java.io package

File Class

- Java File class represents the files and directory pathnames in an abstract manner.
- This class is used for creation of files and directories, file searching, file deletion etc.
- The File object represents the actual file/directory on the disk.
- There are following constructors to create a File object:

- File(String pathname)
- File(File parent, String child);
- File(String parent, String child)

SN	Methods with Description
•	**public String getName()** Returns the name of the file or directory denoted by this abstract pathname.
•	**public String getParent()** Returns the pathname string of this abstract pathname's parent.
•	**public String getPath()** Converts this abstract pathname into a pathname string.
•	**public String getAbsolutePath()** Returns the absolute pathname string of this abstract pathname.
•	**public boolean canRead()** Returns true if and only if the file specified by this abstract pathname exists and can be read by the application; false otherwise.
•	**public boolean canWrite()** Returns true if and only if the file system is allowed to write to the file; false otherwise.
•	**public boolean isDirectory()** Returns true if and only if the file denoted by pathname exists and is a directory; false otherwise.
•	**public boolean isFile()** Returns true if and only if the file denoted by pathname exists and is a normal file; false otherwise.
•	**public long lastModified()** Returns a long value representing the time the file was last modified, measured in milliseconds since the epoch (00:00:00 GMT, January 1, 1970), or 0L if the file does not exist or if an I/O error occurs.
•	**public long length()** Returns the length of the file denoted by this pathname.
•	**public boolean delete()** It returns true and delete the specified file or folder if it is found.
•	**public void deleteOnExit()** Requests that the file or directory denoted by this abstract pathname be deleted when the virtual machine terminates.
•	**public String[] list()** Returns an array of strings naming the files and directories in the directory denoted by this abstract pathname.
•	**public String[] list(FilenameFilter)** Returns an array of strings naming the files and directories in the directory denoted by this abstract pathname that satisfy the specified filter.
•	**public File[] listFiles()** Returns an array of abstract pathnames denoting the files in the directory denoted by this abstract pathname.
•	**public File[] listFiles(FileFilter)** Returns an array of abstract pathnames denoting the files and directories in the directory denoted by this abstract pathname that satisfy the specified filter.
•	**public boolean mkdir()** It returns true and create the directory if parent directory is found. It creates always only one directory.
•	**public boolean mkdirs()** Returns true if and only if the directory was created, along with all necessary parent directories; false otherwise.
•	**public boolean renameTo(File dest)** Returns true if and only if the renaming succeeded; false otherwise.
•	**public boolean setLastModified(long time)** Sets the last-modified time of the file or directory named by this abstract pathname.
•	**public static File createTempFile(String prefix, String suffix, File directory) throws IOException** Creates a new empty file in the specified directory, using the given prefix and suffix strings to generate its name.

Example (Basic file operations)

```
import java.io.File;
import java.util.Date;
class FileOprations
{
	public static void main(String []args){

	File f = new File(args[0]);
	System.out.println(f.isFile());
	System.out.println(f.isDirectory());
	System.out.println(f.isHidden());
	System.out.println(f.canRead());
	System.out.println(f.canWrite());
	System.out.println(f.canExecute());
	System.out.println(f.isAbsolute());
	System.out.println(f.getName());
	System.out.println(f.getPath());
	System.out.println(f.getAbsolutePath());
System.out.println(Math.ceil((float)f.length()/1024)
+" KB");
System.out.println(new Date(f.lastModified()));
	}
}
```

Example (Operating System operations)

```
import java.io.File;
import java.util.Date;
class TestDemo
{
	public static void main(String []args)
	{
		File f1 = new File(args[0]);
		File f2 = new File(args[1]);
		System.out.println(f1.renameTo(f2));
		//System.out.println(f1.mkdir());
		//System.out.println(f1.mkdirs());
		//System.out.println(f1.delete());
	}
}
```

Example (Listing file from specified directory)

```
import java.io.*;
import java.util.Date;
class TestDemo
{
	public static void main(String []args){
		File f = new File(args[0]);
		if(f.isDirectory())
		{
			String name[] = f.list();
			for(String n : name)
				System.out.println(n);
		}
	}
}
```

Explain File Filter and FilenameFilter with Example.

FilenameFilter

- FilenameFilter is an interface in Java that is used to filter file names, such as those returned from a call to a File object's listFiles() or list() method.
- If listFiles() is called with no parameters, it returns all File objects in a directory.
- If we pass in a filter as a parameter, we can selectively return a subset of those objects.
- Creating an object that implements FilenameFilter requires us to implement the **public boolean accept(File dir, String name)** method.
- The dir object is the parent directory of the file, and name is the name of the file.
- If accept() returns true, the file will be returned in the array of File objects from the call to listFiles().
- If accept() returns false, the file isn't returned by the call to listFiles().

Example (Listing java files)

```
import java.io.*;
import java.util.Date;
class TestDemo
{
        public static void main(String []args){
                File f = new File(args[0]);

    if(f.isDirectory()) {

        File name[] = f.listFiles(new FilenameFilter()
        {
         public boolean accept(File path,String name)
        {
                File f1 = new File(path,name);
                //return f1.isHidden();
                return name.endsWith(".java");
        }
        });

   for(File n : name)
     System.out.println(n.getName()+" "+n.length());

} // end of if

} // end of main
}//end of class
```

FileFilter

- The FileFilter interface in java can be used to filter the files from a given directory.
- Instances of this interface may be passed to the listFiles(FileFilter) method of the File.
- It has abstract methods

public boolean accept(File pathname)

```
import java.io.*;
import java.util.Date;
class TestDemo
{
        public static void main(String []args) {
        File f = new File(args[0]);

        if(f.isDirectory())
        {
        File name[] = f.listFiles(new FileFilter()
        {
                public boolean accept(File f1)
                {
                        return f1.length()>2048;
                }
        });

         for(File n : name)
            System.out.println(n.getName()+"
"+n.length());
        } // end of if
        } // end of main
}
```

Example (Listing Images)

```
import java.io.*;
class ImageListing implements FileFilter
{
   private final String[] okFileExtensions =
   new String[] {"jpg", "png", "gif"};

   public boolean accept(File file)
   {
      for (String extension : okFileExtensions)
      {
         if(file.getName().endsWith(extension))
            return true;
      }
      return false;
   }
}
```

STREAM CLASSES

- Java encapsulates Stream under **java.io** package.
- Java defines two types of streams. They are,

1. **Byte Stream :** It provides a convenient means for handling input and output of byte.
2. **Character Stream :** It provides a convenient means for handling input and output of characters. Character stream uses Unicode and therefore can be internationalized.

Byte Stream

1) InputStream : It is used to read data in byte stream.
2) OutputStream : It is used to write data in byte stream.

Character Stream

1) Reader : It is used to read data in char stream.
2) Writer : It is used to write data in char stream.

- OutputStream and Writer classes have write methods to write the data to some destination.
- InputStream and Reader similarly have read methods to read data from the source.
- All the stream classes implements Closeable interface, which has only one method called close().
- It is used to release any system resources which are used by the stream.

Explain RandomAccessFile with Example.

- A FileInputStream can be used to read from a file, and a FileOutputStream can be used to write to a file.
- We may be interested in reading and writing to the same file.
- For this, we have a class called RandomAccessFile which can be used for reading and writing to the file.
- RandomAccessFile implements Closable, DataInput and DataOutput interface.

Constructors of RandomAccessFile

1. RandomAccessFile(String file, String mode)
2. RandomAccessFile(File file, String mode)

- Constructor takes name of the file as a parameter and mode as a parameter.
- mode could be "r", "rw" or "rwd"

Methods of RandomAccessFile

public void seek(long position)
public long getFilePointer()
public long length()

- RandomAccessFile maintains a file pointer.
- The file pointer indicates the position within the file where the next read or write operation will be done.
- Initially file pointer is at 0 position.
- The getFilePointer() method returns the current value of the file pointer.
- Current value of file pointer can be changed by using the seek() method.
- length() method returns size of the file in bytes.

Example (RandomAccessFile)

```
import java.io.*;
class  TestDemo {
        public static void main(String []args)
throws Exception {

        RandomAccessFile r =
            new RandomAccessFile(args[0],"r");

        String str;
        r.seek(10);
        System.out.println(r.length());

        System.out.println(r.getFilePointer());
        while((str=r.readLine())!=null)
                System.out.println(str);

        System.out.println(r.getFilePointer());
        }
}
```

Explain ObjectOutputStream and ObjectInputStream OR Explain Serialization in detail.

- ObjectOutputStream is directly sub classed from the OutputStream class.
- It implements ObjectOutput interface.
- Constructor of ObjectOutputStream requires an OutputStream as a parameter.

ObjectOutput interface
- It extends the DataOutput interface.
- It inherits all the methods of DataOutput interface and has one additional method called ***writeObject().***

- ObjectOutputStream class has the capabilities of handling all the primitive data types similar to the DataOutputStream.
- Additionally it can also handle Objects.
- The writeObject() method gives the capability of persisting objects.

Serialization
- Serialization is the process of converting an object into a sequence of bytes such that those bytes can be used to recreate the object in same state.
- Serialization is the process of making the object's state is persistent.
- That means the state of the object is converted into stream of bytes and stored in a file.
- In the same way we can use the de-serilization concept to bring back the object's state from bytes.
- This is one of the important concept in Java programming because this serialization is mostly used in the network programming.
- Only instance of classes implementing the Serializable interface may be serialized using writeObject() method.
- Serializable is a marker interface without any methods.

transient
- The keyword transient in Java used to indicate that the variable should not be serialized.
- By default all the variables in the object is converted to persistent state.
- In some cases, you may want to avoid persisting some variables because you don't have the necessity to transfer across the network.
- So, you can declare those variables as transient.
- If the variable is declared as transient, then it will not be persisted. It is the main purpose of the transient keyword.
- For example in Account class has an instance variable for PIN number. This is secure information and we would not like this to be persisted using the writeObject() method. This can be done by declaring such fields to be transient.

ObjectInputStream
- It is directly sub-classed from InputStream class.
- It implements ObjectInput interface.
- Constructor of ObjectInputStream requires InputStream as a parameter.

ObjectInput interface
- This interface extends the DataInput interface.
- ObjectInput interface inherits all the methods from the DataInput interface and has one additional method called readObject().
- ObjectInputStream class has the capabilities of handling all the primitive data types similar to the DataInputStream.
- Additionally it can also handle Objects.
- readObject() method is used for deserializing an object.

Example on next page

Example

```
class Employee implements Serializable
{
        private transient int empid;
        private String name;
        private int salary;
        Employee(){}
     Employee(int empid,String name,int salary){
                this.empid=empid;
                this.name=name;
                this.salary=salary;
        }
        public String toString()
        {
        return empid+" "+name+" "+salary+"\n";
        }
}

class TestEmployee
{
    public static void main(String []args)
    throws Exception
    {
Employee e1 = new Employee(101,"Ram",590);
Employee e2 = new Employee(102,"Sita",470);
Employee e3 = new Employee(103,"Lax",450);

//Serialization
ObjectOutputStream oos = new
ObjectOutputStream(new
FileOutputStream(args[0]));

        oos.writeObject(e1);
        oos.writeObject(e2);
        oos.writeObject(e3);
        oos.flush();
        oos.close();

//Deserialization
        ObjectInputStream ois  = new
ObjectInputStream(new
FileInputStream(args[0]));
Employee e4 = (Employee)ois.readObject();
Employee e5 = (Employee)ois.readObject();
Employee e6 = (Employee)ois.readObject();
System.out.println(e4+""+e5+""+e6);
        }
}
```

- In the above example, the variable empid is declared as *transient*, so it will not be stored in the persistent storage.

Write a code to copy content from one fileto another file.

```
import java.io.*;
import java.util.*;
class CopyFile
{
        public static void main(String []args)
throws Exception
        {
        InputStream i =
              new FileInputStream(args[0]);

        byte[] b = new byte[i.available()];
        i.read(b);
        FileWriter f = new FileWriter(args[1]);
        f.write(new String(b));
        f.close();
        }
}
```

Write a code to read content from file.

```
import java.io.*;
class TestDemo
{
    public static void main(String []args)
    throws Exception
    {
            File f = new File(args[0]);
             if(f.exists())
             {
                    FileInputStream in =
                       new FileInputStream(f);
                    int ch;
                    while((ch=in.read())!=-1)
                       System.out.print((char)ch);
              }
    }
}
```

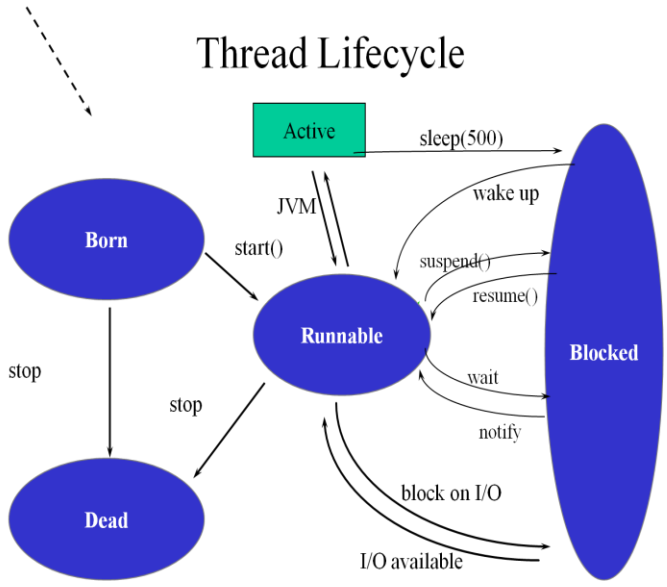
Write a Java code to display content of file with line number and display only even lines of given file.

```
import java.io.*;

class FileReadDemo3{
  public static void main(String[] args) throws
Exception {
   FileInputStream fis = new
                        FileInputStream(args[0]);
   LineNumberReader lr = new
   LineNumberReader(new InputStreamReader
                          (fis));
    String line;
          while((line=lr.readLine())!=null){
          if(lr.getLineNumber()%2==0)
System.out.println(lr.getLineNumber()+" :"
                                          +line);
           }
          fis.close();
          }
}
```

Explain Thread Life Cycle with Example Program. OR Explain Thread States.

```
class Test extends Thread
{
        Test()
        {
                super(); // NEW or BORN
                start();
        }
        //Runnable State
        public void run()
        {
            try{
        Thread.sleep(5000); // BLOCKED
             }catch(InterruptedException e){}
        }
        //DEAD
}
```



NEW State: Initially, when a new instance of Thread is created, it is in NEW or BORN state.

RUNNABLE State : When we call start() method on the instance of Thread, the state changed to RUNNABLE. Initially Runnable State represents the state when thread is either ready for execution.

BLOCKED State: When we call sleep, supend or wait method thread will be in BLOCKED state, after calling resume or after completion of sleep thread will be again in Runnable State.

DEAD State: After completion of run() method, Thread will be in Dead State.

Explain ways to create a Thread.

There are three ways to create a Thread

1. extends Thread
2. implements Runnable
3. Anonymously

1) extends Thread

```
import java.util.Date;
class Test extends Thread
{
    Test(){
                super("Clock Thread");
                start();
     }
     public void run()
     {
          while(true){
          System.out.println(new Date());
                try{
                Thread.sleep(1000);
                }catch(Exception e){}
           }
     }
}
class TestDemo
{
          public static void main(String []args)
          {
                Test t = new Test();
          }
}
```

2) Runnable Example

```
import java.util.Date;
class Test implements Runnable
{
     Test(){}
     public void run()
     {
          while(true){
             System.out.println(new Date());
             try{
              Thread.sleep(1000);
               }catch(Exception e){}
           }
    }
}
class TestDemo
{
          public static void main(String []args)
          {
                Test t = new Test();
                Thread t1 = new Thread(t);
                t1.start();
          }
}
```

3) Creating Thread Anonymously

```
import java.util.Date;
class TestDemo
{
     public static void main(String []args)
     {
        Thread t = new Thread()
        {
              public void run()
              {
               while(true){
               System.out.println(new Date());
               try{
                    Thread.sleep(1000);
                  }catch(Exception e){}
               } // End of while
              }
        };
        t.start();
    }
}
```

Explain Synchronization with Example.

- There are two types of thread synchronization
  - mutual exclusive and
  - inter-thread communication.

- Mutual Exclusive helps keep threads from interfering with one another while sharing data. This can be done by three ways in java:
  - by synchronized method
  - by synchronized block
  - by static synchronization

- Synchronization is built around an internal entity known as the lock or Monitor.
- Every object has a lock associated with it.
- By convention, a thread that needs consistent access to an object's fields has to acquire the object's lock before accessing them, and then release the lock when it's done with them.

Example

```
import java.util.*;
class Table{
synchronized public static void printTable(int n)
{

        for(int i=1;i<=5;i++)
        {
           System.out.print(i*n+" ");
            try{
                Thread.sleep(1000);
             }catch(InterruptedException e){}
        }
        System.out.println();
}
}
class Test extends Thread
{
        Test()
        {
                super("MyThread1");
                start();
        }
        public void run()
        {
                Table.printTable(5);
        }
}
class Test1 extends Thread
{
        Test1()
        {
                super("My Thread2");
                start();
        }
        public void run()
        {
                Table.printTable(15);
        }
}
class TestDemo
{
        public static void main(String args[])
        {
                Test t = new Test();
                Test1 t1 = new Test1();
        }
}
```

Q.11 Explain Use of join() method with Example.

- The join() method waits for a thread to die. In other words, it causes the currently running threads to stop executing until the thread it joins with completes its task.
- Syntax
  - public void join() throws InterruptedException
  - public void join(long milliseconds) throws InterruptedException

```
class Test extends Thread
{
        Test(String name)
        {
                super(name);
                start();
        }
        public void run()
        {
                for(int i=1;i<=5;i++){
                    try{
                        Thread.sleep(1500);
                    }catch(Exception e){}
                }
        }
}
class TestDemo
{
        public static void main(String []args)
        {
                Test t1 = new Test("One");
                Test t2 = new Test("Two");
                Test t3 = new Test("Three");
                try{
                t1.join();
                t2.join();
                t3.join();
                }catch(Exception e){}
               System.out.println("Main Exit");
        }
}
```

Main Thread will wait for all child thread to finish.

Q.12. Differentiate Daemon Thread Vs Non-Daemon Thread

Daemon Thread	Non-Daemon Thread
These threads are generally known as a "Service Provider" thread.	These threads are generally known as a "User thread".
These threads should not be used to execute program code but system code.	These threads are created by programmer. It generally meant to run program code.
These thread run in parallel to your code but JVM can kill them anytime. When JVM finds no user threads, it stops and all daemon thread terminate instantly.	JVM doesn't terminates unless all the user thread terminate.
We can set non-daemon thread to daemon using **setDaemon(true)**	We can set daemon thread to non-daemon using **setDaemon(false)**

System.out.println(i+" : "+this);