
            while(j<n) {
                  c[k]=b[j];
                  j++;
                  k++;
            }
```

4. **In the merge sort the given array is not sorted so to make it sorted, we will divide the array into two by finding the mid element (mid= l+r/2)**
5. **But again after dividing into 2 arrays, the left and right array are not sorted so I cannot apply the two way merging so**
6. **Split the array unless u get the single element. This is called divide and conquer.**
7. **The entire array will be divided in to half until the single element is reached**
8. **Once the single element is reached then we can say that now all the single element arrays are sorted now we can apply the two merging.**
9. **So merge the 2 single element arrays into 1 list then follow the same until you get sorted array**
10. **In the below diagram, total 7 elements are present which are not sorted**
    So first divide into 2 arrays -> still not sorted - > Look at step 1
    So divide further -> Still not sorted ->Look at step 2
    So divide further -> Still not sorted ->Look at step 3
    Now each element is single and sorted
    Apply two way merging for first 2 elements ->look at step 4
    Again apply 2 way merging for arrays which has 2 elements -> Look at step 5
    Again apply 2 way merging for arrays which has 4 elements -> Look at step 6 --Result



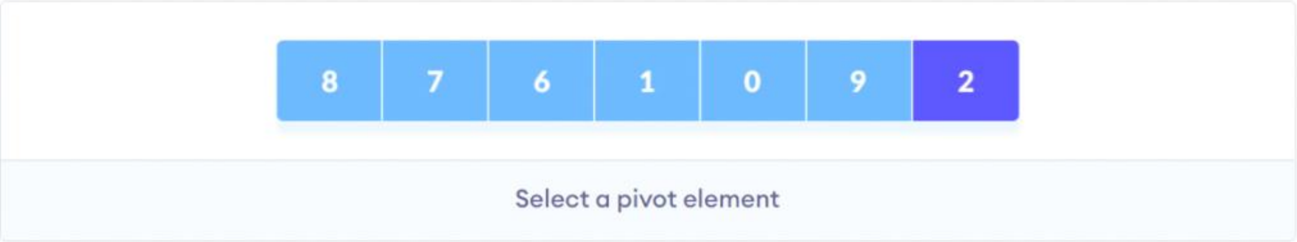
Procs:

1. Preferred for large size list
2. Suitable for linked list
3. Supports external sorting

Quick Sort

07 February 2022 12:02

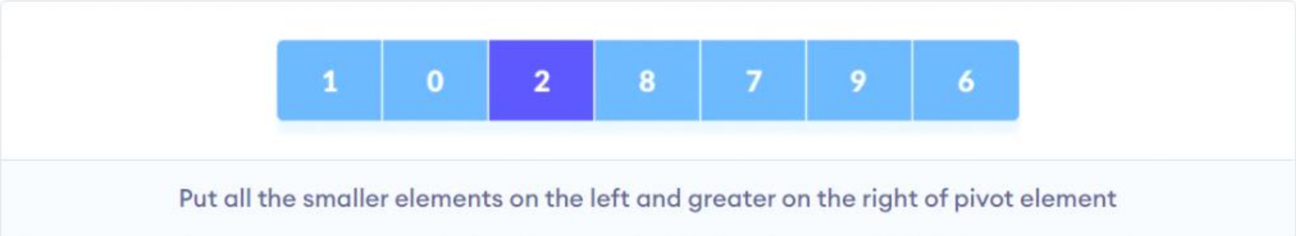
- Quicksort algorithm is based on the divide and conquer approach where an array is divided into subarrays by selecting a pivot element.
- While dividing the array, the pivot element should be positioned in such a way that elements less than pivot are kept on the left side and elements greater than pivot are on the right side.
- The same process is continued for both left and right subarrays. Finally, sorted elements are combined to form a sorted array.
  - Select the Pivot Element
  - There are different variations of quicksort where the pivot element is selected from different positions. Here, we will be selecting the rightmost element of the array as the pivot element. This type of partition is called as Lomuto's Partition
  - **Selected PIVOT element is 2**



8	7	6	1	0	9	2

Select a pivot element

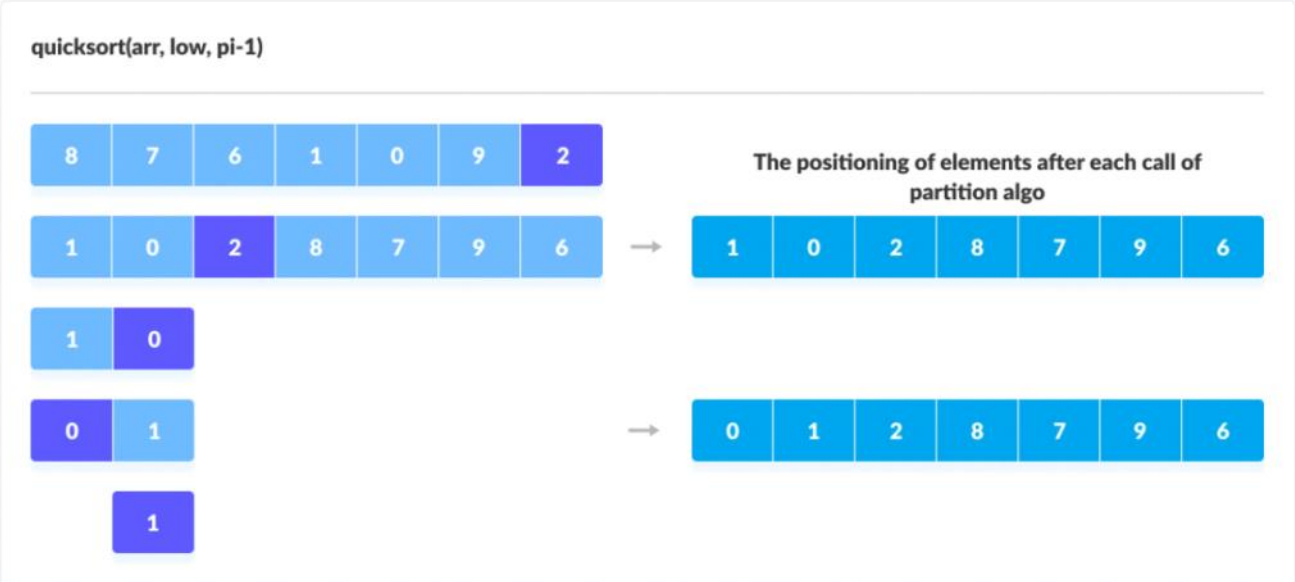
- Re Arrange the array
- Compare the elements from starting of the array with pivot element and if there are any element which is less than PIVOT element then replace with starting position. Continue this logic until end of array.
- Now the elements of the array are rearranged so that elements that are smaller than the pivot are put on the left and the elements greater than the pivot are put on the right.



1	0	2	8	7	9	6

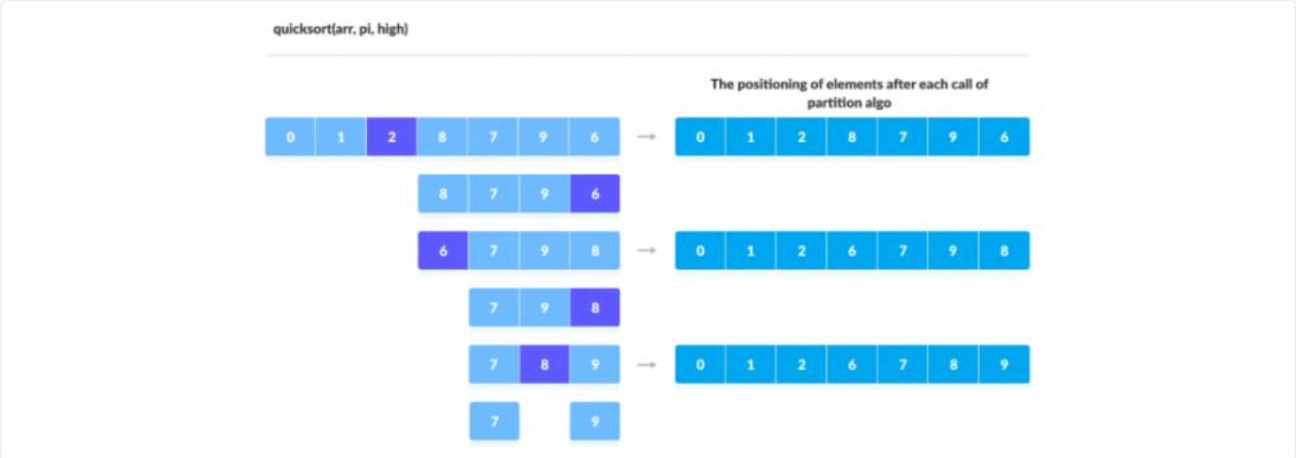
Put all the smaller elements on the left and greater on the right of pivot element

- Now from the above diagram we can say that the elements before 2 are less and elements after 2 are higher but those are not sorted. Even though if we sort those elements, position of 2 is not going to change.
- That means we can say that 2 is SORTED in the entire array.
- Now apply the same logic towards the left side array and towards the right side array.
- In this way Quick sort follows the Divide conquer algorithm
- **Left side apply like this**


quicksort(arr, low, pi-1)
8 7 6 1 0 9 2
1 0 2 8 7 9 6
1 0
0 1
1
The positioning of elements after each call of partition algo
1 0 2 8 7 9 6
0 1 2 8 7 9 6

Sorting the elements on the left of pivot using recursion

Right side Array will be like this



Select pivot element of in each half and put at correct place using recursion

Hoare's Partition :

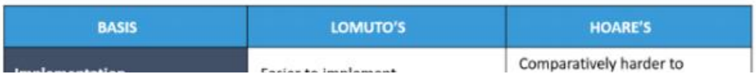
10, 16,8, 12,15, 6,3,9,5

Take 0th element as PIVOT
Take 2 variables I and j
Increment I from starting position untill you get value greater than pivot
Decrement J from end of array untill you get element less than pivot

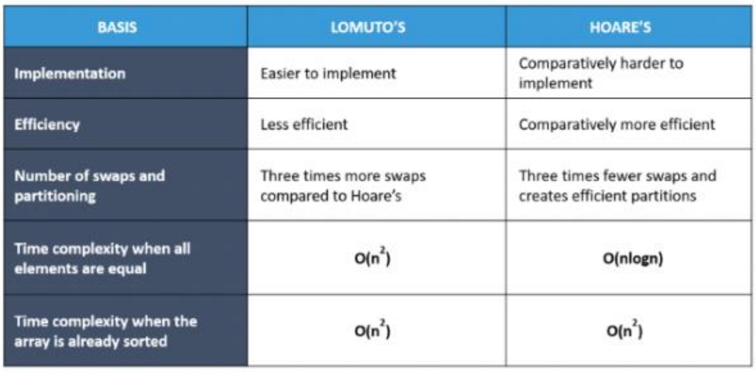
Whenever you see above conditions are satisfied then swap I and j.

Continue the logic
When I crosses J that means i<j then stop logic and replace low with J

Differences between LOMUTO's and HOARE's



BASIS	LOMUTO'S	HOARE'S
Implementation	Easier to implement	Comparatively harder to

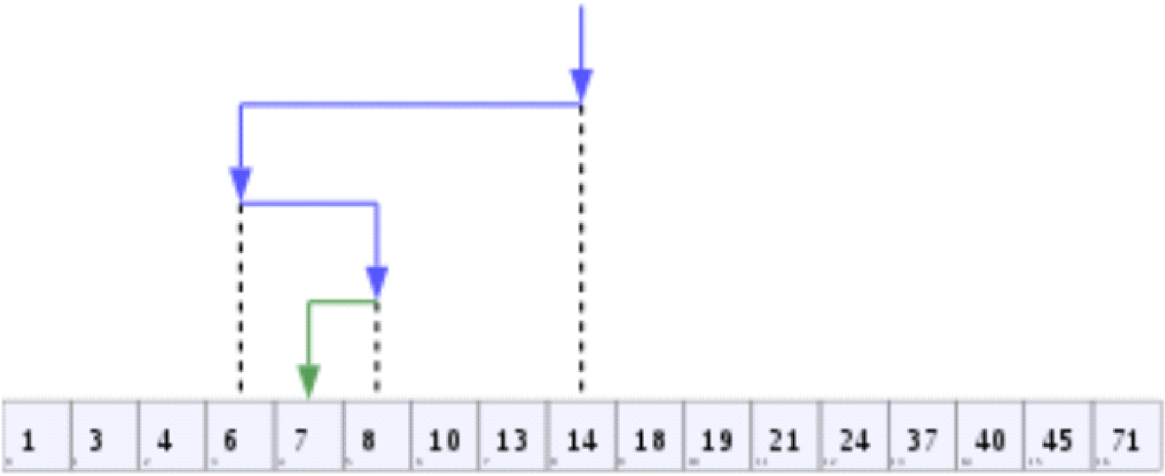


BASIS	LOMUTO'S	HOARE'S
Implementation	Easier to implement	Comparatively harder to implement
Efficiency	Less efficient	Comparatively more efficient
Number of swaps and partitioning	Three times more swaps compared to Hoare's	Three times fewer swaps and creates efficient partitions
Time complexity when all elements are equal	$O(n^2)$	O(nlogn)
Time complexity when the array is already sorted	$O(n^2)$	$O(n^2)$

Binary Search

04 December 2021 15:02

- To perform the Binary search the pre requisite is that array should be in the ascending order
- **Algorithm of Binary search**
  - Name the element to be found as **'X'**
  - Binary Search follows the left, right and middle approach
  - Left is nothing but 0th index and right is nothing but length of array-1
  - And middle will be calculated by the formula (l+r)/2;
  - Once the middle index is found from the array compare the middle element to the element
  - Compare **X** with the middle element.
  - If **X** matches with the middle element, we return the mid index.
  - Else If **X** is greater than the mid element, then x can only lie in the right half subarray after the mid element. So we recur for the right half, continue the same steps from 1, till left<=right
  - Else (**X** is smaller) recur for the left half, continue the same steps from 1, till left<=right



Visualization of the binary search algorithm where 7 is the target value

- **The best case of Binary Search occurs when:**

  The element to be search is in the middle of the list

  In this case, the element is found in the first step itself and this involves 1

comparison.
Therefore, Best Case Time Complexity of Binary Search is O(1).

- **The worst case of Binary Search occurs when:**
  The element is to search is in the first index or last index
  In this case, the total number of comparisons required is logN comparisons.
  Therefore, Worst Case Time Complexity of Binary Search is O(logN).

Problem with (L+r)/2

- The above looks fine except one subtle thing, the expression “m = (l+r)/2”. It fails for large values of l and r. Specifically, it fails if the sum of low and high is greater than the maximum positive int value (2^31 – 1). The sum overflows to a negative value, and the value stays negative when divided by two
- In order to avoid we go for the below solution
  mid = l + (r - l) / 2;

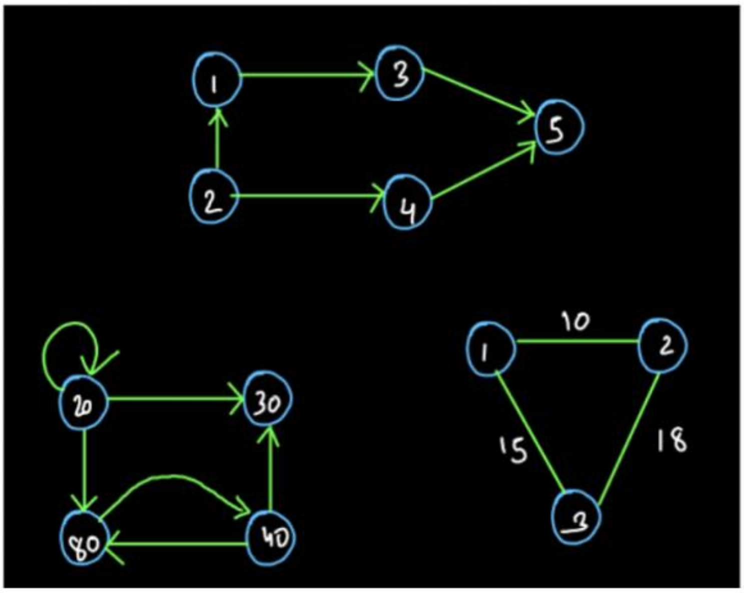
Graphs

16 January 2022 16:24

A Graph is a non-linear data structure consisting of nodes and edges. The nodes are sometimes also referred to as vertices and the edges are lines or arcs that connect any two nodes in the graph. More formally a Graph can be defined as,

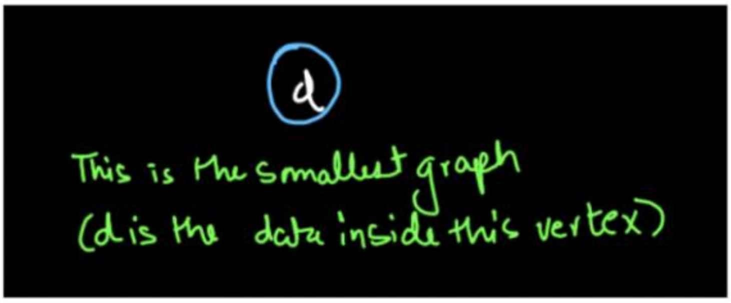
A Graph consists of a finite set of vertices(or nodes) and set of Edges which connect a pair of nodes

A graph is a set of vertices and edges



So, we can define the graph G = (V,E) where V is the set of vertices and E is the set of edges. As in the above diagram, the circles in which the data is stored are the vertices of the graphs and the lines connecting them together are called edges of the graphs. Now the question arises if a graph is a set of vertices and edges then what can be the minimum number of vertices and edges a graph can have?

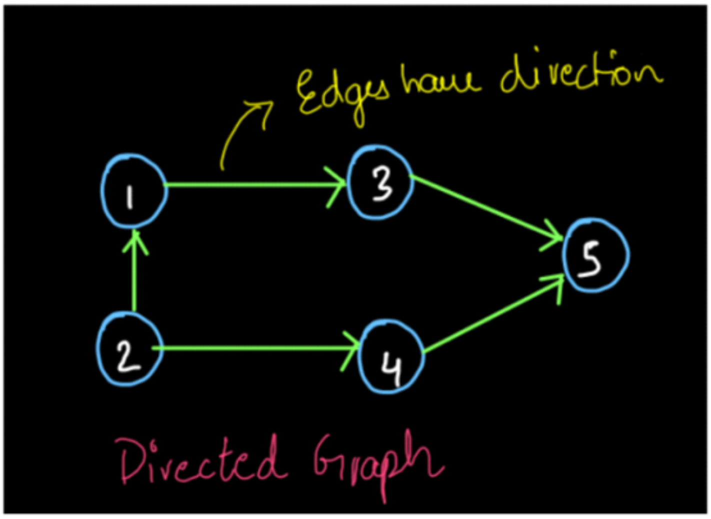
See, a graph is a data structure. This means that it is a way of storing data. Who stores data in a graph? As you saw in the above diagram, a vertex stores the data in the graph. So, presence of at least one vertex is necessary whereas presence of an edge is not. Therefore the smallest graph is a graph with only one vertex and zero edges


d
This is the smallest graph
(d is the data inside this vertex)

So, the set V i.e. the set of vertices for a graph G cannot be empty but the set E can be empty

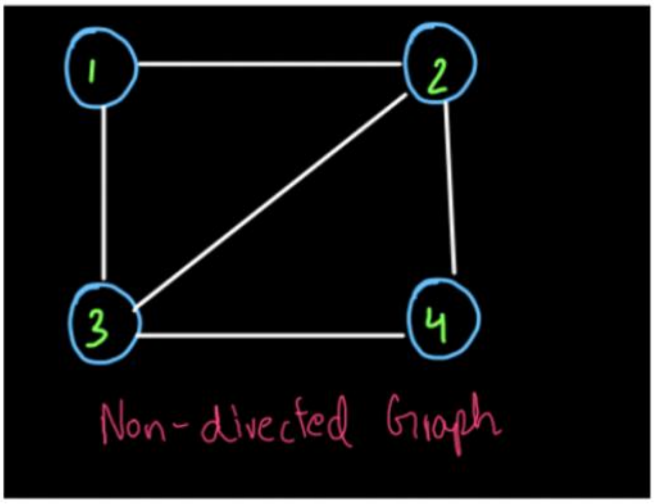
Directed Graph:

If the edges of a graph has directions then the graph is known as directed graph



Non-directed Graph:

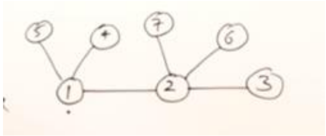
A graph whose edges do not have direction is called an undirected/non-directed graph. They are assumed to be bi-directional.



Graph Traversal

1. **Visiting a vertex**
2. **Exploration of Vertex : If I am at particular vertex then visiting all adjacent vertex is called as exploration of vertex.**

1. **Visit the vertex**
2. **Explore the vertex -> visit it's adjacent vertex in any order**



In the above diagram, 1 is vertex

so **BFS** is : **1,2,4,5,7,6,3**

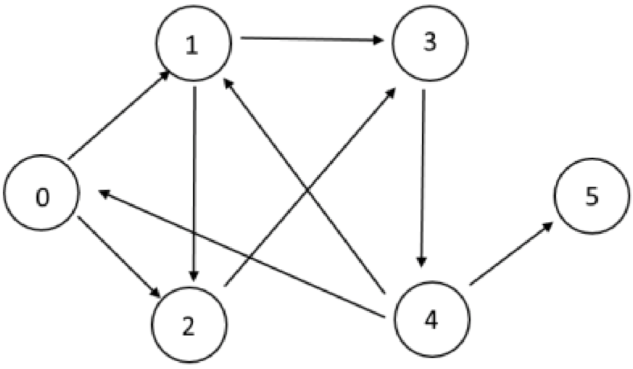
Explanation: After visiting the vertex, explore the adjacent vertex. So here adjacent vertex are 2,4,5 so write 2,4,5 after 1. So for 1 vertex all adjacent vertex is visited so explore any other vertex.
Take is as 2 now
Adjacent to 2 are 7,6,3 so Write these above
All vertex are visited and nothing is remaining.

DFS is **1, 2,7,6, 3 ,5,4**

Explanation:
Start with vertex 1 and explore the vertex which 2 (that is adjacent to 1).
So now don't visit the other adjacent vertex which are 4,5
Complete the exploration of 2, adjacent to 2 are 7,6,3 and take any one
7-> explore 7, there are no adjacent vertices so go to other vertex 6, there are no vertex so write it and go to 3
Now comeback to 1 and explore other vertex 5 and 5 doesn't have adjacent so go to 4

Graph – Depth First Traversal

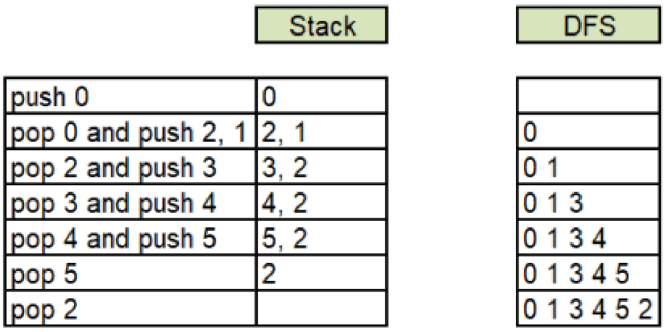
Objective – Given a graph, do the depth first traversal(DFS).
What is depth-first traversal– Depth-first search (DFS) is an algorithm for traversing or searching tree or graph data structures. One starts at the root (selecting some arbitrary node as the root in the case of a graph) and explores as far as possible along each branch before backtracking. Source – Wiki
Example:



Depth First Traversal - 0 1 3 4 5 2

Approach:

- Use Stack.
- First add the Starting Node to the Stack.
- Pop out an element from Stack and add all of its connected nodes to stack.
- Repeat the above two steps until the Stack is empty.
- Below is a walk through of the graph above.



	Stack	DFS
push 0	0	
pop 0 and push 2, 1	2, 1	0
pop 2 and push 3	3, 2	0 1
pop 3 and push 4	4, 2	0 1 3
pop 4 and push 5	5, 2	0 1 3 4
pop 5	2	0 1 3 4 5
pop 2		0 1 3 4 5 2

Implementing Breadth-First Search in Java

There are multiple ways of dealing with the code. Here, we would primarily be sharing the process for a breadth first search implementation in java. Graphs can be stored either using an adjacency list or an adjacency matrix – either one is fine. In our code, we will be using the adjacency list for representing our graph. It is much easier to work with the adjacency list for implementing the Breadth-First Search algorithm in Java as we just have to move through the list of nodes attached to each node whenever the node is dequeued from the head (or start) of the queue.

The graph used for the demonstration of the code will be the same as the one used for the above example.

The implementation is as follows:-

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

class Graph
{
   private int V;                              //number of nodes in the graph
   private LinkedList<Integer> adj[];              //adjacency list
   private Queue<Integer> queue;                   //maintaining a queue

   Graph(int v)
   {
      V = v;
      adj = new LinkedList[v];
      for (int i=0; i<v; i++)
      {
         adj[i] = new LinkedList<>();
      }
      queue = new LinkedList<Integer>();
   }


   void addEdge(int v,int w)
   {
      adj[v].add(w);                          //adding an edge to the adjacency list (edges are bidirectional in this example)
   }

   void BFS(int n)
   {

      boolean nodes[] = new boolean[V];       //initialize boolean array for holding the data
      int a = 0;

      nodes[n]=true;
      queue.add(n);                  //root node is added to the top of the queue

      while (queue.size() != 0)
      {
         n = queue.poll();            //remove the top element of the queue
         System.out.print(n+" ");          //print the top element of the queue
      for (int i = 0; i < adj[n].size(); i++)  //iterate through the linked list and push all neighbors into queue
      {
        a = adj[n].get(i);
        if (!nodes[a])                    //only insert nodes into queue if they have not been explored already
        {
          nodes[a] = true;
          queue.add(a);
        }
      }
    }
  }

  public static void main(String args[])
  {
    Graph graph = new Graph(6);

    graph.addEdge(0, 1);
    graph.addEdge(0, 3);
    graph.addEdge(0, 4);
    graph.addEdge(4, 5);
    graph.addEdge(3, 5);
    graph.addEdge(1, 2);
    graph.addEdge(1, 0);
    graph.addEdge(2, 1);
    graph.addEdge(4, 1);
    graph.addEdge(3, 1);
    graph.addEdge(5, 4);
    graph.addEdge(5, 3);

    System.out.println("The Breadth First Traversal of the graph is as follows :");

    graph.BFS(0);
  }
}
```

Dynamic Programming

05 February 2022 15:24

- It is used to solve the optimization problems
- Breaks down the complex problem into simple sub problems
- Find the optimal solution to sub problems
- Store the results of sub problems
- Re use the sub problems

  Dynamic Programming is mainly an optimization over plain recursion. Wherever we see a recursive solution that has repeated calls for same inputs, we can optimize it using Dynamic Programming. The idea is to simply store the results of subproblems, so that we do not have to re-compute them when needed later. This simple optimization reduces time complexities from exponential to polynomial. For example, if we write simple recursive solution for Fibonacci Numbers, we get exponential time complexity and if we optimize it by storing solutions of subproblems, time complexity reduces to linear.

- Tabulation vs Memoizatation
- Optimal Substructure Property
- Overlapping Subproblems Property
- How to solve a Dynamic Programming Problem ?
- Greedy Method and Dynamic Programming both are used for solving the optimization problem
- Dynamic Program follow principle of Optimality
- Memoization follows top down approach

Memoization means no re-computation, which makes for a more efficient algorithm. Thus, memoization ensures that dynamic programming is efficient, but it is choosing the right sub-problem that guarantees that a dynamic program goes through all possibilities in order to find the best one.

Tabulation Method:
This approach follows bot up approach
It uses the iterative approach

Recursion vs Dynamic Programming
Dynamic programming is mostly applied to recursive algorithms. This is not a coincidence, most optimization problems require recursion and dynamic programming is used for optimization.
But not all problems that use recursion can use Dynamic Programming. Unless there is a presence of overlapping subproblems like in the fibonacci sequence problem, a recursion can only reach the solution using a divide and conquer approach.
That is the reason why a recursive algorithm like Merge Sort cannot use Dynamic Programming, because the subproblems are not overlapping in any way

Heap

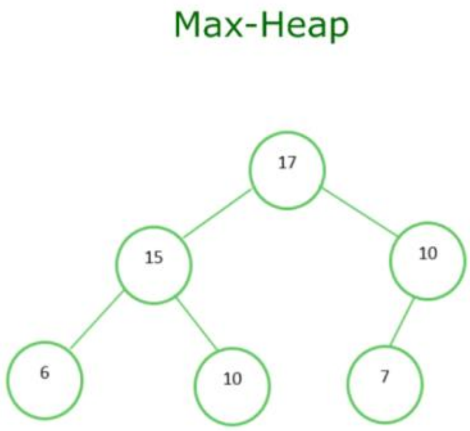
09 May 2022 07:01

What is a heap?

A heap is a complete binary tree, and the binary tree is a tree in which the node can have the utmost two children. A complete binary tree is a binary tree in which all the levels except the last level, i.e., leaf node, should be completely filled, and all the nodes should be left-justified.

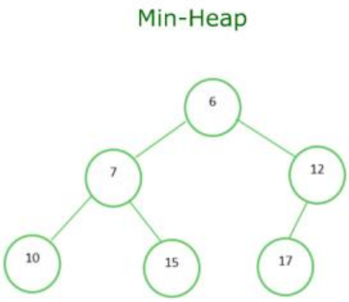
Max Heap:

A max-heap is a complete binary tree in which the value in each internal node is greater than or equal to the values in the children of that node



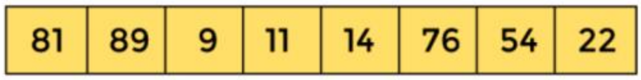
Min Heap:

A min-heap is a complete binary tree in which the root value in each internal node is less than to the values in the children of that node



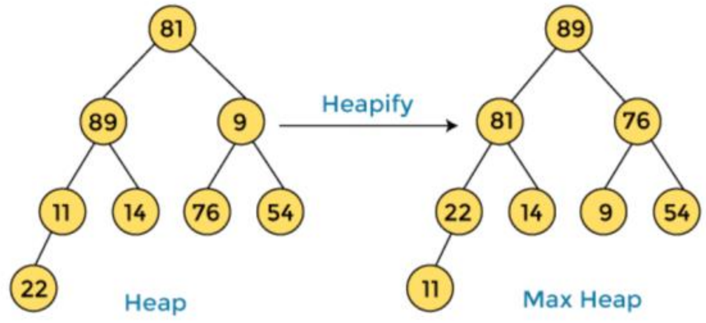
How to Construct Max heap ?

- **First Construct the heap (complete binary tree)**
- **Then convert the heap into max heap by comparing the elements and make sure that root is greater than or equal to children values**



81	89	9	11	14	76	54	22

First, we have to construct a heap from the given array and convert it into max heap.



We Always delete the root and last element will take the root place and later we have to put it in max heap order that means, push the elements to wards down

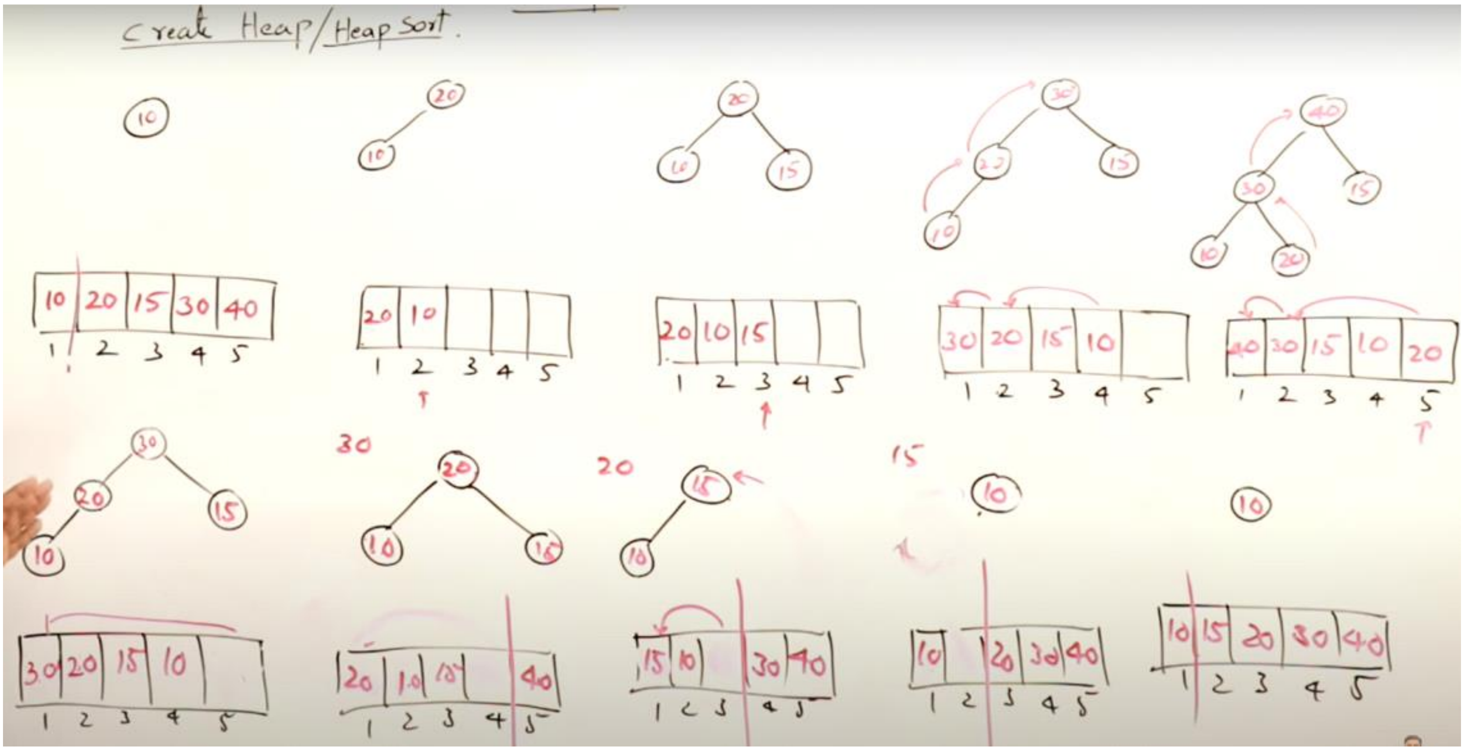
In Inserting the element in Max heap will happen from bottom to top :

In Deletion the element in max heap will happen from top to bottom

From Max heap whenever we delete we get the Maximum element
From Min heap whenever we delete we get the minimum element

Example:
In the below diagram at first in heap there is only 1 element that is 10.
Later when we want to insert second element 20. the question comes that do we keep this 20 as Left child or right child. Answer is make sure that u follow the complete binary tree .
I can say that If I insert 20 towards left child then I can say that it is complete binary tree
After that compare 10 with 20 and shift to get max heap.

Later for 3rd element 15 -> insert 15 towards Right side of 20 because at this position only we can have complete binary tree.
Look at the below diagram



Steps on Heap Sort:

1. Build Max Heap

2. Delete the elements from Max Heap

In a complete binary tree leaf nodes will be at n/2+1 to n position
So apply heapify from n/2 position if array position is starting from 1

Apply heapify from n/2-1 if array position is starting from 0

Generally in the array the index starts from 0 so we should use the heapify from n/2-1 position.

To perform the heapsort first we need to create the Max heap. To create the max heap we should compare the child with parent. In the below code. We are writing the condition as n/2-1 -> reason for that is we need to compare or we need to identify the root node then compare it with children.

Max Heap Code

```
public void sort(int arr[])
   {
       int n = arr.length;

       // Build heap (rearrange array)
       for (int i = n / 2 - 1; i >= 0; i--)
           heapify(arr, n, i);

       // One by one extract an element from heap
       for (int i = n - 1; i > 0; i--) {
           // Move current root to end
           int temp = arr[0];
           arr[0] = arr[i];
           arr[i] = temp;

           // call max heapify on the reduced heap
           heapify(arr, i, 0);
       }
```

Heapify Code

Here-> Based on the root node passed to this method we need to find the left child and right child values and compare it with root node if needed we should do swap to achieve max heap.

Formula for Left child= 2*i+1
Right child 2*i+2

```
       int largest = i; // Initialize largest as root
       int l = 2 * i + 1; // left = 2*i + 1
       int r = 2 * i + 2; // right = 2*i + 2


       /*if(arr[l]>arr[r]) {
            largest=l;
       }else {
            largest=r;
       }*/
       // If left child is larger than root
       if (l < n && arr[l] > arr[largest])
           largest = l;

       // If right child is larger than largest so far
       if (r < n && arr[r] > arr[largest])
           largest = r;

       // If largest is not root
       if (largest != i) {
           int swap = arr[i];
           arr[i] = arr[largest];
           arr[largest] = swap;

           // Recursively heapify the affected sub-tree
           heapify(arr, n, largest);
       }
```

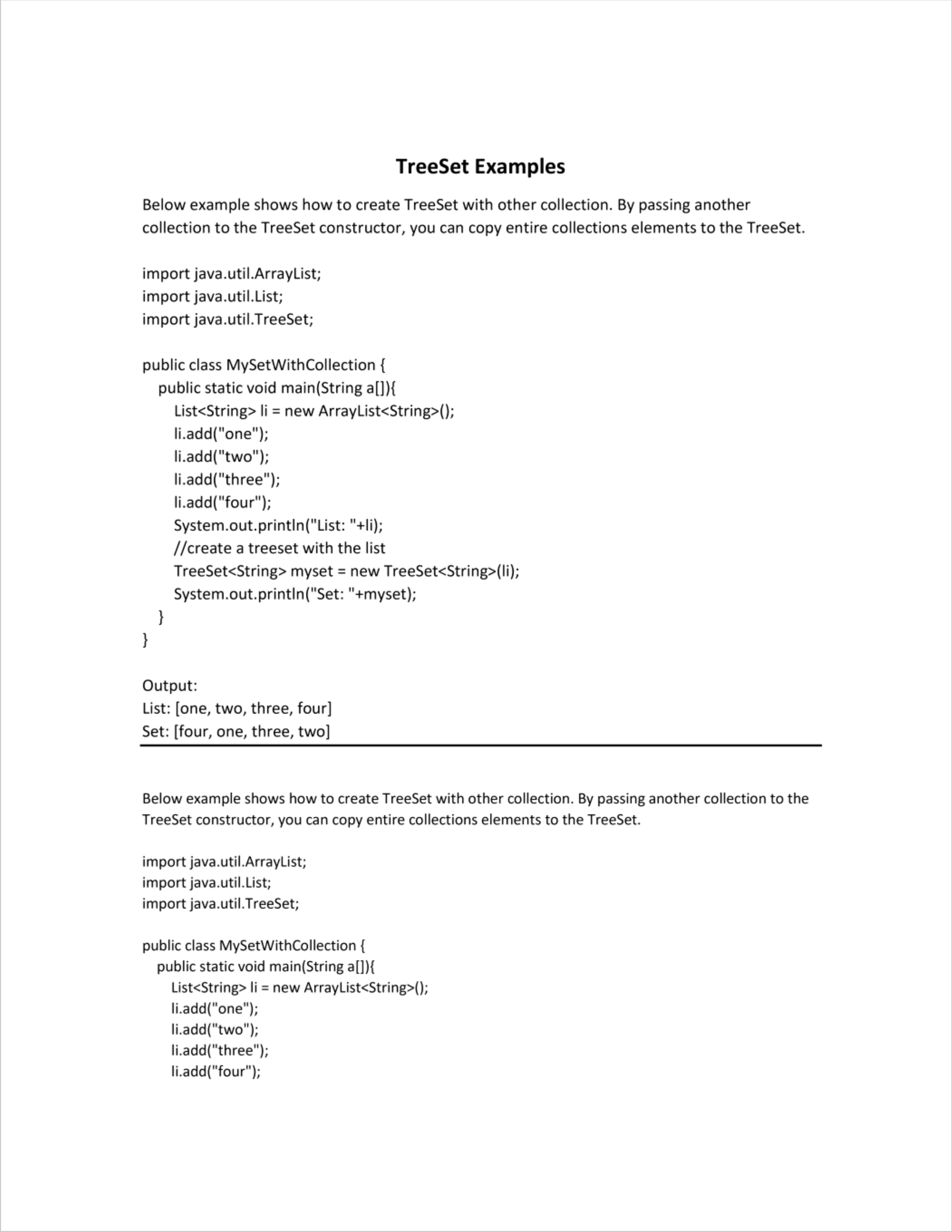
Interview Questions

16 January 2022 16:52

- What is the internal data structure used by ArrayList ? **:**
- How does add and remove method of array list works ?
- Write the custom arraylist ?
- What is the internal data Structure used by Hash Map ?
- What is collision in hashMap ?
- What is the internal implementation of hashMap ?
- What is the internal data structure used by Hash Set ?
- What is concurrent modification exception ? And how java knew this ?
- Difference between synchronized vs concurrent ?
- How to remove element from linked list ? (deleting middle, head and last)
- What is the difference between Array Blocking Queue and Linked Blocking Queue?
- What is the difference between pop, peek ?
- What is the difference between poll, peek ?
- What is Priority Queue ?
- What is the difference between array d queue and priority queue ?
- What is Stack and how can I implement Stack ?
- Difference between linear Search or Binary search ?
- What is pre requisite for binary search ?
- What is Queue and how can I implement Queue ?
- Write a program to implement Stack and Queue using linked list ?
- How to merge two sorted arrays ? (result should be sorted array) ?
- Write the algorithm for merge sort ?
- Write the algorithm for Quick sort ?
- Write a program to sort hashMap by values ?
- What is a binary tree ?
- What is the difference between full binary tree and complete binary tree ?
- Write a program to get the minimum and maximum values from binary search tree ?
- What are the tree traversal techniques ?
- What is the difference between hash set and tree set ?
- What are the difference between tree map and hash map ?
- What is copy on write Array set ?
- What is concurrent modification exception ?
- What is the internal implementation of concurrent modification exception ?

Interview Questions- Part2

07 February 2022 12:21



TreeSet Examples

Below example shows how to create TreeSet with other collection. By passing another collection to the TreeSet constructor, you can copy entire collections elements to the TreeSet.

```
import java.util.ArrayList;
import java.util.List;
import java.util.TreeSet;

public class MySetWithCollection {
   public static void main(String a[]){
      List<String> li = new ArrayList<String>();
      li.add("one");
      li.add("two");
      li.add("three");
      li.add("four");
      System.out.println("List: "+li);
      //create a treeset with the list
      TreeSet<String> myset = new TreeSet<String>(li);
      System.out.println("Set: "+myset);
   }
}
```

```
Output:
List: [one, two, three, four]
Set: [four, one, three, two]
```

Below example shows how to create TreeSet with other collection. By passing another collection to the TreeSet constructor, you can copy entire collections elements to the TreeSet.

```
import java.util.ArrayList;
import java.util.List;
import java.util.TreeSet;

public class MySetWithCollection {
   public static void main(String a[]){
      List<String> li = new ArrayList<String>();
      li.add("one");
      li.add("two");
      li.add("three");
      li.add("four");
```

```
        System.out.println("List: "+li);
        //create a treeset with the list
        TreeSet<String> myset = new TreeSet<String>(li);
        System.out.println("Set: "+myset);
    }
}
```

```
Output:
List: [one, two, three, four]
Set: [four, one, three, two]
```

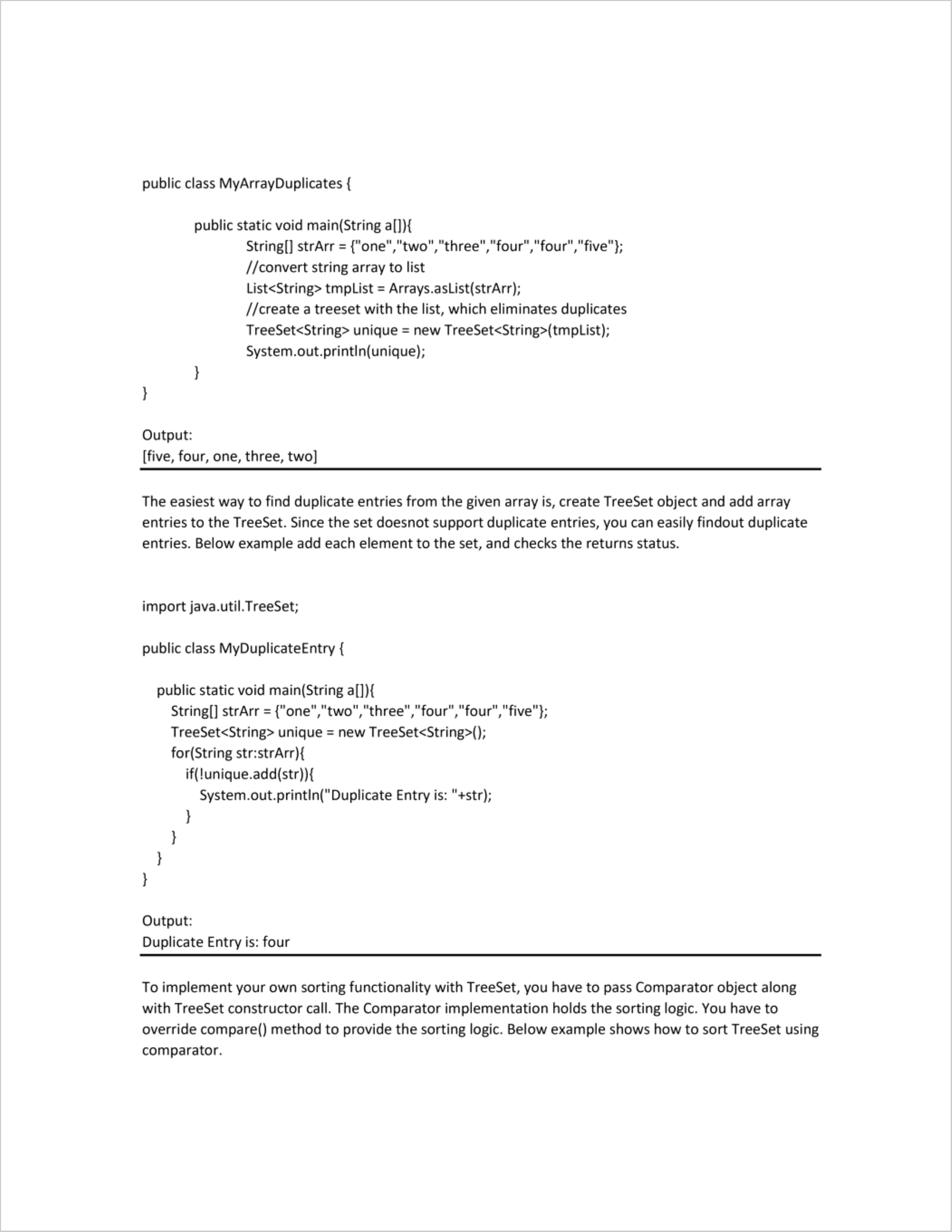
Below example shows how to read objects using Iterator. By calling iterator() method you will get Iterator object, through which you can iterate through all the elements of the TreeSet.

```
import java.util.Iterator;
import java.util.TreeSet;

public class MySetIteration {

    public static void main(String a[]){

        TreeSet<String> ts = new TreeSet<String>();
        ts.add("one");
        ts.add("two");
        ts.add("three");
        Iterator<String> itr = ts.iterator();
        while(itr.hasNext()){
            System.out.println(itr.next());
        }
    }
}
```

```
Output:
one
three
two
```

The easiest way to remove duplicate entries from the given array is, create TreeSet object and add array entries to the TreeSet. Since the set doesnot support duplicate entries, you will get only unique elements left with TreeSet.

```
import java.util.Arrays;
import java.util.List;
import java.util.TreeSet;
```



```
public class MyArrayDuplicates {

        public static void main(String a[]){
                String[] strArr = {"one","two","three","four","four","five"};
                //convert string array to list
                List<String> tmpList = Arrays.asList(strArr);
                //create a treeset with the list, which eliminates duplicates
                TreeSet<String> unique = new TreeSet<String>(tmpList);
                System.out.println(unique);
        }
}
```

Output:

```
[five, four, one, three, two]
```

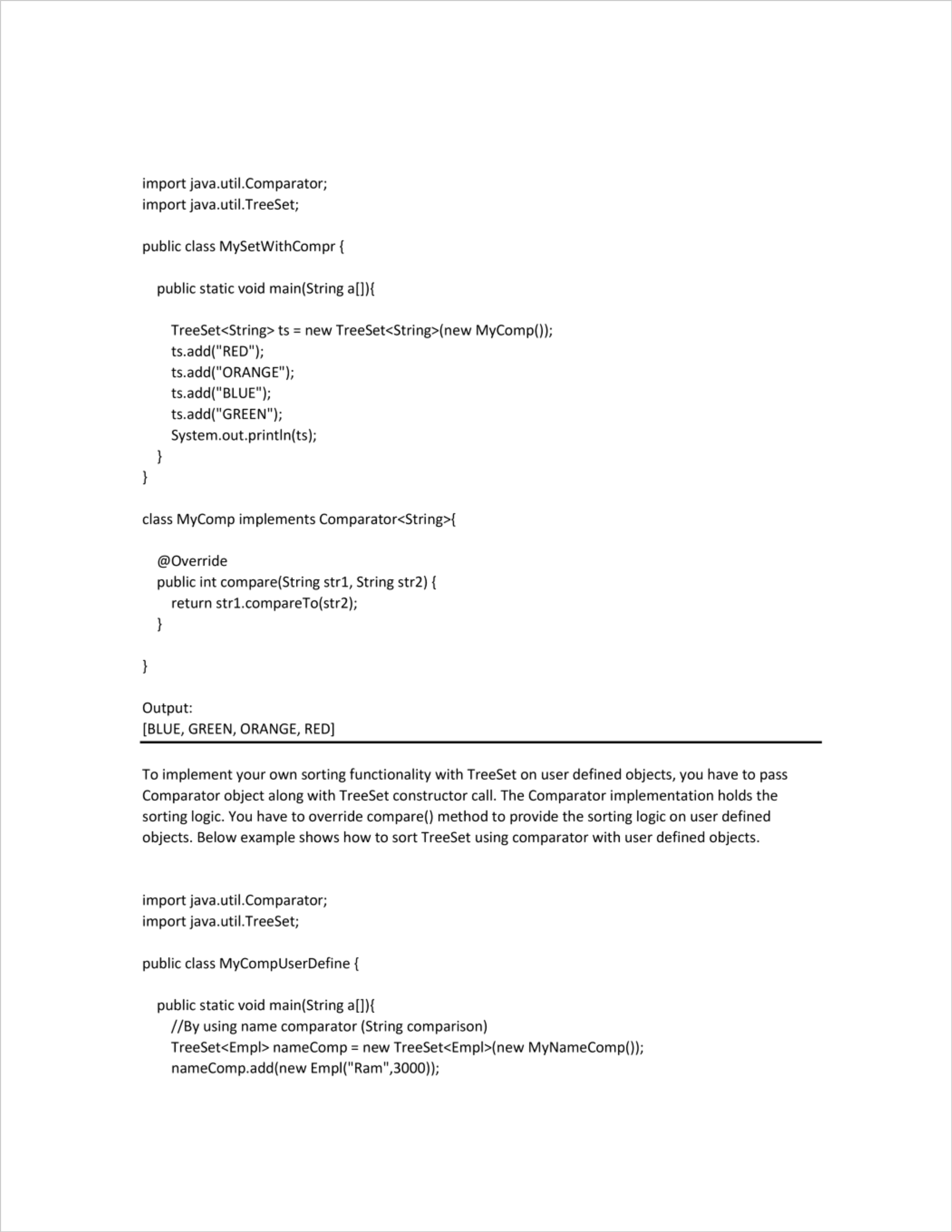
The easiest way to find duplicate entries from the given array is, create TreeSet object and add array entries to the TreeSet. Since the set doesnot support duplicate entries, you can easily findout duplicate entries. Below example add each element to the set, and checks the returns status.

```
import java.util.TreeSet;

public class MyDuplicateEntry {

   public static void main(String a[]){
      String[] strArr = {"one","two","three","four","four","five"};
      TreeSet<String> unique = new TreeSet<String>();
      for(String str:strArr){
         if(!unique.add(str)){
            System.out.println("Duplicate Entry is: "+str);
         }
      }
   }
}
```

Output:

```
Duplicate Entry is: four
```

To implement your own sorting functionality with TreeSet, you have to pass Comparator object along with TreeSet constructor call. The Comparator implementation holds the sorting logic. You have to override compare() method to provide the sorting logic. Below example shows how to sort TreeSet using comparator.



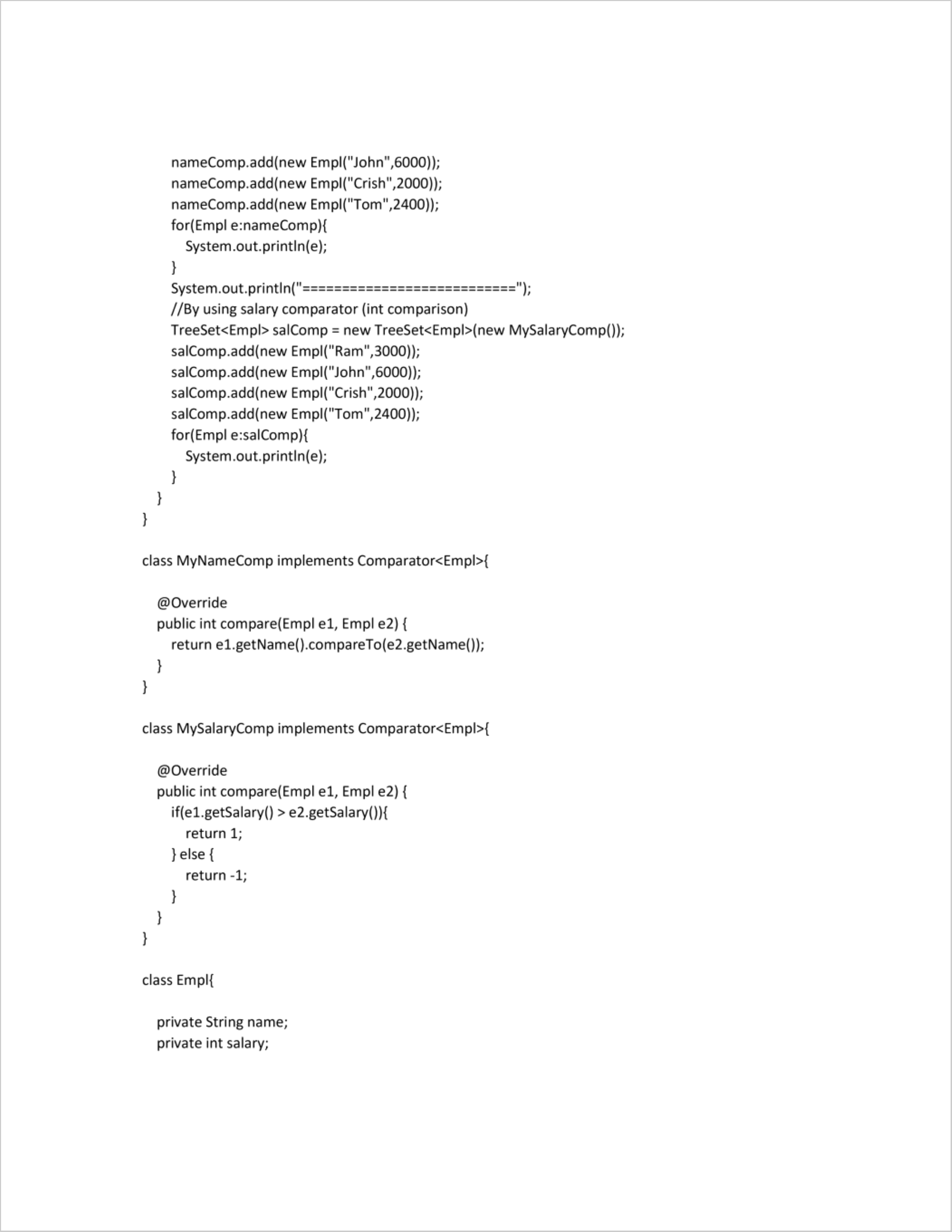
```
import java.util.Comparator;
import java.util.TreeSet;

public class MySetWithCompr {

   public static void main(String a[]){

      TreeSet<String> ts = new TreeSet<String>(new MyComp());
      ts.add("RED");
      ts.add("ORANGE");
      ts.add("BLUE");
      ts.add("GREEN");
      System.out.println(ts);
   }
}

class MyComp implements Comparator<String>{

   @Override
   public int compare(String str1, String str2) {
      return str1.compareTo(str2);
   }

}
```

Output:

```
[BLUE, GREEN, ORANGE, RED]
```

To implement your own sorting functionality with TreeSet on user defined objects, you have to pass Comparator object along with TreeSet constructor call. The Comparator implementation holds the sorting logic. You have to override compare() method to provide the sorting logic on user defined objects. Below example shows how to sort TreeSet using comparator with user defined objects.

```
import java.util.Comparator;
import java.util.TreeSet;

public class MyCompUserDefine {

   public static void main(String a[]){
      //By using name comparator (String comparison)
      TreeSet<Empl> nameComp = new TreeSet<Empl>(new MyNameComp());
      nameComp.add(new Empl("Ram",3000));
```

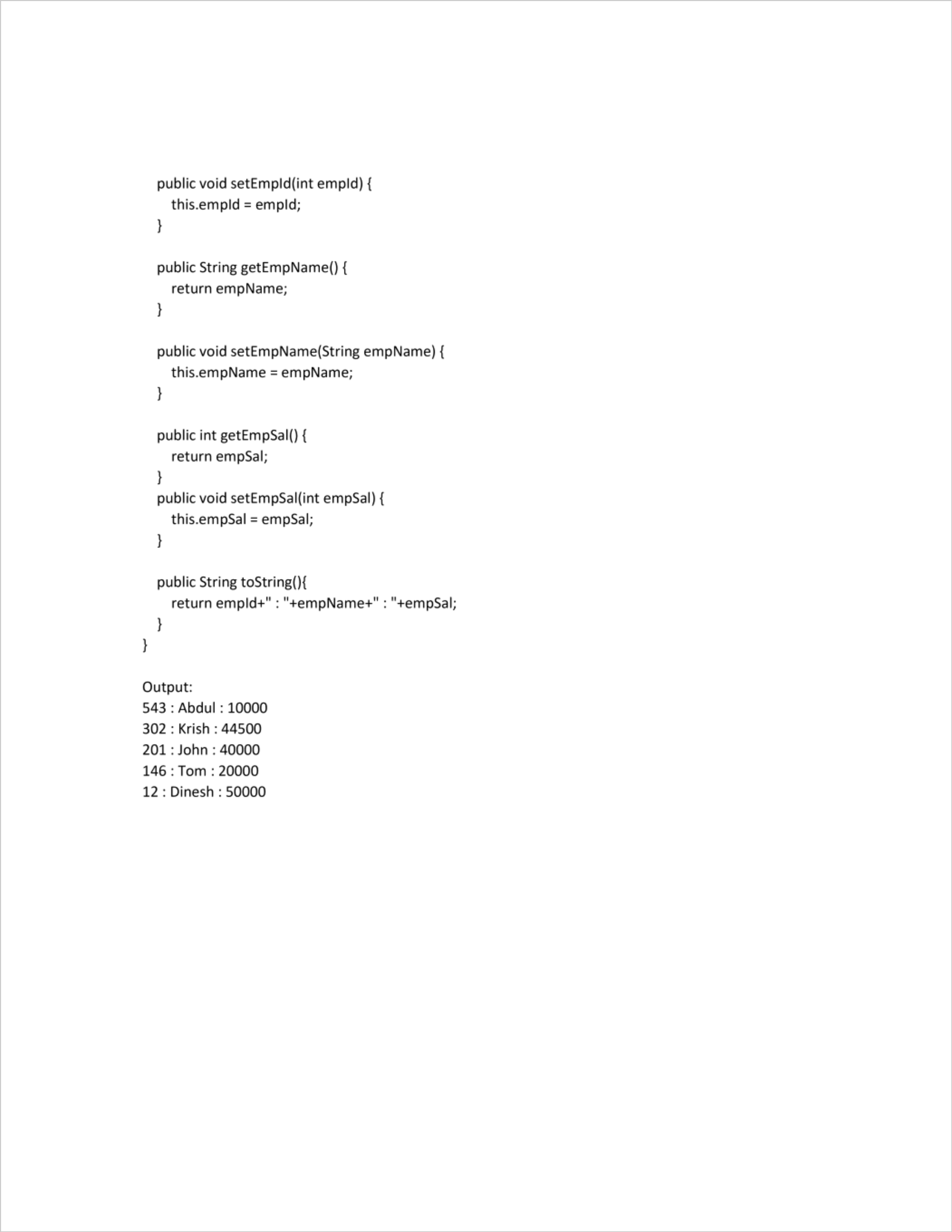


```
        nameComp.add(new Empl("John",6000));
        nameComp.add(new Empl("Crish",2000));
        nameComp.add(new Empl("Tom",2400));
        for(Empl e:nameComp){
            System.out.println(e);
        }
        System.out.println("===========================");
        //By using salary comparator (int comparison)
        TreeSet<Empl> salComp = new TreeSet<Empl>(new MySalaryComp());
        salComp.add(new Empl("Ram",3000));
        salComp.add(new Empl("John",6000));
        salComp.add(new Empl("Crish",2000));
        salComp.add(new Empl("Tom",2400));
        for(Empl e:salComp){
            System.out.println(e);
        }
    }
}

class MyNameComp implements Comparator<Empl>{

    @Override
    public int compare(Empl e1, Empl e2) {
        return e1.getName().compareTo(e2.getName());
    }
}

class MySalaryComp implements Comparator<Empl>{

    @Override
    public int compare(Empl e1, Empl e2) {
        if(e1.getSalary() > e2.getSalary()){
            return 1;
        } else {
            return -1;
        }
    }
}

class Empl{

    private String name;
    private int salary;
    public Empl(String n, int s){
        this.name = n;
        this.salary = s;
    }

    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getSalary() {
        return salary;
    }
    public void setSalary(int salary) {
        this.salary = salary;
    }
    public String toString(){
        return "Name: "+this.name+"-- Salary: "+this.salary;
    }
}
```

```
Output:
Name: Crish-- Salary: 2000
Name: John-- Salary: 6000
Name: Ram-- Salary: 3000
Name: Tom-- Salary: 2400
===========================
Name: Crish-- Salary: 2000
Name: Tom-- Salary: 2400
Name: Ram-- Salary: 3000
Name: John-- Salary: 6000
```

To avoid duplicate user defined objects in TreeSet, you have to implement Comparator interface with equality verification. Below example gives a sample code to implement it.

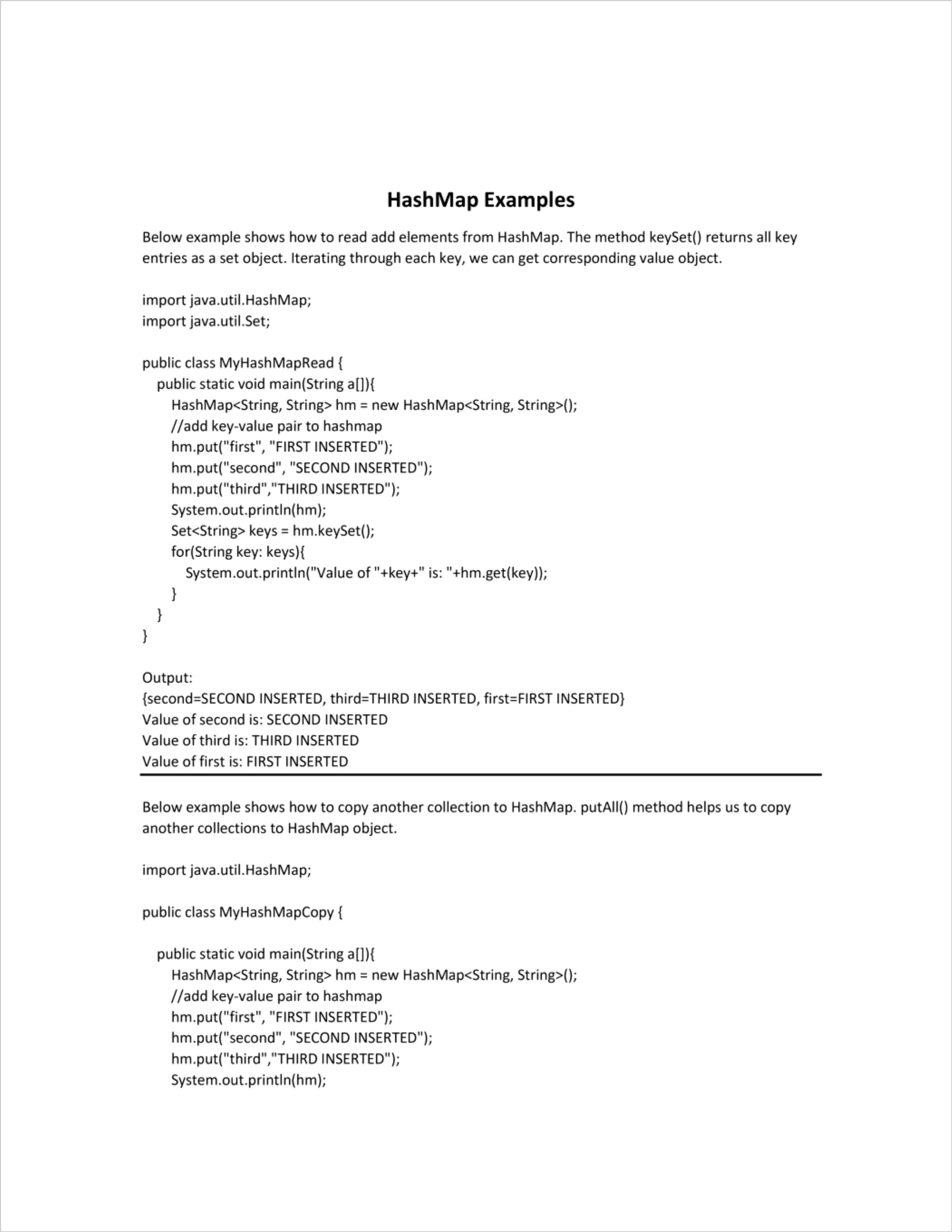
```
import java.util.Comparator;
import java.util.Set;
import java.util.TreeSet;

public class MyUserDuplicates {

    public static void main(String a[]){
        Set<Emp> ts = new TreeSet<Emp>(new EmpComp());
        ts.add(new Emp(201,"John",40000));
        ts.add(new Emp(302,"Krish",44500));
        ts.add(new Emp(146,"Tom",20000));
        ts.add(new Emp(543,"Abdul",10000));
        ts.add(new Emp(12,"Dinesh",50000));
        //adding duplicate entry
        ts.add(new Emp(146,"Tom",20000));
        //check duplicate entry is there or not
        for(Emp e:ts){
            System.out.println(e);
        }
    }
}

class EmpComp implements Comparator<Emp>{

    @Override
    public int compare(Emp e1, Emp e2) {
        if(e1.getEmpId() == e2.getEmpId()){
            return 0;
        } if(e1.getEmpId() < e2.getEmpId()){
            return 1;
        } else {
            return -1;
        }
    }
}

class Emp {

    private int empId;
    private String empName;
    private int empSal;

    public Emp(int id, String name, int sal){
        this.empId = id;
        this.empName = name;
        this.empSal = sal;
    }

    public int getEmpId() {
        return empId;
    }
```



```
  public void setEmpId(int empId) {
    this.empId = empId;
  }

  public String getEmpName() {
    return empName;
  }

  public void setEmpName(String empName) {
    this.empName = empName;
  }

  public int getEmpSal() {
    return empSal;
  }
  public void setEmpSal(int empSal) {
    this.empSal = empSal;
  }

  public String toString(){
    return empId+" : "+empName+" : "+empSal;
  }
}
```

Output:

```
543 : Abdul : 10000
302 : Krish : 44500
201 : John : 40000
146 : Tom : 20000
12 : Dinesh : 50000
```



HashMap Examples

Below example shows how to read add elements from HashMap. The method keySet() returns all key entries as a set object. Iterating through each key, we can get corresponding value object.

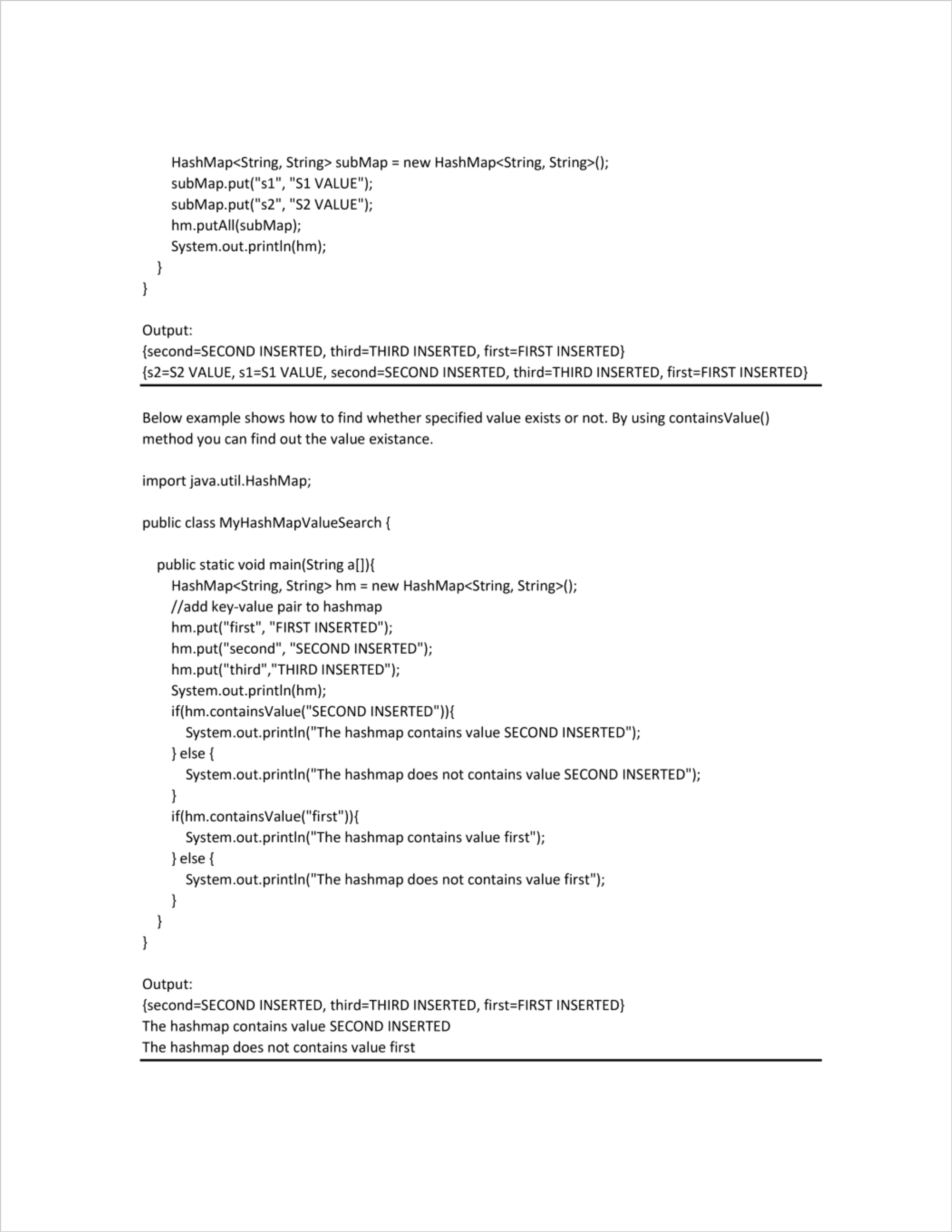
```
import java.util.HashMap;
import java.util.Set;

public class MyHashMapRead {
   public static void main(String a[]){
      HashMap<String, String> hm = new HashMap<String, String>();
      //add key-value pair to hashmap
      hm.put("first", "FIRST INSERTED");
      hm.put("second", "SECOND INSERTED");
      hm.put("third","THIRD INSERTED");
      System.out.println(hm);
      Set<String> keys = hm.keySet();
      for(String key: keys){
         System.out.println("Value of "+key+" is: "+hm.get(key));
      }
   }
}
```

Output:

```
{second=SECOND INSERTED, third=THIRD INSERTED, first=FIRST INSERTED}
Value of second is: SECOND INSERTED
Value of third is: THIRD INSERTED
Value of first is: FIRST INSERTED
```

Below example shows how to copy another collection to HashMap. putAll() method helps us to copy another collections to HashMap object.

```
import java.util.HashMap;

public class MyHashMapCopy {

   public static void main(String a[]){
      HashMap<String, String> hm = new HashMap<String, String>();
      //add key-value pair to hashmap
      hm.put("first", "FIRST INSERTED");
      hm.put("second", "SECOND INSERTED");
      hm.put("third","THIRD INSERTED");
      System.out.println(hm);
```



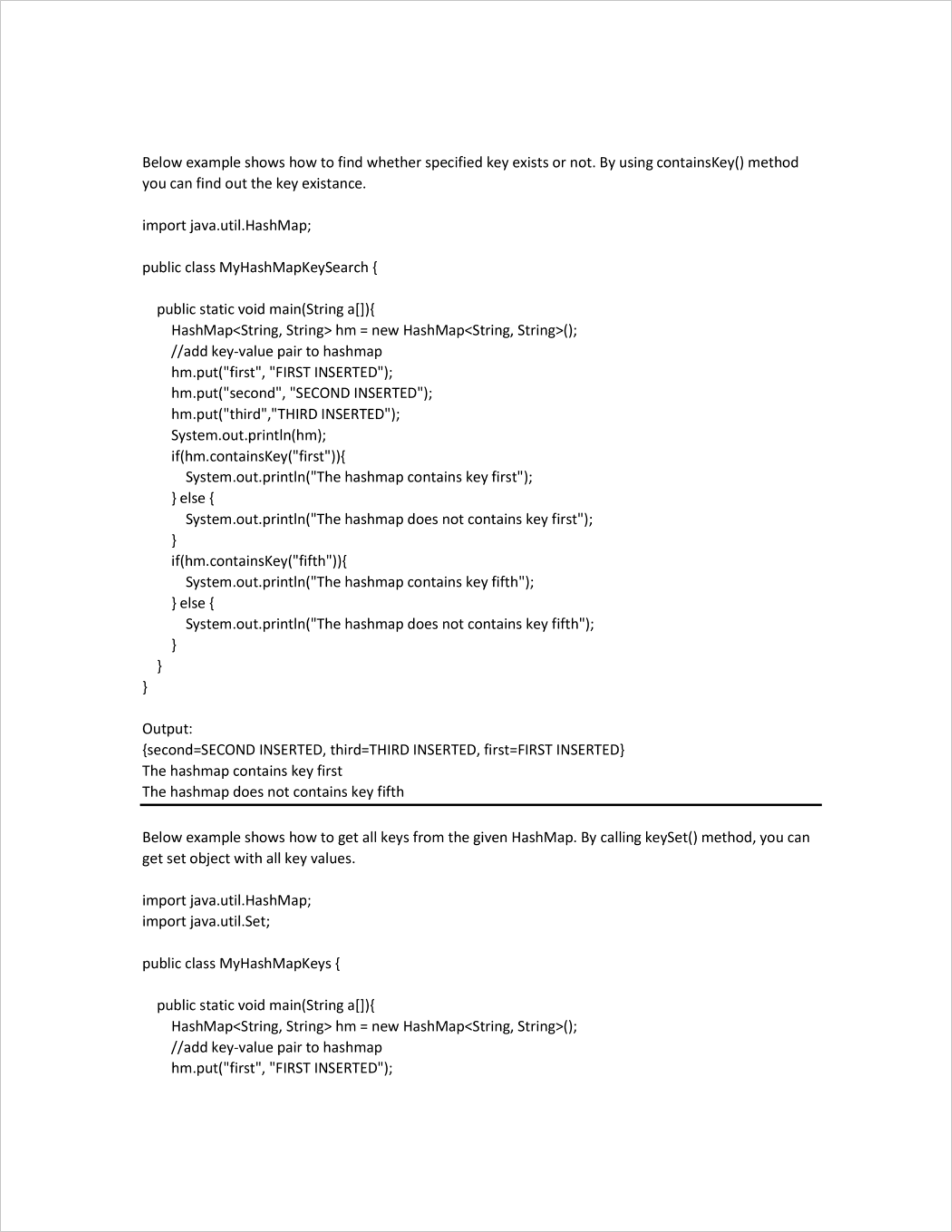
```
        HashMap<String, String> subMap = new HashMap<String, String>();
        subMap.put("s1", "S1 VALUE");
        subMap.put("s2", "S2 VALUE");
        hm.putAll(subMap);
        System.out.println(hm);
    }
}
```

```
Output:
{second=SECOND INSERTED, third=THIRD INSERTED, first=FIRST INSERTED}
{s2=S2 VALUE, s1=S1 VALUE, second=SECOND INSERTED, third=THIRD INSERTED, first=FIRST INSERTED}
```

Below example shows how to find whether specified value exists or not. By using containsValue() method you can find out the value existance.

```
import java.util.HashMap;

public class MyHashMapValueSearch {

    public static void main(String a[]){
        HashMap<String, String> hm = new HashMap<String, String>();
        //add key-value pair to hashmap
        hm.put("first", "FIRST INSERTED");
        hm.put("second", "SECOND INSERTED");
        hm.put("third","THIRD INSERTED");
        System.out.println(hm);
        if(hm.containsValue("SECOND INSERTED")){
            System.out.println("The hashmap contains value SECOND INSERTED");
        } else {
            System.out.println("The hashmap does not contains value SECOND INSERTED");
        }
        if(hm.containsValue("first")){
            System.out.println("The hashmap contains value first");
        } else {
            System.out.println("The hashmap does not contains value first");
        }
    }
}
```

```
Output:
{second=SECOND INSERTED, third=THIRD INSERTED, first=FIRST INSERTED}
The hashmap contains value SECOND INSERTED
The hashmap does not contains value first
```



Below example shows how to find whether specified key exists or not. By using containsKey() method you can find out the key existance.

```
import java.util.HashMap;

public class MyHashMapKeySearch {

    public static void main(String a[]){
        HashMap<String, String> hm = new HashMap<String, String>();
        //add key-value pair to hashmap
        hm.put("first", "FIRST INSERTED");
        hm.put("second", "SECOND INSERTED");
        hm.put("third","THIRD INSERTED");
        System.out.println(hm);
        if(hm.containsKey("first")){
            System.out.println("The hashmap contains key first");
        } else {
            System.out.println("The hashmap does not contains key first");
        }
        if(hm.containsKey("fifth")){
            System.out.println("The hashmap contains key fifth");
        } else {
            System.out.println("The hashmap does not contains key fifth");
        }
    }
}
```

Output:

```
{second=SECOND INSERTED, third=THIRD INSERTED, first=FIRST INSERTED}
The hashmap contains key first
The hashmap does not contains key fifth
```

Below example shows how to get all keys from the given HashMap. By calling keySet() method, you can get set object with all key values.

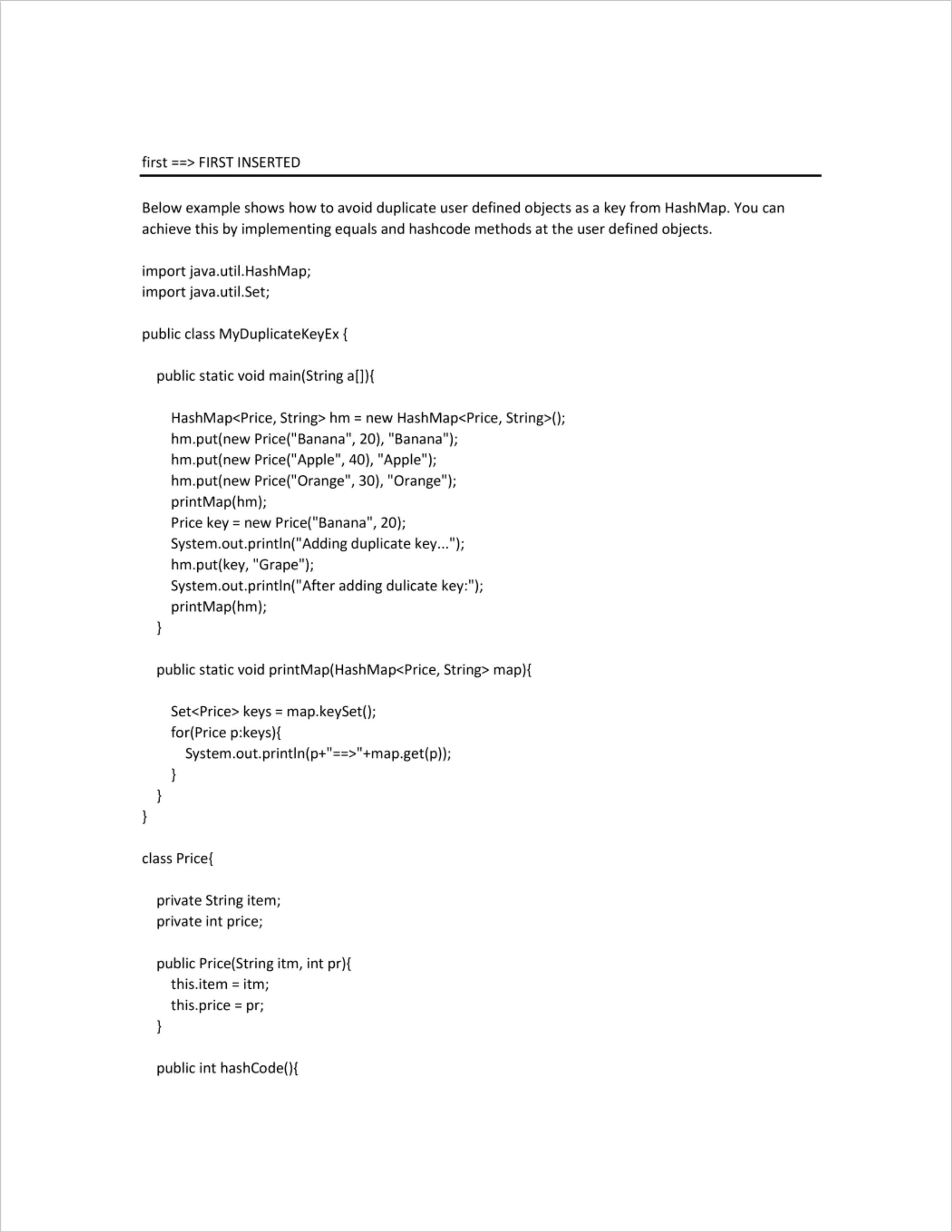
```
import java.util.HashMap;
import java.util.Set;

public class MyHashMapKeys {

    public static void main(String a[]){
        HashMap<String, String> hm = new HashMap<String, String>();
        //add key-value pair to hashmap
        hm.put("first", "FIRST INSERTED");
        hm.put("second", "SECOND INSERTED");
        hm.put("third","THIRD INSERTED");
        System.out.println(hm);
        Set<String> keys = hm.keySet();
        for(String key: keys){
            System.out.println(key);
        }
    }
}
```

```
Output:
{second=SECOND INSERTED, third=THIRD INSERTED, first=FIRST INSERTED}
second
third
first
```

Below example shows how to get all key-value pair as Entry objects. Entry class provides getter methods to access key-value details. The method entrySet() provides all entries as set object.

```
import java.util.HashMap;
import java.util.Map.Entry;
import java.util.Set;

public class MyHashMapEntrySet {

    public static void main(String a[]){
        HashMap<String, String> hm = new HashMap<String, String>();
        //add key-value pair to hashmap
        hm.put("first", "FIRST INSERTED");
        hm.put("second", "SECOND INSERTED");
        hm.put("third","THIRD INSERTED");
        System.out.println(hm);
        //getting value for the given key from hashmap
        Set<Entry<String, String>> entires = hm.entrySet();
        for(Entry<String,String> ent:entires){
            System.out.println(ent.getKey()+" ==> "+ent.getValue());
        }
    }
}
```

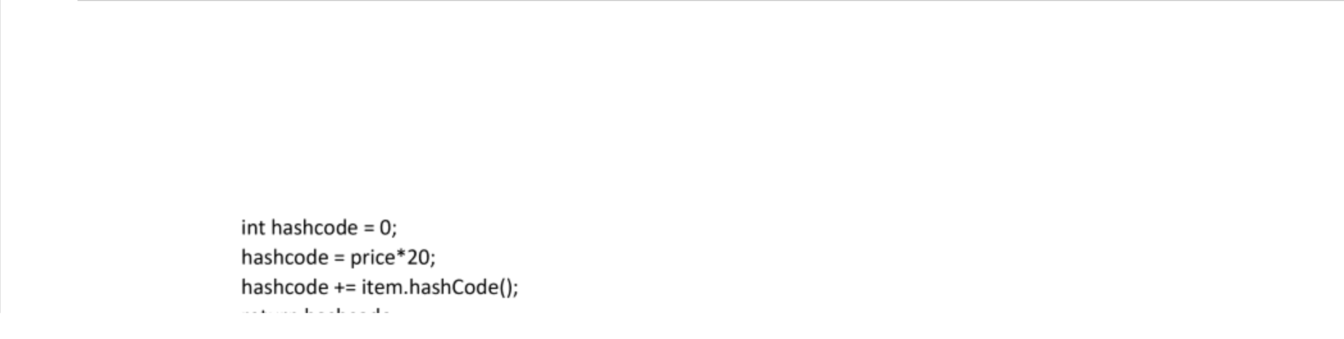
```
Output:
{second=SECOND INSERTED, third=THIRD INSERTED, first=FIRST INSERTED}
second ==> SECOND INSERTED
third ==> THIRD INSERTED
```



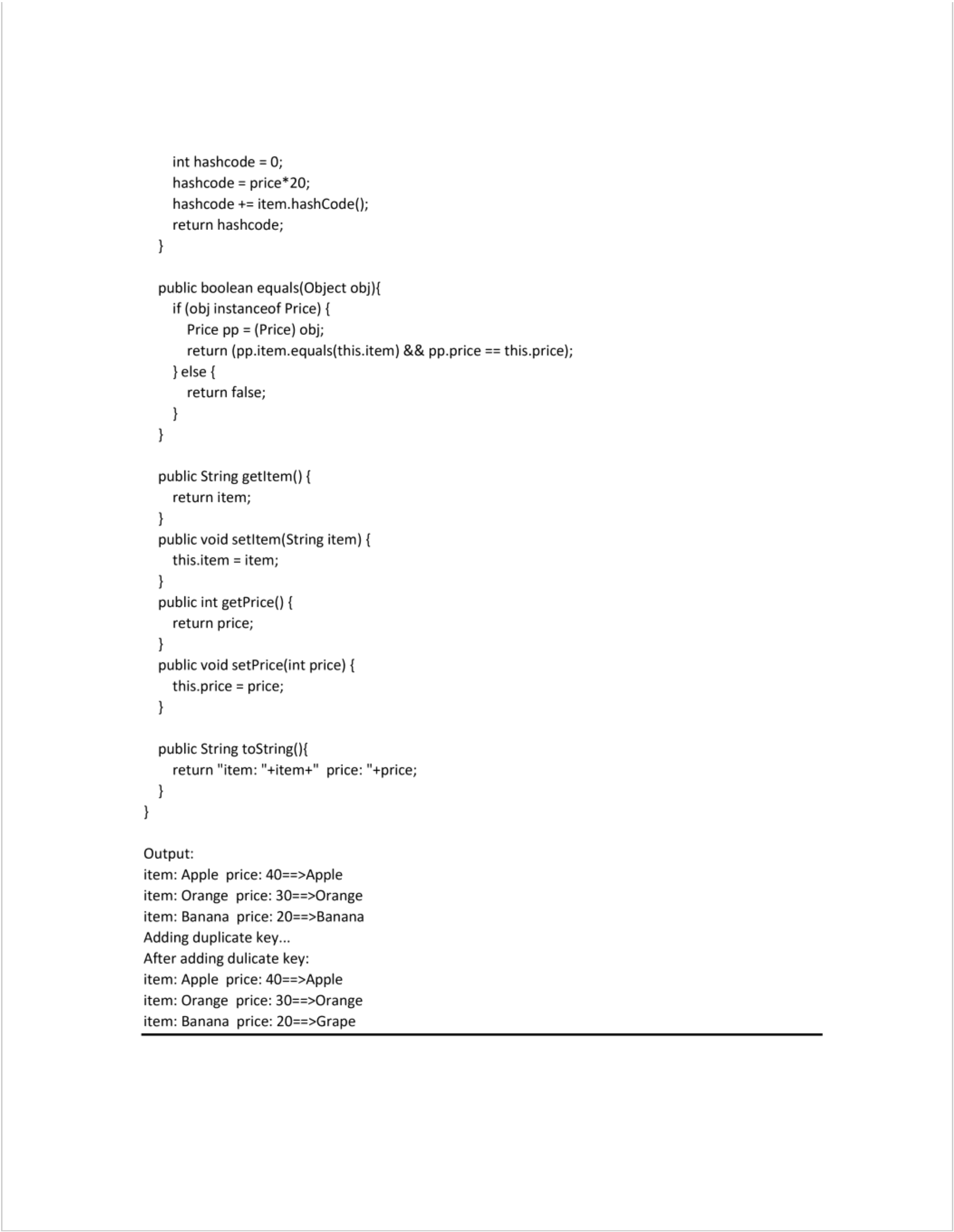
```
first ==> FIRST INSERTED
```

Below example shows how to avoid duplicate user defined objects as a key from HashMap. You can achieve this by implementing equals and hashcode methods at the user defined objects.

```
import java.util.HashMap;
import java.util.Set;

public class MyDuplicateKeyEx {

   public static void main(String a[]){

      HashMap<Price, String> hm = new HashMap<Price, String>();
      hm.put(new Price("Banana", 20), "Banana");
      hm.put(new Price("Apple", 40), "Apple");
      hm.put(new Price("Orange", 30), "Orange");
      printMap(hm);
      Price key = new Price("Banana", 20);
      System.out.println("Adding duplicate key...");
      hm.put(key, "Grape");
      System.out.println("After adding dulicate key:");
      printMap(hm);
   }

   public static void printMap(HashMap<Price, String> map){

      Set<Price> keys = map.keySet();
      for(Price p:keys){
         System.out.println(p+"==>"+map.get(p));
      }
   }
}

class Price{

   private String item;
   private int price;

   public Price(String itm, int pr){
      this.item = itm;
      this.price = pr;
   }

   public int hashCode(){
```



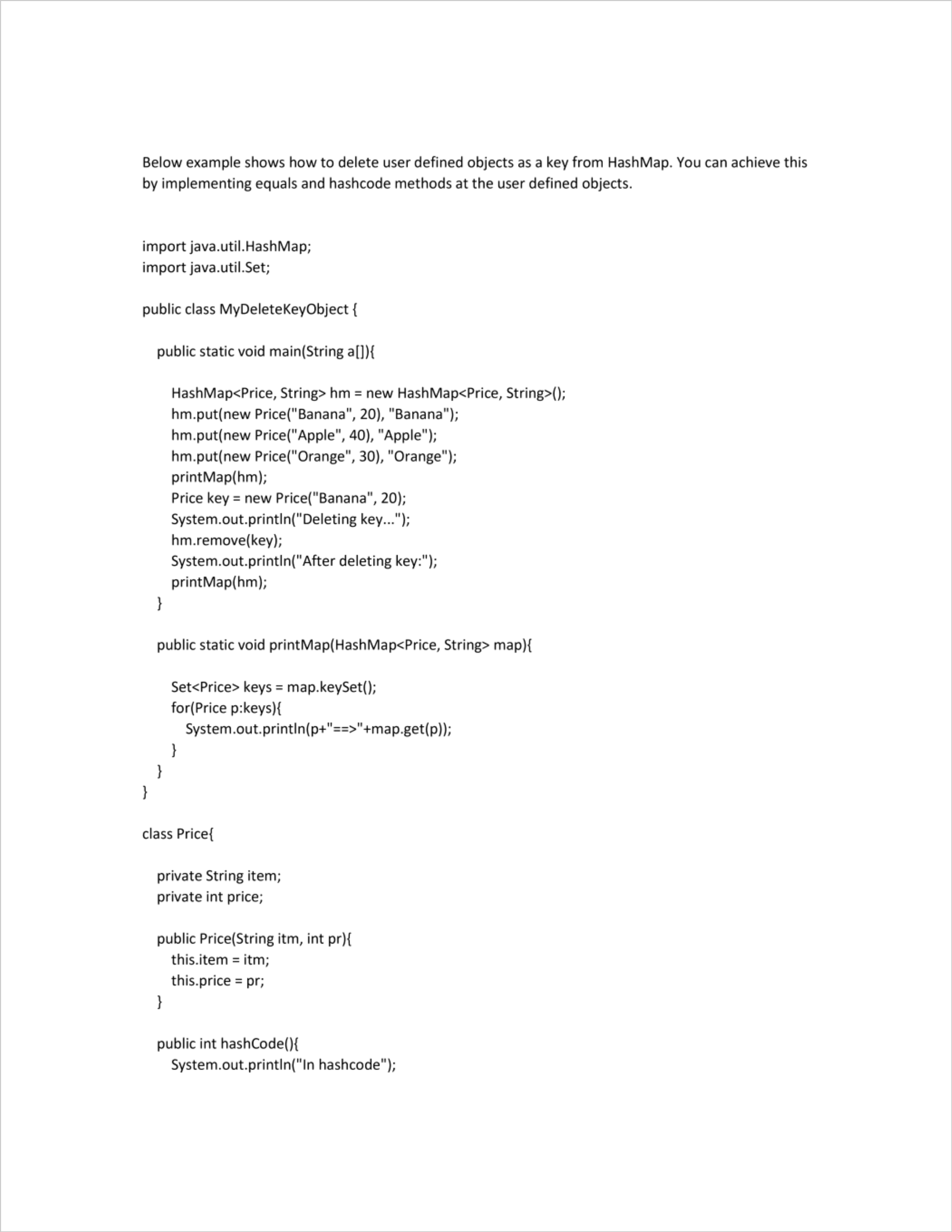
```
int hashcode = 0;
hashcode = price*20;
hashcode += item.hashCode();
return hashcode;
```



```
        int hashcode = 0;
        hashcode = price*20;
        hashcode += item.hashCode();
        return hashcode;
    }

    public boolean equals(Object obj){
        if (obj instanceof Price) {
            Price pp = (Price) obj;
            return (pp.item.equals(this.item) && pp.price == this.price);
        } else {
            return false;
        }
    }

    public String getItem() {
        return item;
    }
    public void setItem(String item) {
        this.item = item;
    }
    public int getPrice() {
        return price;
    }
    public void setPrice(int price) {
        this.price = price;
    }

    public String toString(){
        return "item: "+item+"  price: "+price;
    }
}
```

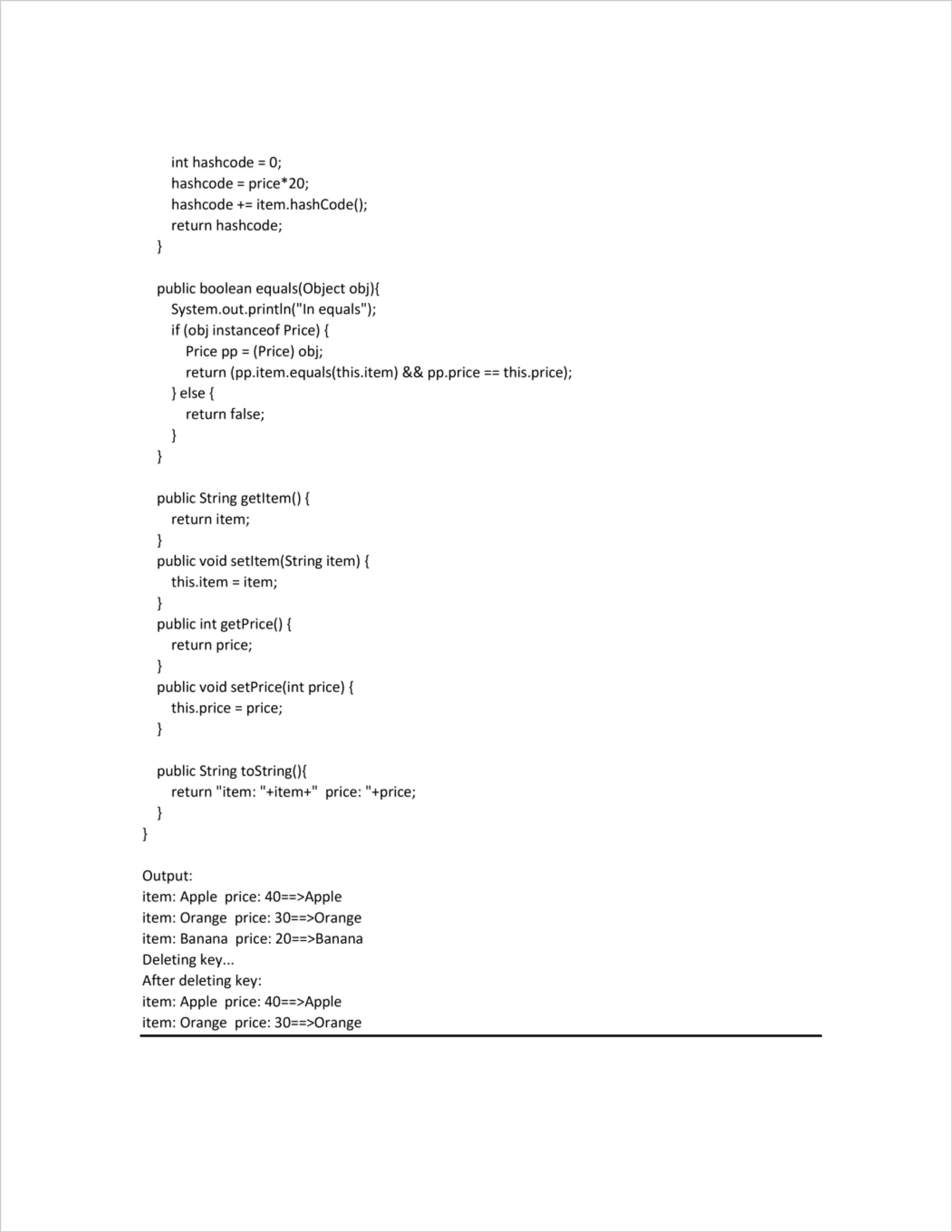
Output:

```
item: Apple  price: 40==>Apple
item: Orange  price: 30==>Orange
item: Banana  price: 20==>Banana
Adding duplicate key...
After adding dulicate key:
item: Apple  price: 40==>Apple
item: Orange  price: 30==>Orange
item: Banana  price: 20==>Grape
```



Below example shows how to delete user defined objects as a key from HashMap. You can achieve this by implementing equals and hashcode methods at the user defined objects.

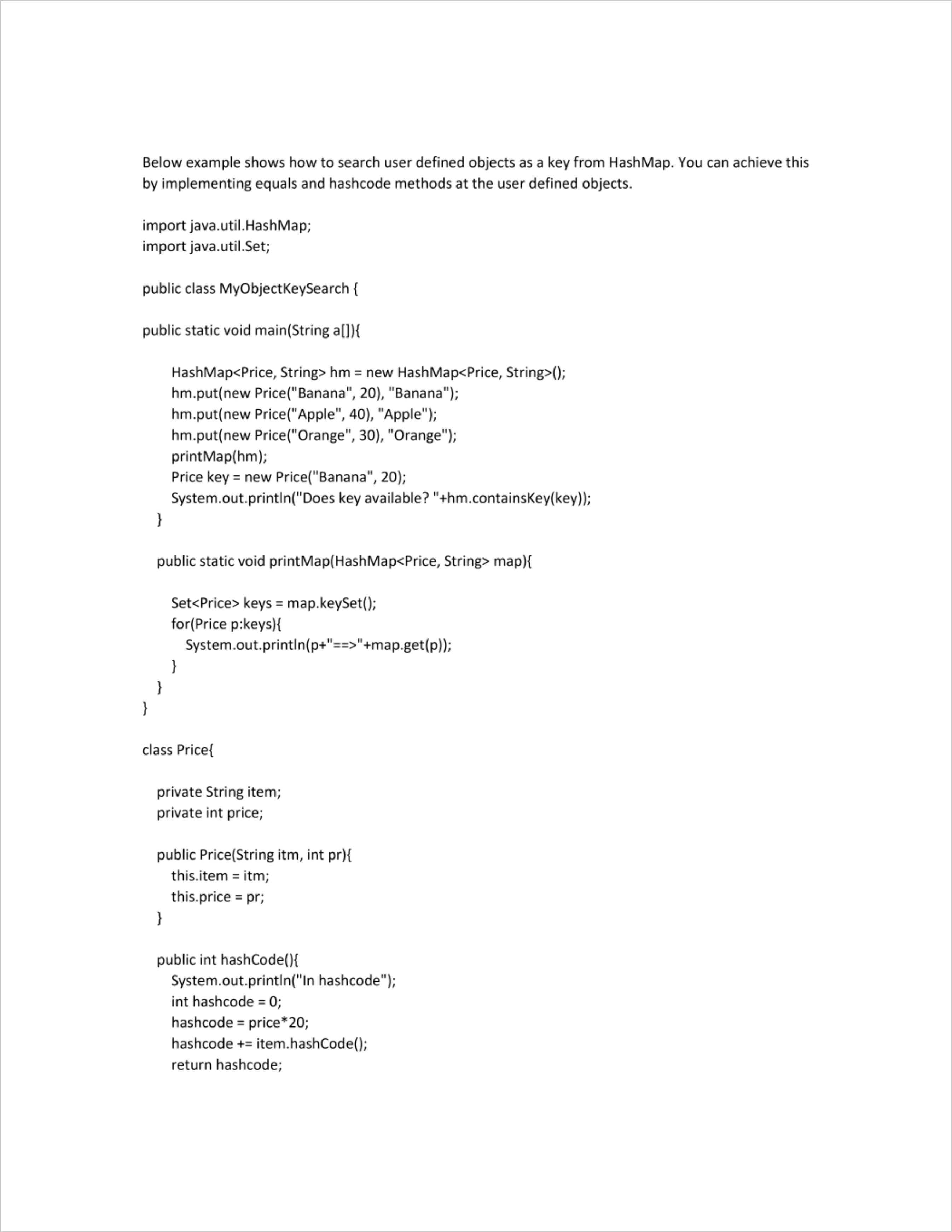
```
import java.util.HashMap;
import java.util.Set;

public class MyDeleteKeyObject {

    public static void main(String a[]){

        HashMap<Price, String> hm = new HashMap<Price, String>();
        hm.put(new Price("Banana", 20), "Banana");
        hm.put(new Price("Apple", 40), "Apple");
        hm.put(new Price("Orange", 30), "Orange");
        printMap(hm);
        Price key = new Price("Banana", 20);
        System.out.println("Deleting key...");
        hm.remove(key);
        System.out.println("After deleting key:");
        printMap(hm);
    }

    public static void printMap(HashMap<Price, String> map){

        Set<Price> keys = map.keySet();
        for(Price p:keys){
            System.out.println(p+"==>"+map.get(p));
        }
    }
}

class Price{

    private String item;
    private int price;

    public Price(String itm, int pr){
        this.item = itm;
        this.price = pr;
    }

    public int hashCode(){
        System.out.println("In hashcode");
```



```
      int hashcode = 0;
      hashcode = price*20;
      hashcode += item.hashCode();
      return hashcode;
   }

   public boolean equals(Object obj){
      System.out.println("In equals");
      if (obj instanceof Price) {
         Price pp = (Price) obj;
         return (pp.item.equals(this.item) && pp.price == this.price);
      } else {
         return false;
      }
   }

   public String getItem() {
      return item;
   }
   public void setItem(String item) {
      this.item = item;
   }
   public int getPrice() {
      return price;
   }
   public void setPrice(int price) {
      this.price = price;
   }

   public String toString(){
      return "item: "+item+"  price: "+price;
   }
}
```

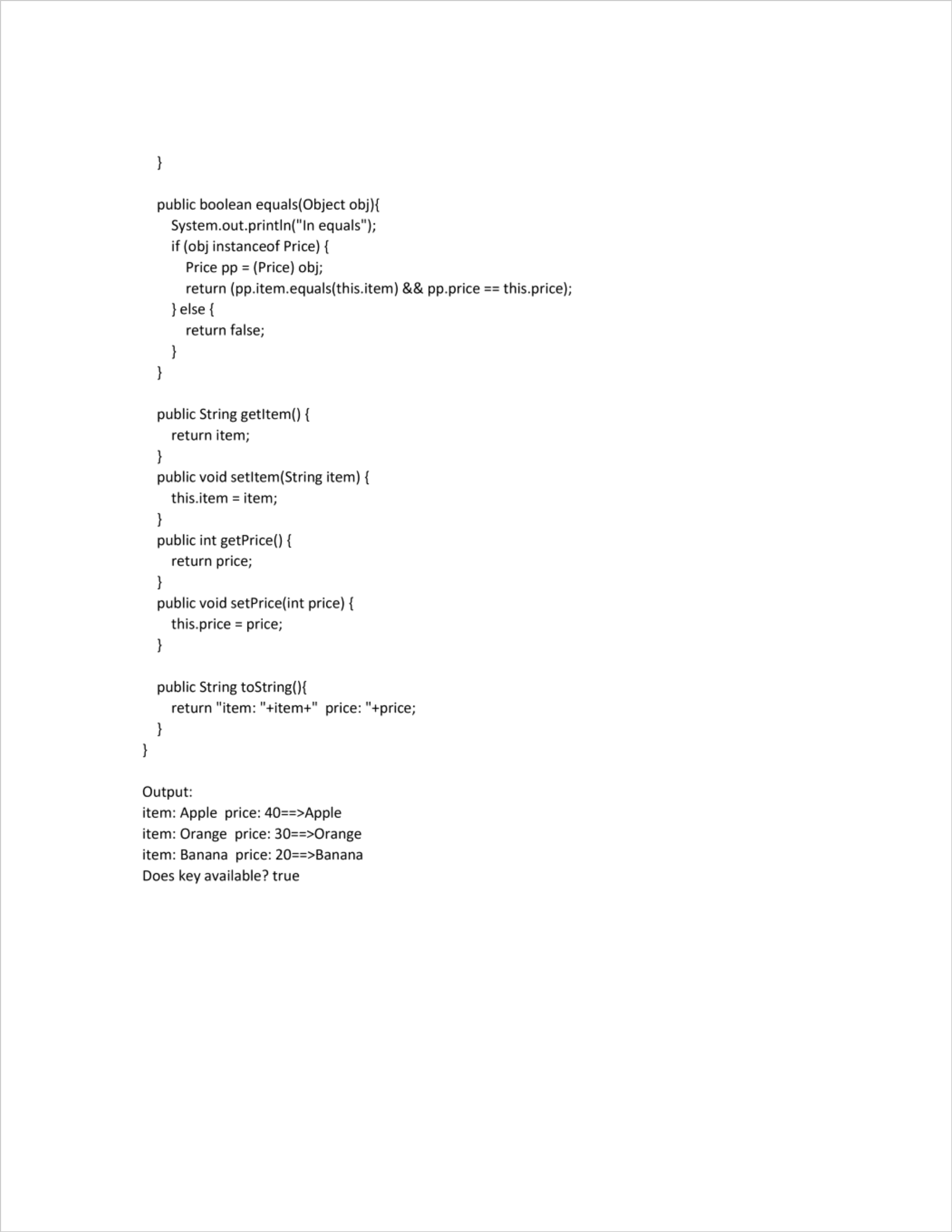
Output:

```
item: Apple  price: 40==>Apple
item: Orange  price: 30==>Orange
item: Banana  price: 20==>Banana
Deleting key...
After deleting key:
item: Apple  price: 40==>Apple
item: Orange  price: 30==>Orange
```



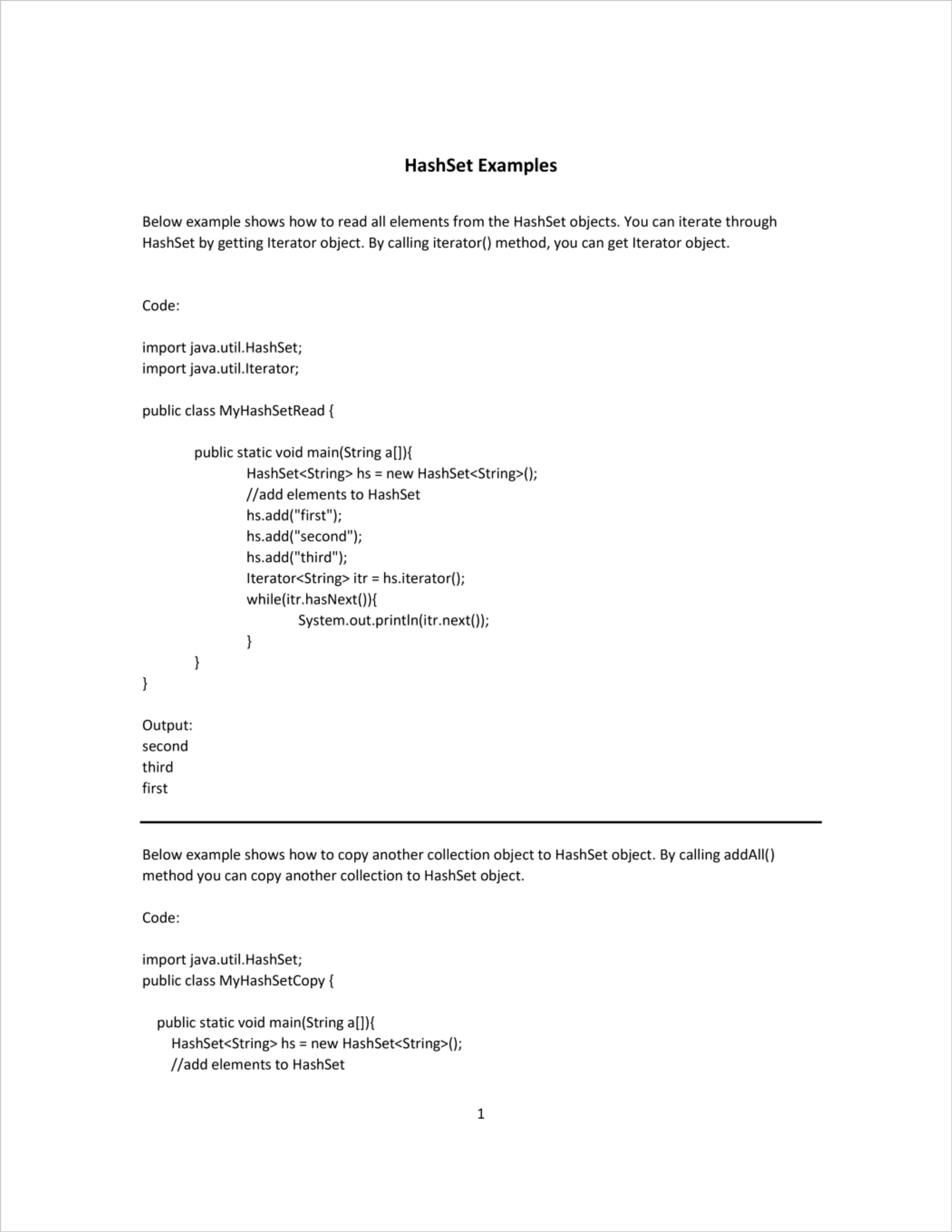
Below example shows how to search user defined objects as a key from HashMap. You can achieve this by implementing equals and hashcode methods at the user defined objects.

```
import java.util.HashMap;
import java.util.Set;

public class MyObjectKeySearch {

public static void main(String a[]){

        HashMap<Price, String> hm = new HashMap<Price, String>();
        hm.put(new Price("Banana", 20), "Banana");
        hm.put(new Price("Apple", 40), "Apple");
        hm.put(new Price("Orange", 30), "Orange");
        printMap(hm);
        Price key = new Price("Banana", 20);
        System.out.println("Does key available? "+hm.containsKey(key));
    }

    public static void printMap(HashMap<Price, String> map){

        Set<Price> keys = map.keySet();
        for(Price p:keys){
            System.out.println(p+"==>"+map.get(p));
        }
    }
}

class Price{

    private String item;
    private int price;

    public Price(String itm, int pr){
        this.item = itm;
        this.price = pr;
    }

    public int hashCode(){
        System.out.println("In hashcode");
        int hashcode = 0;
        hashcode = price*20;
        hashcode += item.hashCode();
        return hashcode;
```



```
    }

    public boolean equals(Object obj){
        System.out.println("In equals");
        if (obj instanceof Price) {
            Price pp = (Price) obj;
            return (pp.item.equals(this.item) && pp.price == this.price);
        } else {
            return false;
        }
    }

    public String getItem() {
        return item;
    }
    public void setItem(String item) {
        this.item = item;
    }
    public int getPrice() {
        return price;
    }
    public void setPrice(int price) {
        this.price = price;
    }

    public String toString(){
        return "item: "+item+"  price: "+price;
    }
}
```

```
Output:
item: Apple  price: 40==>Apple
item: Orange  price: 30==>Orange
item: Banana  price: 20==>Banana
Does key available? true
```



HashSet Examples

Below example shows how to read all elements from the HashSet objects. You can iterate through HashSet by getting Iterator object. By calling iterator() method, you can get Iterator object.

Code:

```
import java.util.HashSet;
import java.util.Iterator;

public class MyHashSetRead {

        public static void main(String a[]){
                HashSet<String> hs = new HashSet<String>();
                //add elements to HashSet
                hs.add("first");
                hs.add("second");
                hs.add("third");
                Iterator<String> itr = hs.iterator();
                while(itr.hasNext()){
                        System.out.println(itr.next());
                }
        }
}
```

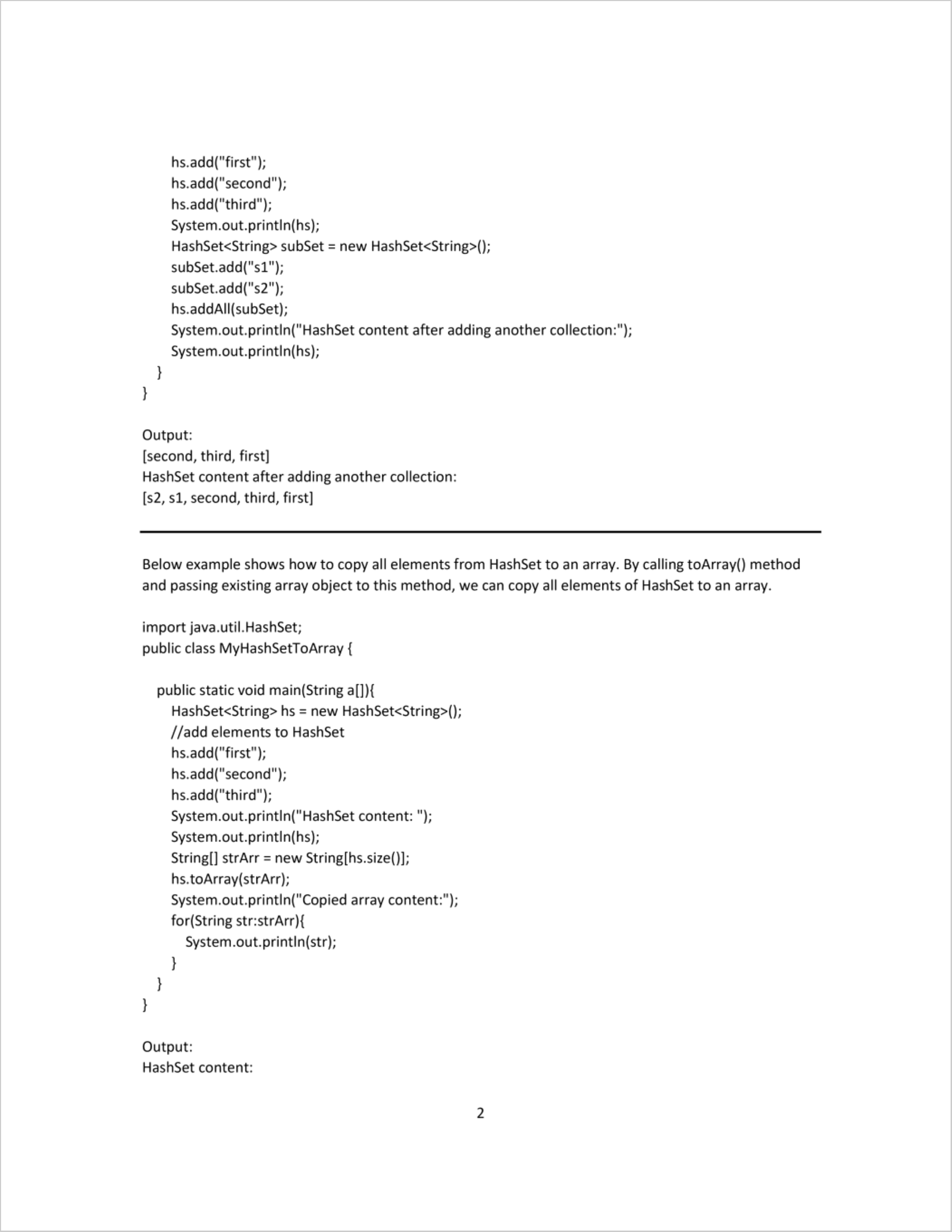
Output:

```
second
third
first
```

Below example shows how to copy another collection object to HashSet object. By calling addAll() method you can copy another collection to HashSet object.

Code:

```
import java.util.HashSet;
public class MyHashSetCopy {

   public static void main(String a[]){
      HashSet<String> hs = new HashSet<String>();
      //add elements to HashSet
```



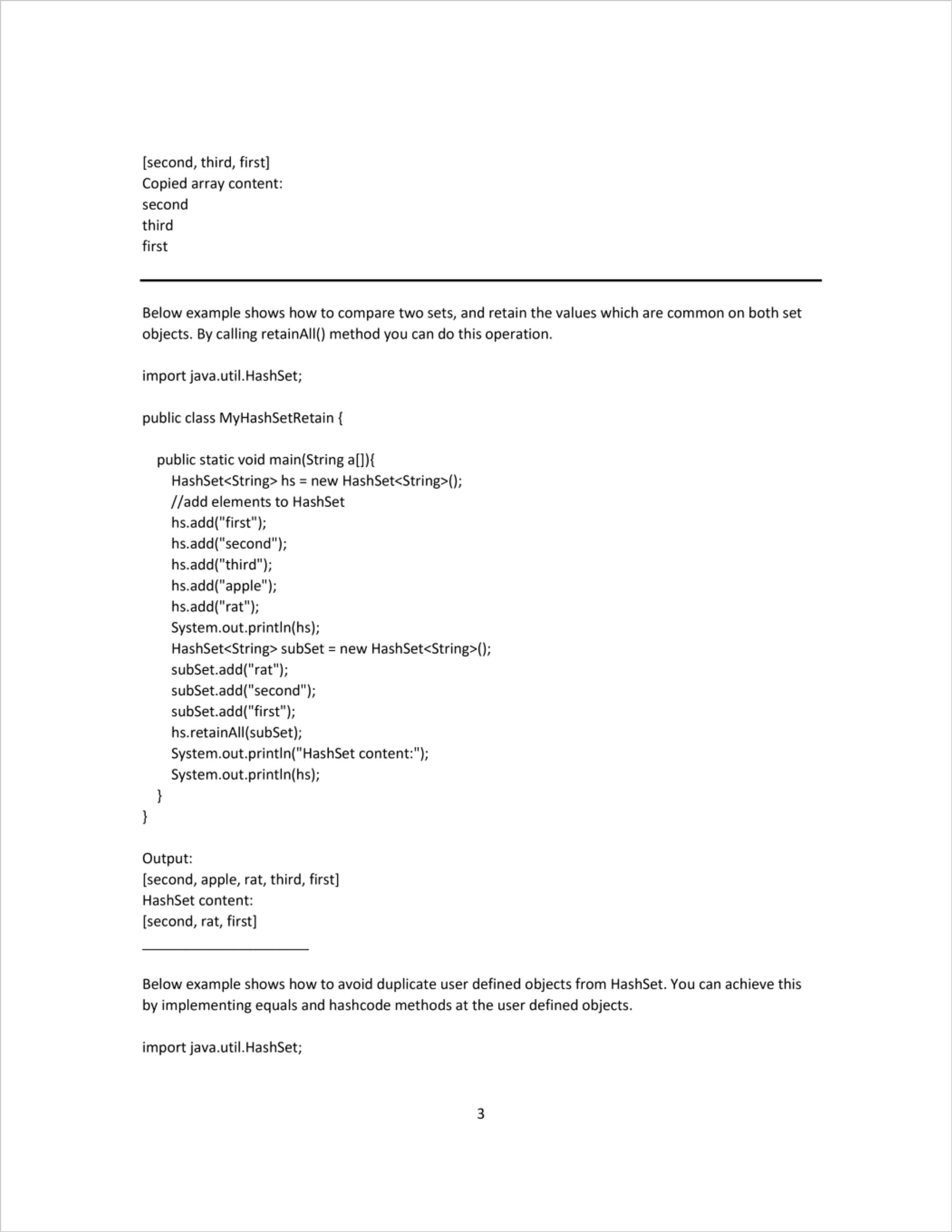
```
        hs.add("first");
        hs.add("second");
        hs.add("third");
        System.out.println(hs);
        HashSet<String> subSet = new HashSet<String>();
        subSet.add("s1");
        subSet.add("s2");
        hs.addAll(subSet);
        System.out.println("HashSet content after adding another collection:");
        System.out.println(hs);
    }
}
```

```
Output:
[second, third, first]
HashSet content after adding another collection:
[s2, s1, second, third, first]
```

Below example shows how to copy all elements from HashSet to an array. By calling toArray() method and passing existing array object to this method, we can copy all elements of HashSet to an array.

```
import java.util.HashSet;
public class MyHashSetToArray {

    public static void main(String a[]){
        HashSet<String> hs = new HashSet<String>();
        //add elements to HashSet
        hs.add("first");
        hs.add("second");
        hs.add("third");
        System.out.println("HashSet content: ");
        System.out.println(hs);
        String[] strArr = new String[hs.size()];
        hs.toArray(strArr);
        System.out.println("Copied array content:");
        for(String str:strArr){
            System.out.println(str);
        }
    }
}
```

```
Output:
HashSet content:
```



```
[second, third, first]
Copied array content:
second
third
first
```

Below example shows how to compare two sets, and retain the values which are common on both set objects. By calling retainAll() method you can do this operation.

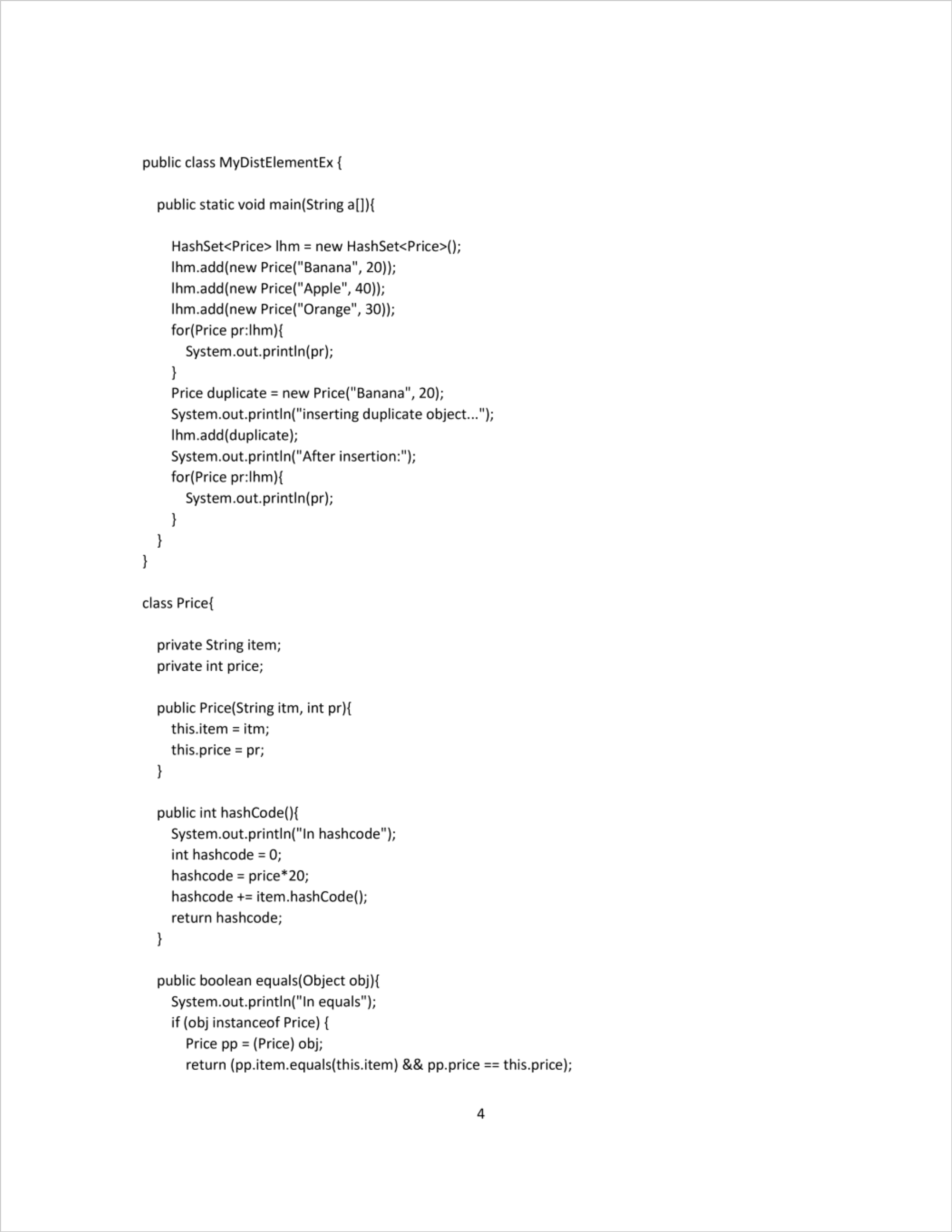
```
import java.util.HashSet;

public class MyHashSetRetain {

   public static void main(String a[]){
      HashSet<String> hs = new HashSet<String>();
      //add elements to HashSet
      hs.add("first");
      hs.add("second");
      hs.add("third");
      hs.add("apple");
      hs.add("rat");
      System.out.println(hs);
      HashSet<String> subSet = new HashSet<String>();
      subSet.add("rat");
      subSet.add("second");
      subSet.add("first");
      hs.retainAll(subSet);
      System.out.println("HashSet content:");
      System.out.println(hs);
   }
}
```

Output:

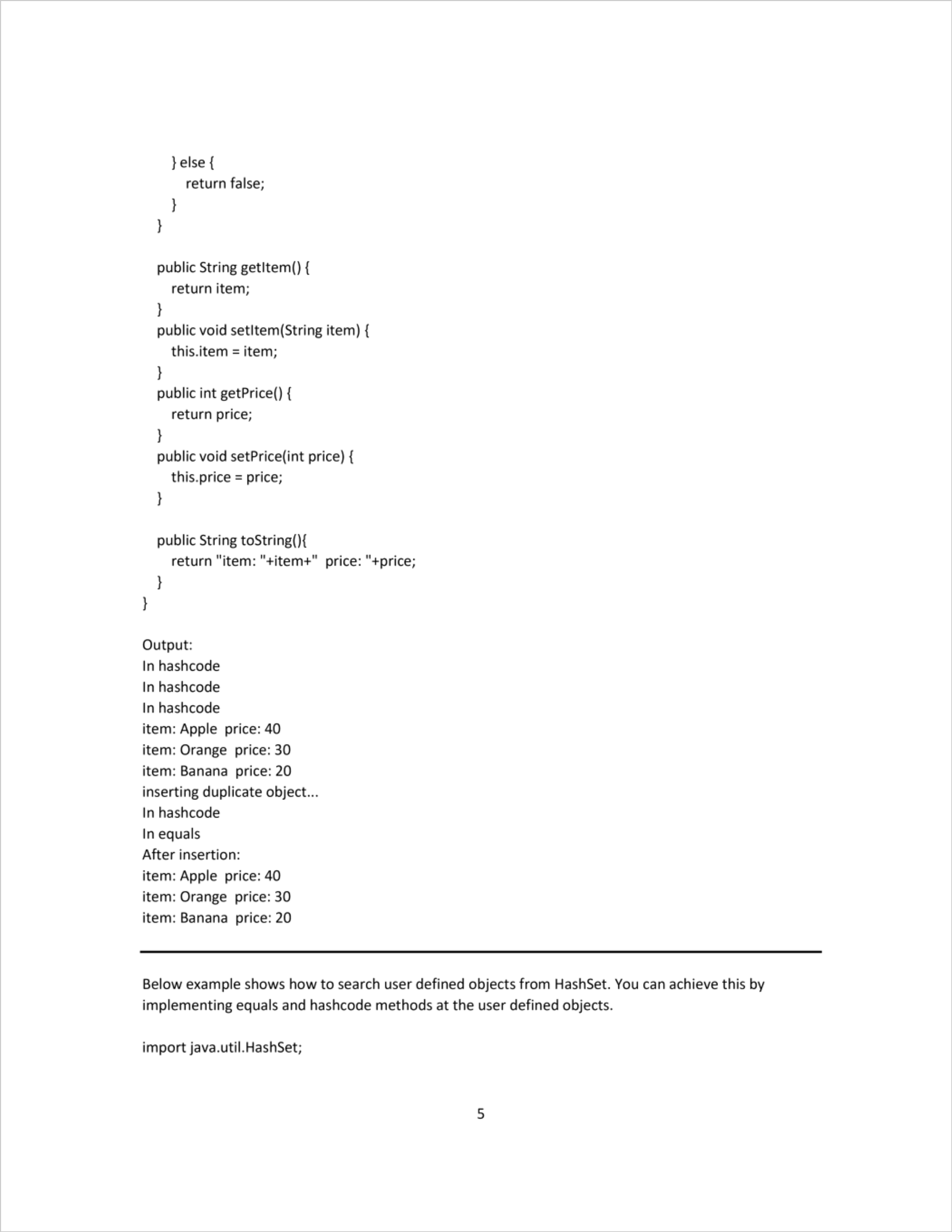
```
[second, apple, rat, third, first]
HashSet content:
[second, rat, first]
```

Below example shows how to avoid duplicate user defined objects from HashSet. You can achieve this by implementing equals and hashcode methods at the user defined objects.

```
import java.util.HashSet;
```



```
public class MyDistElementEx {

   public static void main(String a[]){

      HashSet<Price> lhm = new HashSet<Price>();
      lhm.add(new Price("Banana", 20));
      lhm.add(new Price("Apple", 40));
      lhm.add(new Price("Orange", 30));
      for(Price pr:lhm){
         System.out.println(pr);
      }
      Price duplicate = new Price("Banana", 20);
      System.out.println("inserting duplicate object...");
      lhm.add(duplicate);
      System.out.println("After insertion:");
      for(Price pr:lhm){
         System.out.println(pr);
      }
   }
}

class Price{

   private String item;
   private int price;

   public Price(String itm, int pr){
      this.item = itm;
      this.price = pr;
   }

   public int hashCode(){
      System.out.println("In hashcode");
      int hashcode = 0;
      hashcode = price*20;
      hashcode += item.hashCode();
      return hashcode;
   }

   public boolean equals(Object obj){
      System.out.println("In equals");
      if (obj instanceof Price) {
         Price pp = (Price) obj;
         return (pp.item.equals(this.item) && pp.price == this.price);
```

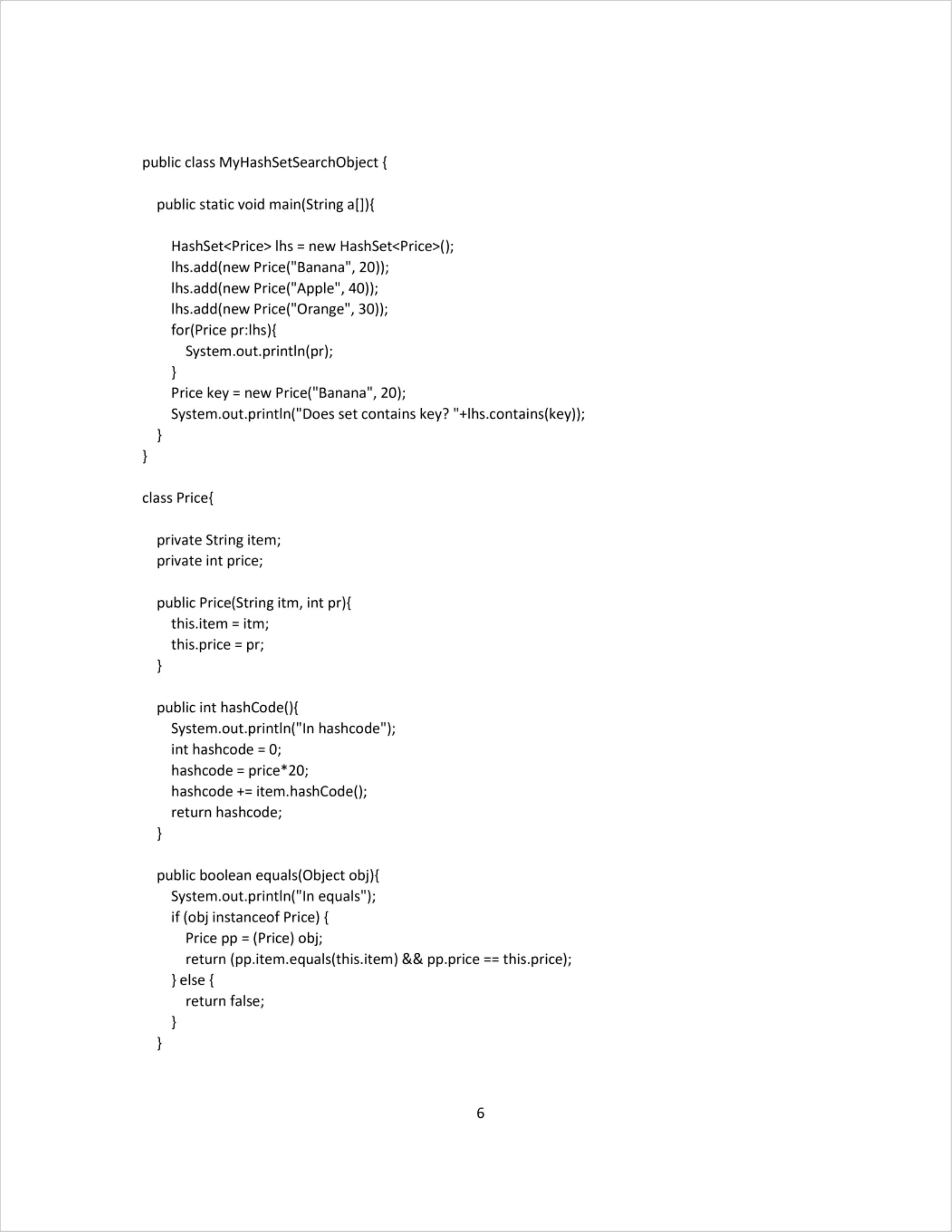


```
        } else {
            return false;
        }
    }

    public String getItem() {
        return item;
    }
    public void setItem(String item) {
        this.item = item;
    }
    public int getPrice() {
        return price;
    }
    public void setPrice(int price) {
        this.price = price;
    }

    public String toString(){
        return "item: "+item+"  price: "+price;
    }
}
```

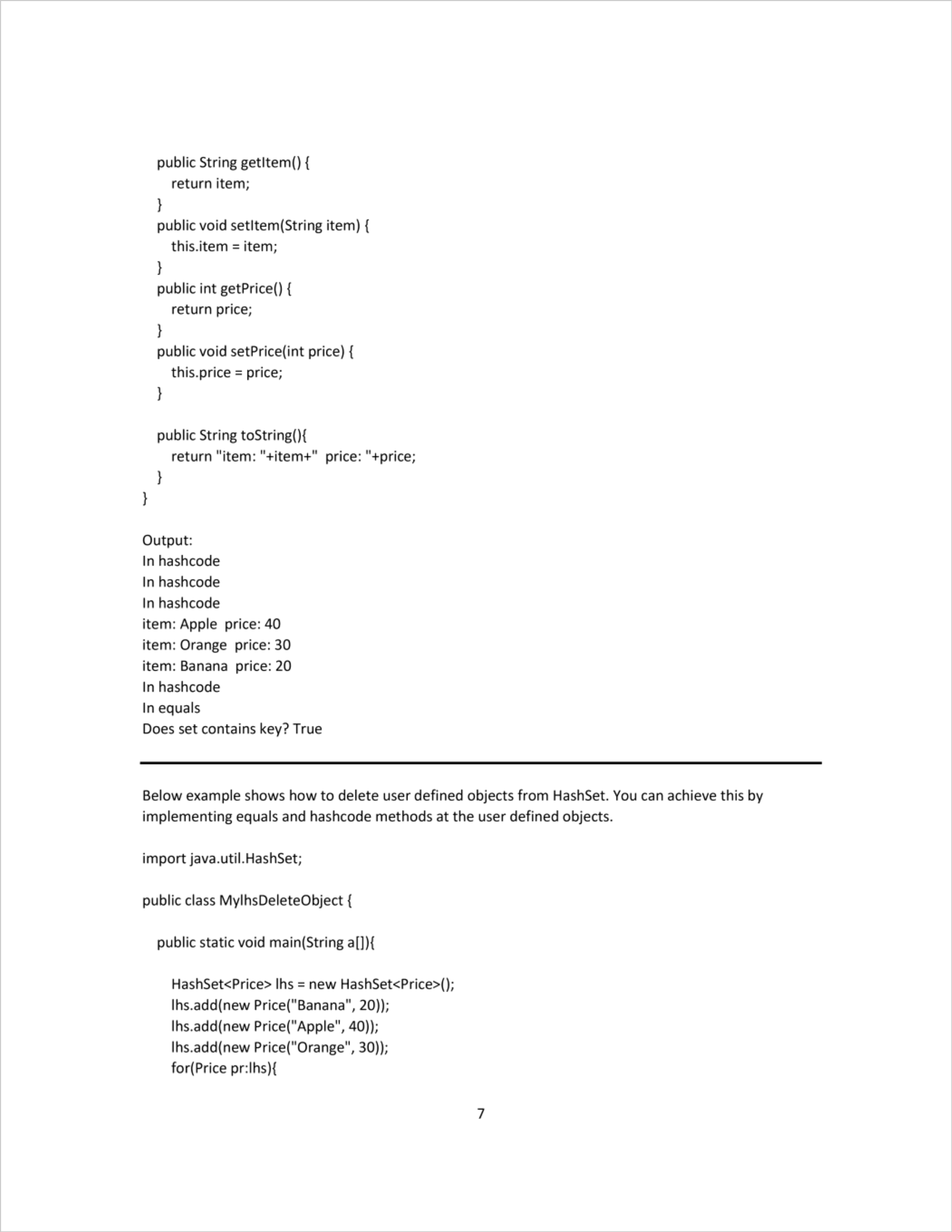
```
Output:
In hashcode
In hashcode
In hashcode
item: Apple  price: 40
item: Orange  price: 30
item: Banana  price: 20
inserting duplicate object...
In hashcode
In equals
After insertion:
item: Apple  price: 40
item: Orange  price: 30
item: Banana  price: 20
```

Below example shows how to search user defined objects from HashSet. You can achieve this by implementing equals and hashcode methods at the user defined objects.

```
import java.util.HashSet;
```



```
public class MyHashSetSearchObject {

   public static void main(String a[]){

      HashSet<Price> lhs = new HashSet<Price>();
      lhs.add(new Price("Banana", 20));
      lhs.add(new Price("Apple", 40));
      lhs.add(new Price("Orange", 30));
      for(Price pr:lhs){
         System.out.println(pr);
      }
      Price key = new Price("Banana", 20);
      System.out.println("Does set contains key? "+lhs.contains(key));
   }
}

class Price{

   private String item;
   private int price;

   public Price(String itm, int pr){
      this.item = itm;
      this.price = pr;
   }

   public int hashCode(){
      System.out.println("In hashcode");
      int hashcode = 0;
      hashcode = price*20;
      hashcode += item.hashCode();
      return hashcode;
   }

   public boolean equals(Object obj){
      System.out.println("In equals");
      if (obj instanceof Price) {
         Price pp = (Price) obj;
         return (pp.item.equals(this.item) && pp.price == this.price);
      } else {
         return false;
      }
   }
```

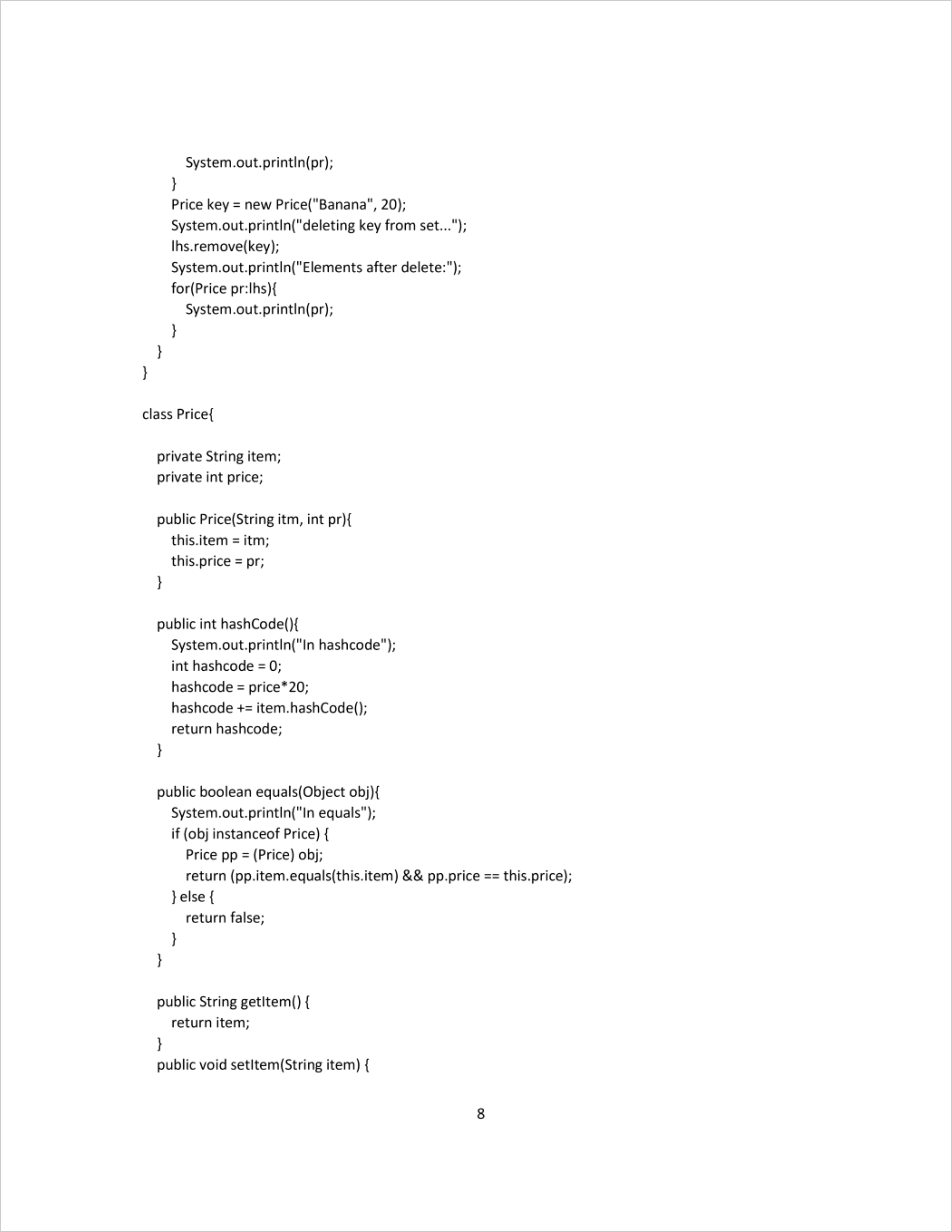


```
  public String getItem() {
    return item;
  }
  public void setItem(String item) {
    this.item = item;
  }
  public int getPrice() {
    return price;
  }
  public void setPrice(int price) {
    this.price = price;
  }

  public String toString(){
    return "item: "+item+"  price: "+price;
  }
}
```

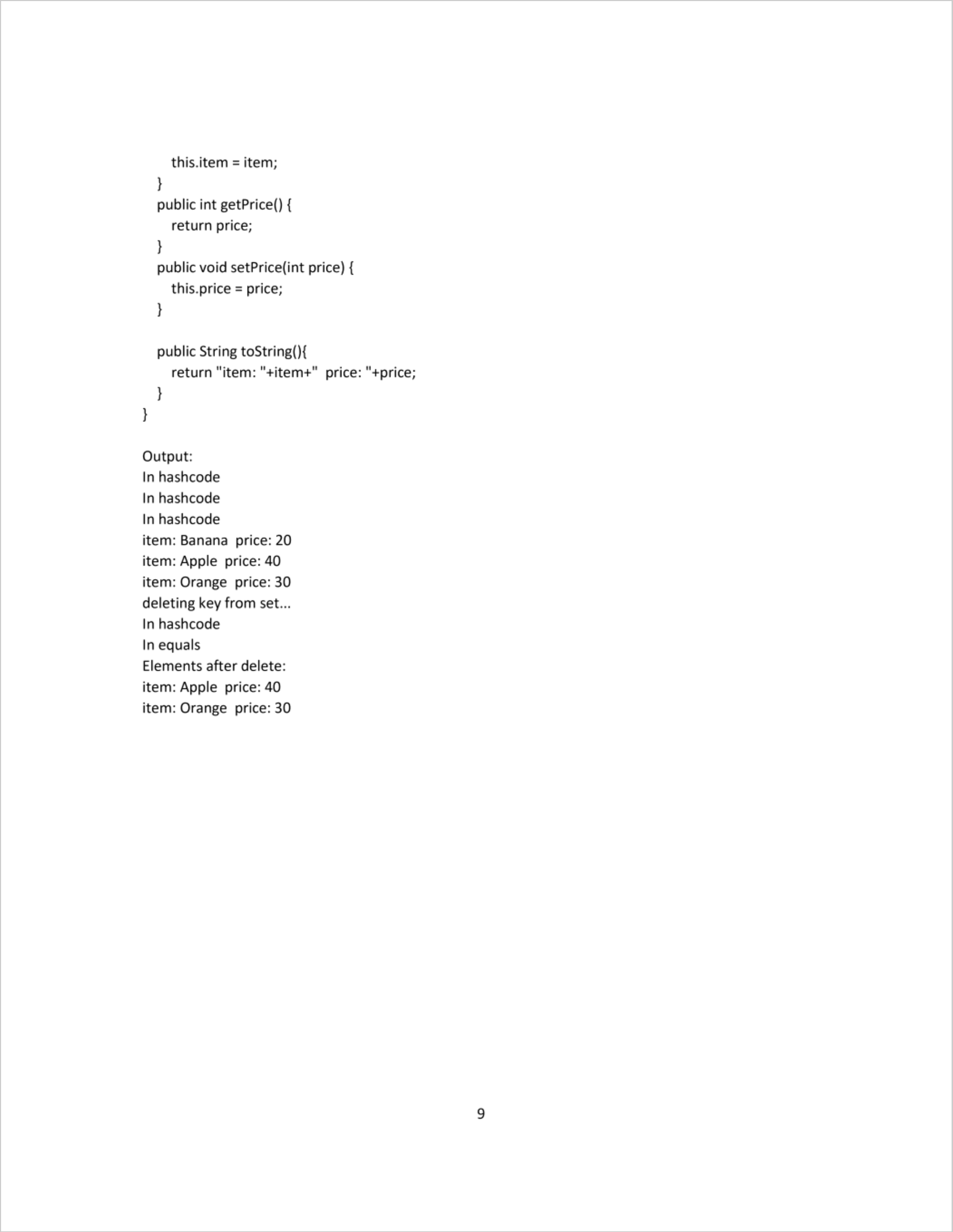
```
Output:
In hashcode
In hashcode
In hashcode
item: Apple  price: 40
item: Orange  price: 30
item: Banana  price: 20
In hashcode
In equals
Does set contains key? True
```

Below example shows how to delete user defined objects from HashSet. You can achieve this by implementing equals and hashcode methods at the user defined objects.

```
import java.util.HashSet;

public class MylhsDeleteObject {

  public static void main(String a[]){

    HashSet<Price> lhs = new HashSet<Price>();
    lhs.add(new Price("Banana", 20));
    lhs.add(new Price("Apple", 40));
    lhs.add(new Price("Orange", 30));
    for(Price pr:lhs){
```



```
         System.out.println(pr);
      }
      Price key = new Price("Banana", 20);
      System.out.println("deleting key from set...");
      lhs.remove(key);
      System.out.println("Elements after delete:");
      for(Price pr:lhs){
         System.out.println(pr);
      }
   }
}

class Price{

   private String item;
   private int price;

   public Price(String itm, int pr){
      this.item = itm;
      this.price = pr;
   }

   public int hashCode(){
      System.out.println("In hashcode");
      int hashcode = 0;
      hashcode = price*20;
      hashcode += item.hashCode();
      return hashcode;
   }

   public boolean equals(Object obj){
      System.out.println("In equals");
      if (obj instanceof Price) {
         Price pp = (Price) obj;
         return (pp.item.equals(this.item) && pp.price == this.price);
      } else {
         return false;
      }
   }

   public String getItem() {
      return item;
   }
   public void setItem(String item) {
```

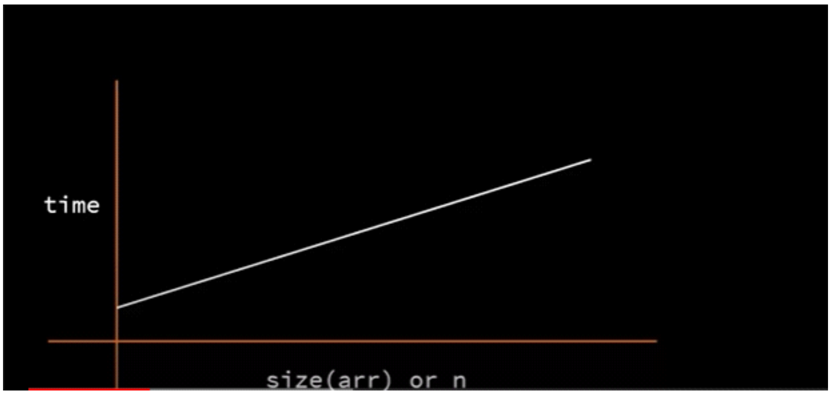


```
        this.item = item;
    }
    public int getPrice() {
        return price;
    }
    public void setPrice(int price) {
        this.price = price;
    }

    public String toString(){
        return "item: "+item+"  price: "+price;
    }
}
```

```
Output:
In hashcode
In hashcode
In hashcode
item: Banana  price: 20
item: Apple  price: 40
item: Orange  price: 30
deleting key from set...
In hashcode
In equals
Elements after delete:
item: Apple  price: 40
item: Orange  price: 30
```

Time Complexity

11 May 2022 06:27



The time depends on CPU -> so measuring time in terms of sec is not useful. So we should represent in terms of mathematical function.

For constant function: a-> time is O(1) -> executing the method -> declare the method and represent the time taken

```
For(i=0;i<n;i++){ -> executes n+1
Sysout  -> executes n times
}
Total -----------------> 2n+1 -> remove constants
```

Example: a*n+b; here n is the length so generally we take time as A(n)

```
For(i=o;i<n;i++){ -----executes for n+1 times
  for(j=0;j<n;j++){ --------executes for n(n+1)
      Sysout -------------n*n
      }
}
```

Total time taken is --------> n+1 +(n^2+n)+n^2 => 2n^2+2n+2 ==> remove constant and keep higher term so it will be O(n^2)
For loop inside for loop
The time complexity will be time= a*n^2+b; -> b is constant so we can remove and notation Is O(n^2)

Bubblesort logic -> n^2
Afterwards printing elements -n
Then total time we consider as -> a*n2+b*n+c;

So from this remove the constant and remove the fastest growing term. O(n^

2)
Why to keep fastest growing term.
F= 5n^2+3n+2; O(n) - notation is rough estimate of term. So we consider the max time taken by the loop
In this above example if we take the n value as 1000
f= 5*1000^2+3(1000)+2
So here compared to 5*1000^2 -> the remaining term is small so remove it hence O(n^2)

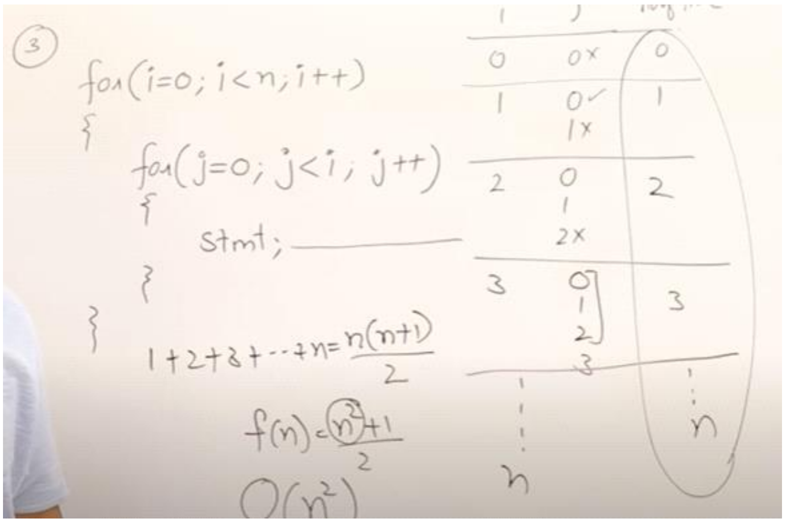
Liner Search
Search for number write 1 for loop and check each number so loop gets executed for n times so we can say time taken is O(n)

Binary Search:
Take 8 elements -> search for last but 1-> Time taken is first iteration will be -> n/2
Second iteration ->n/2/2-> n/2^2
Third iteration: n/2/2/2-> n/2^3

Iteration k= n/2^k
2^k=n
Log 2^k=logn
k=long(n)
Time taken is O(logn)

Linear search best case is -> searching the element at first position so -> O(1)
Worst case is O(n)-> Searching for Last element

Binary Search or Binary search tree-> best case is -> searching root element or middle element ->O(1)
Worst case is -> Searching for Leaf or Last element O(logn)



③

```
for(i=0; i<n; i++)
{
    for(j=0; j<i; j++)
    {
        stmt;
    }
}
```

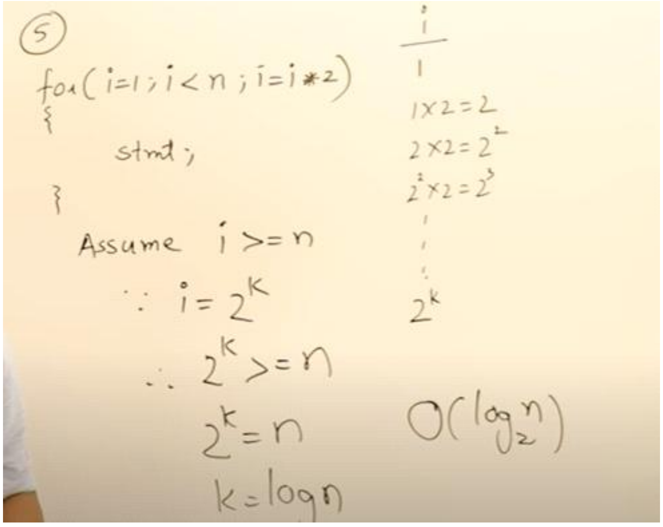
i	j	
0	0×	0
1	0✓ 1×	1
2	0 1 2×	2
3	0 1 2 3	3
⋮		⋮
n		n

$1+2+3+\dots+n = \frac{n(n+1)}{2}$

$f(n) = \frac{n^2+1}{2}$

$O(n^2)$

Here sum of n numbers is n*(n+1)/2 ==> n^2+n/2



⑤

```
for(i=1; i<n; i=i*2)
{
    stmt;
}
```

Assume $i >= n$

$\because i = 2^k$

$\therefore 2^k >= n$

$2^k = n$

$k = \log n$

i
1
$1\times2=2$
$2\times2=2^2$
$2^2\times2=2^3$
⋮
2^k

$O(\log_2 n)$

Interview Questions

16 January 2022 16:52

- What is the internal data structure used by ArrayList ? **:**
  - **Object[]**
- How does add and remove method of array list works ?
  - Add(): check capacity-> if it is full increment by 50% and then add
  - Increment of array using arrays.copy()
- Write the custom arraylist ?
  - Create the class, constructor should initialize the array, later implement add , remove, get methods
  - Remove(): index is >size of array -> throw exception
      - If not do left shift operation-> system.arrayCopy
      - How to know howmany shift operation-> Length of array-positiontoBeDeleted-1
  - Get(): idex>size-> throw exception
    Value=Array[requestedSize]
- What is the internal data Structure used by Hash Map ?
  - Node[]
- What is collision in hashMap ?
  - When we insert multiple objects using put method, if the index is determined as same index for multiple put methods then it can be treated as collision. To overcome this, we use LinkedList concept,-> this linkedlist has performance issue->in 1.8 it is overcome by Tree approach
- What is the internal implementation of hashMap ?
  - How does add() will work :-> first find the hashcode of key
    Then get the index
    Then create node object-> which has hashcode, key, value and addressofNext node
- What is the difference between Array Blocking Queue and Linked Blocking Queue?
  - Refer to the notes
- What is Priority Queue ?
  - Priority queue follows natural sorting order, it uses comparator internally.
  - PriorityQueue<Integer> p= new PriorityQueue<>();
  - p.add(10);
  - p.add(05);
  - p.add(11);
  - Sysout(p) ->
  - To know the insertion order on queue call the poll method
- What is the difference between array d queue and priority queue ?
  - Operations can be Performed from both the ends where as priority queue follows natural sorting order and only one ended queue
  - pollFirst()->similar to poll() method
  - pollLast-> pop() method
  - addFIrst()
  - addLast()
  - peekFirst()-> similar to peek method in Queue
  - peekLast()->similar to peek method of Stack
- What is Stack and how can I implement Stack ?
  - Stack is extending from Vector -> Since vector is synchronized stack is also synchronized

    - Pop() ->returns the last element of stack and removes it
    - Peek()-> returns the last element but not deletion
    - Push()-> add the element at last
    - Difference of add() and push() method-> add()- returns void ->add method is from vector
    - Push() method return boolean- from Stack
    - Pop and push methods on empty Stack- >Empty Stack exception
    - Custom Stack-> using array ->Object[] a=null; -> initialize the object[] a in constructor
            - Add()->create index variable -> capacity check->a[index++]=obj;
            - Peek()->if() length ==0 -> throw empty stack else return a[a.length-1];
            - Pop() ->if() length ==0 -> throw empty stack else return a[a.length-1] and perform assign null to last position and reduce the size and reduce index;
- What is Queue and how can I implement Queue ?
    - Queue is an interface
        - Poll()->returns the first element and deletes it ->
            - If the Queue size is 5 - and poll() is called ->size will become 4->
        - Peek()->return the first element
        - Push()->add the element
        - Performing poll and peek on empty Queue -> null
        - Custom Queue : ->Object[] a=null; -> initialize the object[] a in constructor
            - Add()->create index variable -> capacity check->a[index++]=obj;
            - poll()->if() length ==0 -> return null else return a[0] and perform shift operation
            - Peek() ->if() length ==0 -> returns null else return a[0];
- Write a program to implement Stack and Queue using linked list ?
    - Instead of Object[] -> we use Node class
- Write a program to sort hashMap by values without using tree Map?
    - Map<Integer,String> m= new HashMap<>();
    - m.put(10,"A");
    - m.put(5,"B");
    - m.put(11,"C");
    - Write a program to sort this hashmap by keys ->[[5,"B"],[10,"A"],[11,"C"]]
    - Map<Integer, String > m1= new TreeMap<>(m);
    - Sysout(m1);
    - Another method -> Set<Entry<Integer,String>> s1=m.entrySet();
    - Refer to the program shared in code
- What is the difference between hash set and tree set ?
    - Hashset is not ordered -> it uses hash map
    - Tree set follow order-> natural sorting order it used tree Map internally
- What are the difference between tree map and hash map ?
    - HashMap is not ordered ->
    - Tree Map follow order-> natural sorting order
- What is copy on write Array set ?
    - To achieve the concurrency and overcome concurrent modification set we go for copy on write array set
- What is concurrent modification exception ?
    - While iterating and perform structure modification then concurrent modification exception is thrown
- What is the internal implementation of concurrent modification exception ?
    - modCount variable is used-> next ()method checks current modcount and expected modunt if both are not matching then it throws concurrent modification
- Write a program to swap adjacent nodes in a Linked List

- Write a code to reverse a Linked List from position X to position Y
- For a given Linked List, write a code to return the node value where the cycle in question begins
- Which type of partition Lomuto or Hoare Partition in Quick Sort. This is Morgan Stanley Interview question

- What is Tree ?
  - Tree is one Data structure. We use Node as a class which has left and right
- What is complete Binary Search Tree ?
  - It is a tree in which, left node should be lessthan the root node and right node should be greaterthan equal the root node
- What is Full Binary Tree?
  - A tree which should have all left and right nodes.
- What is complete binary tree ?
  - If the height of tree is h-> h-1 level it should be full binary tree and at h level nodes should fill from left
- What is Heap ?
  - Heap is nothing but complete binary tree
- What is Min Heap ?
  - Min Heap is a complete binary tree in which root element should be lesser than child
- What is Max Heap ?
  - Max Heap is a complete binary tree in which root element should be greater than child
- How to represent min heap and max heap ?
  - First find out parent positions. If the array length is is N then -> 0 to n/2-1 positions will be parents afterwards child
  - To prepare Max or Min heap from the given array we need to start from n/2-1 position and for each parent get the LC and RC -> LC: 2*i+1; RC: 2*i+2 where "i" the parent position and this formula holds correct when array starts with index as zero
  - After that swap the root with greater child. Repeat the same process untill all the roots have greater elements than child --> This is for Max Heap
  - Same process for min heap-> only the difference is, findout the lower element among child and replace it with root
- How to represent Tree in Java ?
- What are tree traversal techniques ?
- What is inOrder() write code for inorder ?
  - Left->Root->Right
- What is preOrder() write code for it ?
  - Root->Left->Right
- What is postOrder() write code for it ?
  - Left->Right->Root
- What is Graph ?
  - Graph is a DS which has Vertex and Edge
- How graph is represented in Java ?
  - To represent Graph, Adjacency List - List[]
- What is DFS ? To represent in DFS what data structure is used ?
  - DFS:->Graph traversal technique-> visit any random vertex, from there take the adject and then explore -> Stack is used
- What is BFS ? To represent in BFS what data structure is used ?
  - BFS:->Graph traversal technique-> visit any random vertex, and explore adjcent vertex immediately -> Queue is used