Properties :
---
public class Properties extends Hashtable<K,V>

It is a legacy class and It represents a persistent set of properties.

It is subclass of Hashtable available in java.util package.

It is used to maintain the persistent data in the key-value form. It takes both key and value as a String format.

It is used to load properties file in our java application directly at runtime without re-compilation/deployment/ and re-starting the server.

Constructors :
---
Commonly we are using this constructor :

```
Properties p1 = new Properties();
Creates an empty property list.
```

Methods :
---
1) public void load(InputStream stream): Reads a property list (key and value pair) from the input byte stream.

2) public void load(Reader reader):Reads a property list (key and value pair) from the Character Oriented stream.

3) Object setProperty(String key, String value) : It Calls the Hashtable method put internally to store the key and value pair in String format.

4) public String getProperty(String key) :Searches for the property with the specified key in this property list.

5) public void store(OutputStream out, String comments) : It
Writes this property list (key and element pairs) in this Properties table to the output stream.

6) public void store(Writer writer, String comments) : It
Writes this property list (key and element pairs) in this Properties table to the character stream.

Steps :
---
1) Create a file whose extension must be .properties as shown below

db.properties
---
driver = oracle.jdbc.driver.OracleDriver
user = scott
password = ravi

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

public class PropertiesDemo1
{
    public static void main(String[] args) throws Exception
    {
        FileReader fr = new FileReader("D:\\new\\db.properties");

        Properties prop = new Properties();
        prop.load(fr);

      String driver = prop.getProperty("driver");
        System.out.println(driver);

      String user = prop.getProperty("user");
        System.out.println(user);

      String pwd = prop.getProperty("password");
        System.out.println(pwd);
    }
}
```

If we make changes in the db.properties file then directly (without compilation) we can take the the value in our java file so after any modification in the properties file we need not to re-compile/re-deploy our java program.

---

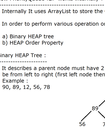
```
package com.ravi.properties;

import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.util.Properties;

public class PropertiesDemo2
{
    public static void main(String[] args) throws IOException
    {

        Properties properties = new Properties();


        String filePath = "D:\\new\\bookdata.properties";

        var writer = new FileWriter(filePath);
        try(writer)
        {
            properties.setProperty("book", "Java");
            properties.setProperty("author", "James");
            properties.setProperty("price", "1200");

            properties.store(writer, "Book Properties set");


        }
        catch(Exception e)
        {
            e.printStackTrace();
        }

        //Reading the data from Properties file

        var reader = new FileReader(filePath);

        try(reader)
        {
            properties.load(reader);
            System.out.println("Book Name is :"+properties.getProperty("book"));
            System.out.println("Author Name is :"+properties.getProperty("author"));
            System.out.println("Price Name is :"+properties.getProperty("price"));
        }
        catch(Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

---

Queue<E> interface :
---
* It is predefined interface which is the sub interface of Collection available from JDK 1.5V.

* It is the sub interface of Collection hence It supports all the methods of Collection interface.

* It works on FIFO (First In First Out) basis.




RRS

* It is an ordered collection.

* In a queue, insertion is possible from last is called REAR end whereas deletion is possible from the starting is called FRONT end of the queue.

* In order to support Basis Queue operation, LinkedList class implements Deque and Deque interface extends Queue interface.

PriorityQueue<E>
---
public class PriorityQueue<E> extends AbstractQueue<E> implements Serializable

* It is an implemented class of Queue<E> interface available from JDK 1.5V.

* It provides default natural sorting order by using **BINARY HEAP tree** that means elements are sorted based on the binary heap tree, the highest priority element will be in the head of the queue OR element will be inserted by using Comparator<E> which is provided as a constructor at the time of Queue construction.

* It does not accept null and non comparable objects.

* The default capacity is 11.

* It is not synchronized so synchronization we can use PriorityBlockingQueue which is available in java.util.concurrent sub package.

How PriorityQueue<E> works internally ?
---
* Internally It uses ArrayList to store the Object but using Binary HEAP tree.

* In order to perform various operation on PriorityQueue object It uses the following two concepts

a) Binary HEAP tree
b) HEAP Order Property

Binary HEAP Tree :
---
* It describes a parent node must have 2 child node as well as the insertion of the element must be from left to right (first left node then right node)
Example :
90, 89, 12, 56, 78




HEAP Order Property :
---
* It describes the parent node must have highest priority than its child node.
* Always the comparison will be done between parent and child, we will never compare two child node.
Example :
90, 89, 12, 56, 78

| In ArrayList form : | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| | 90 | 89 | 12 | 56 | 78 |

Same element in the Binary HEAP tree with HEAP order property

Binary HEAP Tree
= HEAP Order Property




In the same binary tree I want to insert 95 (adding an element)




At the time insertion we have good time complexity because element will be added in the last node of the queue nothing but at the last position of the ArrayList

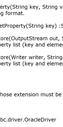
I want to remove an element from the Queue.




If we want to remove the element from the Queue then elements are extracted from the head of the queue (the element which is having highest Priority).

* Deleting 12 from the queue will provide O(n) time complexity so to solve this issue we will swap first and last element.


Now we can delete by using O(1) time complexity but HEAP order property volates so re-arrange





Constructor :
---
1) PriorityQueue pq1 = new PriorityQueue();
Will create PriorityQueue object with default capacity is 11, Elements will be inserted based on binary heap tree.

2) PriorityQueue pq2 = new PriorityQueue(int initialCapacity);
Will create PriorityQueue with user specified capacity

3) PriorityQueue pq3 = new PriorityQueue(int initialCapacity, Comparator cmp);
Will create PriorityQueue with user specified capacity and own userdefined order.

4) PriorityQueue pq4 = new PriorityQueue(Comparator cmp);
Will create PriorityQueue with user defined sorting order

5) PriorityQueue pq5 = new PriorityQueue(Collection c);
Loose coupling

6) PriorityQueue pq6 = new PriorityQueue(PriorityQueue<E> p);

7) PriorityQueue pq7 = new PriorityQueue(SortedSet<E> p);