Day15: File and IO-stream, Serialization, Java NIO package

File class in Java:

The **File** class belongs to **java.io** package.

This class is used to know the characteristics of something present in the hard-disk.

This **java.io.File** class has many functionalities using which we can identify the features of file or folder/directory present in the given location.

Example:

```
File f = new File("abc.txt");
```

The above line will not create a new physical file.

If the file "abc.txt" is already exist in the current folder then f will point the already existing file. if the file is not present the f will represent simply name of the file.

```
    File f=new File("abc.txt");

    System.out.println(f); //abc.txt

    System.out.println(f.exists()); //false

    f.createNewFile(); // it will create a abc.txt file in the current location

    System.out.println(f.exists()); //true

    File f2=new File("d://myfiles//abc.txt");

    f2.createNewFile();// if d://myfiles location is not there
                       //then it will throw an exception
```

Note:- if the file is already there, it instead of creating a new file, it will points to the existing file only.

A file object can be used for a folder(directory) also.

Example:

```
File f=new File("d://myfiles2");

f.mkdir(); //to create folder

System.out.println(f);//  d:\myfiles2

System.out.println(f.exists()); //true
```

File class constructors:

```
File f = new File(String fname); //we can provide relative of absolute path.

File f=new File(String subdir,String fname);

File f =new File(File subdir,String fname);
```

Some of the methods of the File class:

Method name	Description
boolean exist()	Tests whether the file or directory exists.
boolean createNewFile() throws IOException	Atomically creates a new, empty file.
boolean mkdir()	Creates the directory/folder.
boolean isFile()	Tests whether the file denoted by this is a normal file.
boolean isDirectory()	Tests whether the file denoted by this is a directory/folder.
long length();	Returns the length of the file(how many characters are there)
String[] list()	list out all the files and folders present in the specified directory in the form of String array (only the file name without the path).
File[] listFiles()	list out all the files and folders present in the specified directory in the form of File array (file names with the full path).
String getName()	return the name of the file.
String getParent()	It returns the pathname string of this pathname's parent, or null if this pathname does not name a parent directory.

Example: creating a new file if it is not present:

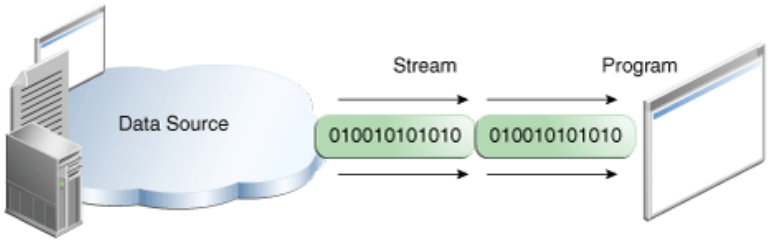
```
import java.io.*;
public class Main {
    public static void main(String[] args) {

        try {
            File file = new File("a1.txt");
            if (file.createNewFile()) {
                System.out.println("New File is created!");
            } else {
                System.out.println("File already exists.");
            }
        } catch (IOException e) {
            e.printStackTrace();
        }

    }
}
```

IO-Stream:

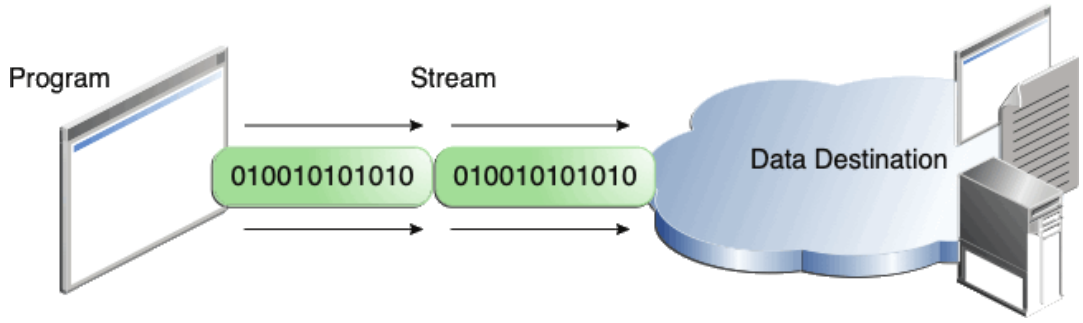
An *I/O Stream* represents an input source or an output destination. A stream can represent many different kinds of sources and destinations, including disk files, devices, other programs, and memory arrays.

It is basically a continuous flow of data from one place to another, here the flow of data is in between the program and the peripherals(devices, other programs, etc.)

Reading data into a program from the peripherals:



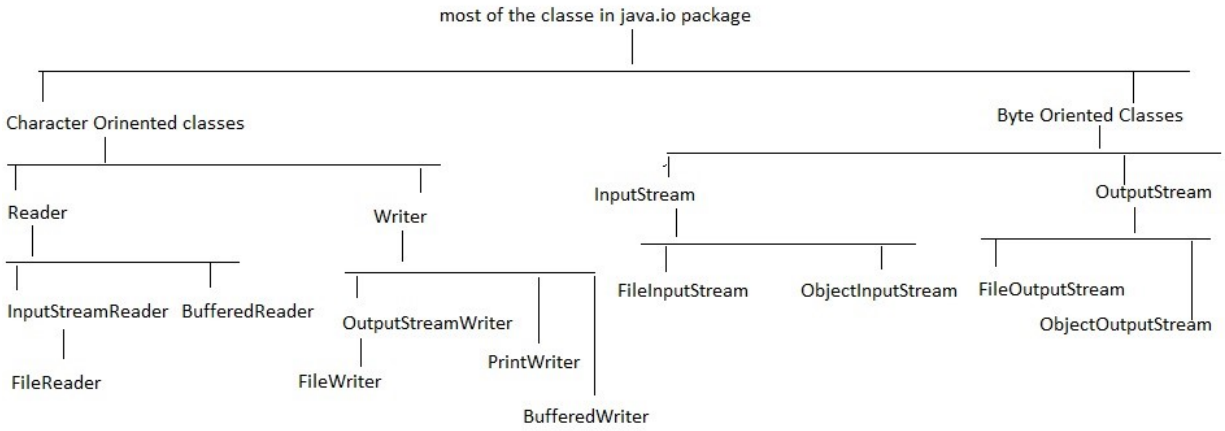
Writing data from the program into the peripherals:



For data transfer to be done, first of all the data should be represented to its low level equivalent either in the form of **bytes(8 bit)** or in the form of **character(16 bit)**.

Depending upon the representation of the data most of the classes in the **java.io** package is classified into two categories:

1. **Byte oriented classes**
2. **Character oriented classes**



FileWriter class:

Java FileWriter class is used to write character-oriented data to a file. It is character-oriented class which is used for file handling in java.

.

Note:- the main limitation of the FileWriter class is, while writing the data in a file we need to insert the line separator manually("\n") or ("\r\n"), depending upon different Operating System.

Example:

```
import java.io.*;
public class Main {

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

        FileWriter fw = new FileWriter("d://abc.txt");

        fw.write(100);//d will be added
        fw.write("ramesh\nwelcome");
        fw.write("\n");
        fw.write("india");
        fw.write("\n");
        char ch[]={'a','b','c'};
        fw.write(ch);

        fw.flush();
        fw.close();

       System.out.println("done");

    }
  }


Output:
The above program will create a new file abc.txt inside the D:// drive and write
dramesh
welcome
india
abc
```

FileReader class:

We can use this class to read the data from the file in the form of Character.

Constructor of FileReader class:

- **FileReader(String file);** // It gets filename in string. It opens the given file in read mode. If file doesn't exist, it throws FileNotFoundException.
- **FileReader(File file);** // It gets filename in File instance. It opens the given file in read mode. If file doesn't exist, it throws FileNotFoundException.

Methods of FileReader class:

- **int read();** // it attempts to read the next character from the file and returns its value(unicode). if the next character is not available then it will return -1. (EOF)

  Example:

  abc ⇒ 97 98 99 -1

- **int read(char[] ch);** // it attempts to read the enough character from the file into the character array and returns the number of characters copied.

Example1: reading from the file

```
import java.io.*;
public class Main {

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

        FileReader fr=new FileReader("d://abc.txt");

        int i=fr.read();

        while( i != -1) {
            System.out.print((char) i);
            i = fr.read();
        }
    }
}
```

Example2: reading from the file (another approach)

```
import java.io.*;
public class Main {

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

        File f=new File("d://abc.txt");

        FileReader fr=new FileReader(f);

        char[] chr=new char[(int)f.length()];

        fr.read(chr);

        for(char c:chr){
            System.out.print(c);
        }
    }
}
```

Note: the main limitation of the FileReader is we have to read character by character which is not convenient to the programmer and it is not recommended with respect to performance also.

To overcome the problem of FileWriter and FileReader we should use the **BufferedReader** and the **BufferedWriter** class.

BufferedWriter class:

The BufferedWriter class is used to writer the character data to the file. It makes the performance fast compare to the FileWriter class.

Compare to the FileWriter class this BufferedWriter class provides **newLine()** method support to insert a new line into the file. and this **newLine()** method is independent to different Operating System.

Note: The BufferedWriter class never communicate directly with the file. it communicate with the file via some other Writer object.

Example:

```
BufferedWriter bw = new BufferedWriter(Writer writer);

ex:

BufferedWriter bw = new BufferedWriter("abc.txt"); //error

BufferedWriter bw = new BufferedWriter(new File("abc.txt")); //error

BufferedWriter bw = new BufferedWriter(new FileWriter("abc.txt")); //OK
```

Example:

```
import java.io.*;
public class Main {

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

        FileWriter fw = new FileWriter("d://abc.txt");

        BufferedWriter bw = new BufferedWriter(fw);

        bw.write(100);//d will be added
        bw.newLine(); //inserting a new line
        bw.write("ramesh");
        bw.newLine();
        bw.write("india");
        bw.newLine();
        char ch[]={'a','b','c'};
        bw.write(ch);
```

```
        bw.flush();
        bw.close();

        System.out.println("done");

    }
}
```

BufferedReader class:

With the help of BufferedReader class we can read character data from the file.

The main advantage of BufferedReader over the FileReader is we can read the data line by line instead of character by character.

So, internally it gives better performance.

Note:-BufferedReader also can't communicate directly with the file, it will communicate with file via some Reader object.

Example:

```
BufferedReadr br=new BufferedReader(Reader r);

ex:

BufferedReadr br=new BufferedReader(new FileReader("abc.txt"));
```

Methods of BufferedReader class:

int read();

int read(char ch[]);

void close();

String readLine();//it attempts to read the next line from the destination. if the next line is not available then we will return a null value.

Example:

```
import java.io.*;
public class Main {

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

        FileReader fr=new FileReader("d://abc.txt");

        BufferedReader br=new BufferedReader(fr);
```

```
        String line=br.readLine();

        while(line != null){
            System.out.println(line);
            line=br.readLine();
        }
        br.close();
    }
}
```

PrintWriter class:

- It is the child class of Writer class.
- It is the most enhanced writer to write any type of data to the file.
- By using the **FileWriter** and the **BufferedWriter** classes we can write only character data to the file, we can not write the primitive data to the file directly (like boolean, int, float, etc.) but with the help of the **PrintWriter** class we can write any type of primitive data to the file

Note: PrintWriter class can communicate directly to the file and even via some writer object also.

```
PrintWriter pw=new PrintWriter(String fname);

PrintWriter pw=new PrintWriter(File f);

PrintWriter pw=new PrintWriter(Writer w);
```

Methods of the PrintWriter class:

1. All the methods of Writer class(like write(int c), write(String), etc.)
2. print(int i);

   print(double d);

   print(boolean b);

   print(String s);
3. println(int i);

   println(String s);

   etc.

Note: difference between write(100) and print(100) is write(100) method will write corresponding character 'd' whereas print(100) method will print the int value to the file.

Note: The most enhanced reader is the BufferedReader class and the most enhanced Writer is the PrintWriter class.

```
import java.io.*;
public class Main {

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

        //FileWriter fw=new FileWriter("abc.txt",true);
        //PrintWriter out=new PrintWriter(fw);

        //or

        PrintWriter out=new PrintWriter("d://abc.txt");

        out.write(100);//d will be added
        out.println(100);
        out.println(true);
        out.println('c');
        out.println("amit");

        out.flush(); //need not call the flush method
        out.close();
        System.out.println("done..");
    }
}
```

Note: As we can use Reader and Writer class to handle character data, we can use InputStream and OutputStream class to handle binary data like images, audio, video, a class Object state, etc.

Example: copy the image from one location to another location (make sure that img.jpg file is there in the d:// drive)

```
import java.io.*;
public class Main {

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

        FileInputStream fis=new FileInputStream("d://img.jpg");

        FileOutputStream fos=new FileOutputStream("d://img1.jpg");

        int i=fis.read();

        while(i != -1){
            fos.write(i);
            i=fis.read();
        }

        fos.flush();
        fos.close();
        fis.close();
```

```
        System.out.println("success.....");

    }
}
```

If we use FileReader and FileWriter class for the above application instead of FileInputStream and FileOutputStream then the image will not be copied properly because image file is a byte oriented file not a character oriented file.

Serialization:

Serialization is the conversion of the state of an object into a byte stream. which we can then save to a database or transfer over a network.

The reverse operation of serialization is called *deserialization* where byte-stream is converted into an object. The serialization and deserialization process is platform-independent, it means you can serialize an object on one platform and deserialize it on a different platform.

We can't serialize any java class object, There is an interface called **java.io.Serializable,** The class which implements this Serializable interface, that class object is only eligible for serialization.

Note: If we try to serialize a non-serializable object (If the object of a class has not implemented the Serializable interface)then we will get a runtime exception called NotSerializableExcception

Serializable is a marker interface (has no data member and method). It is used to "mark" Java classes so that the objects of these classes may get a certain capability.

Note: all the Wrapper classes, String classes and collection framework related classes internally implements the Serializable interface.

The **java.io.ObjectOutputStream** class helps us to serialize an object of a java class (converts it into a sequence (stream) of bytes)

Similarly, The **java.io.ObjectInputStream** class helps us read a stream of bytes and convert it back into a Java object.(Deserialization)

ObjectOutputStream class has a method by name **writeObject(Serializable s)** ,it takes an object of a class whose class implements Serializable interface. and converts it into a sequence (stream) of bytes.

Note: we can serialize multiple object, but in which order we serialized those objects, in the same order only we have to deserialize those objects.

Example: Serialization

```
import java.io.FileOutputStream;
import java.io.ObjectOutputStream;
import java.io.Serializable;

 class A implements Serializable
{

    int i=10;

    public void funA(){

        System.out.println("inside funA() of A");
        System.out.println(i);
    }

}

class Main {

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

        A a1=new A();

        a1.i=22; //change the state of a1 object

        FileOutputStream fos=new FileOutputStream("file1.txt");
        ObjectOutputStream oos=new ObjectOutputStream(fos);

        oos.writeObject(a1);

        oos.writeObject("Welcome");//String class object
        oos.writeObject(10); //autoboxing

        oos.close();

        System.out.println("a1 object is serailized");
    }
}
```

Example of Java Deserialization:

Deserialization is the process of reconstructing the object from the serialized state. It is the reverse operation of serialization

To deserialize an object we need to use following method **ObjectInputStream** class.

```
public Object readObject();
```

To get back our object we have to downcast the object of Object class into object of Original class.

```
class Main {

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

        FileInputStream fis=new FileInputStream("file1.txt");

        ObjectInputStream ois=new ObjectInputStream(fis);

        Object obj=ois.readObject();

        A a1=(A)obj;

        a1.funA();

        String ss=(String)ois.readObject();
        System.out.println(ss);

        int z=(Integer)ois.readObject();
        System.out.println(z);

        ois.close();


    }
}
```

transient keyword:

This keyword is applicable only for the variables, at the time of serialization if we don't want to save the original value of a particular variable for some security reason, such type of variable we should declare with transient keyword..

For example, if a program accepts a user's login details and password. But we don't want to store the original password in the file. Here, we can use transient keyword and when JVM reads the transient keyword it ignores the original value of the object and instead stores the default value of the object.

We Problem:

Serializing an object with transient keyword:

```
import java.io.*;
public class Student implements Serializable{

 int id;
 String name;
 transient int age;//Now it will not be serialized

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

```
 public static void main(String args[])throws Exception{
  Student s1 =new Student(211,"ravi",22);//creating object
  //writing object into file
  FileOutputStream f=new FileOutputStream("f.txt");
  ObjectOutputStream out=new ObjectOutputStream(f);

  out.writeObject(s1);

  out.flush();
  out.close();

  System.out.println("success");
 }
}
```

You Problem:

Write a application to de-serialize the above Student object and print their values.

transient vs static:

static variables are not part of object, hence they will also not participate in serialization.

Object Graph: (Serialization with respect to Has-A relationship)

Whenever we are serializing an object, the set of all the objects which are reachable from that object will be serialized automatically. this group of object is nothing but object-graph..

In object-graph every object should be serializable otherwise we will get a Runtime exception NotSerializableException.

Example:

```
import java.io.*;
class Dog implements Serializable{

  Cat c=new Cat();

}

class Cat impelements Serializable{

  Rat r=new Rat();

}

class Rat implements Serializable {
```

```
}

class Main{

  public static void main(String[] args){

    Dog d=new Dog();

    FileOutputStream fos=new FileOutputStream("file1.txt");
    ObjectOoutputStream oos=new ObjectOoutputStream(fos);
    oos.writeObect(d);
    oos.flush();
    oos.close();

    System.out.println("done");

}
}
```

In the above program whenever we are serializing Dog obj, automatically Cat and Rat object will be serialized. because these are the part of the object-graph of the dog object.

Among above classes if at-least one class is not Serializable then we will get a Runtime exception.

Serialization with the respect of Inheritance:

If the parent class is implementing Serializable interface then automatically every child class implements Serializable. weather we mention it or not in a child class.

It is a Serializable nature of inheriting parent to child.

Example:

```
import java.io.*;
class Animal implements Serializable{

    int i=10;

}

class Dog extends Animal{

    int j=20;
}


class Main {

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

        FileOutputStream fos=new FileOutputStream("file1.txt");

        ObjectOutputStream oos = new ObjectOutputStream(fos);
```

```
        Dog d = new Dog();

        oos.writeObject(d);;

        System.out.println("done..");


    }
}
```

Here we have serialized the dog object, even though Dog class has not implemented the Serializable interface.

If the parent class doesn't implements the Serializable interface still we can serialize the child objects. if the child class implements Serializable interface.

At the time of serialization JVM will ignores the updated values of instance variables which are inherited from the parent.

At the time of Deserialization JVM will check if the parent class is serializable or not. if the parent class is non-serializable then JVM will creates an object for non-serializable parent class by executing default constructor of the parent class and share its default instance variable value to the child objects.

Example

```
import java.io.*;
class A {

    int i=10;

}

class B extends A implements Serializable{

    int j=20;

}


class Main {

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


        B b1 = new B();

        b1.i = 200;
        b1.j = 500;


        FileOutputStream fos=new FileOutputStream("file1.txt");

        ObjectOutputStream oos = new ObjectOutputStream(fos);
```

```
        oos.writeObject(b1);;

        System.out.println("serialized the Child object ");

        System.out.println("After Deserialization");

        ObjectInputStream ois = new ObjectInputStream(new FileInputStream("file1.txt"));

        B b2 = (B)ois.readObject();

        System.out.println(b2.i);
        System.out.println(b2.j);

    }
}

Output:
serialized the Child object
After Deserialization
10
500
```

java.nio package:

This java.nio package was introduced in java 1.4 version to implement high-speed IO operations. It is an alternative to the standard IO API.

java.nio.file package:

In this package there are some following classes and Interfaces are there by using which we can create a file or folder, we can read or write from a file in much more efficient way.

java.nio.file.Path inteface
java.nio.file.Paths class
java.nio.file.Files class

java.nio.file.Path interface:

The object of this Path represents actual location of a file or folder.

In computer every file and folder in a file system can be uniquely identified by the Path object.

A Path can be an absolute or relative, an absolute path means the location/address from the root to the file or folder. where as a relative path is the location/address which is relative to some other path.

We get the Path object by the help of java.nio.file.Paths class static method called get().

Example:

```
Path p = Paths.get("d://abc"); // for folder

Path p = Paths.get("d://abc//a1.txt"); // for files
```

Note:- Getting the object of Path doesn't mean that creating a new File or folder.

After getting the object of Path we need to supply this Path object to the some of the static methods of **java.nio.file.Files** class to perform manipulation on files and folders.

Some of the static method presents in java.nio.file.Files class:

createFile(); // used to create a file

createDirectory(); // used to create a folder

List<String> readAllLines(); // here we can read in the form of List of String , here no need to take BufferedReader, it is very fast.

byte[] readAllBytes(); // here we can read in the form of byte array.

Stream<String> lines(); // read all the line from a file and return the java.util.stream.Stream object

delete(); // used to delete a file or folder

deleteIfExist(); checks before deleting a file or folder

write(Path path, byte[] bytes); // Writes bytes to a file specified by the Path object.

copy(); // to copy a file //copy and paste

move() // to move a file // cut and paste

etc.

Example: creating a new File:

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;

class Main {

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

        Path p = Paths.get("d://abc//a1.txt"); // here d:/abc folder must be there,otherwise we get an exception

       if(Files.exists(p)) {
            System.out.println("File is aready exist");
```

```
        }else {
            Path p2 = Files.createFile(p);;
            System.out.println("created a file at :" + p2);

        }
    }
}
```

Example: writing some String to the File:

```
import java.io.IOException;
import java.nio.file.*;
import java.util.*;
class Main {

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

        Path p = Paths.get("d://a1.txt");

        String msg="welcome to java";

        //writing a normal string
        Files.write(p, msg.getBytes());

        List<String> list= Arrays.asList("delhi","mumbai","kolkata","chennai");

        //writing a List of String
        //Files.write(p, list);
        //In append mode
        Files.write(p, list,StandardOpenOption.APPEND);

        System.out.println("done...");


    }
}
```

Example: Reading a file line by line

```
import java.io.IOException;
import java.nio.file.*;
import java.util.*;
class Main {

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

        Path p = Paths.get("d://a1.txt");

        List<String> list= Files.readAllLines(p);

        System.out.println("Reading from the file");
        for(String line:list) {
            System.out.println(line);
        }

    }
}
```

Example: reading a file using Stream:

```
import java.io.IOException;
import java.nio.file.*;
import java.util.stream.Stream;

class Main {

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

        Path p = Paths.get("d://a1.txt");

        Stream<String> str= Files.lines(p);

        str.forEach(line -> System.out.println(line));

    }
}
```

Example: applying map to the file: (if some specific text is there then convert it in different text)

If any Line Admin is there, then convert it as “Welcome Admin”;

```
import java.io.IOException;
import java.nio.file.*;
import java.util.stream.Stream;

class Main {

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

        Path p = Paths.get("d://a1.txt");

        Stream<String> str= Files.lines(p);

        str.map(line -> {

            if(line.contains("Admin"))
                return line.replace("Admin","Welcome Admin");
            else
                return line;

        }).forEach( line -> System.out.println(line));
    }
}
```

Example: Reading from one file and writing to another file:

```
import java.io.IOException;
import java.nio.file.*;
import java.util.List;

class Main {

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

        Path sourcePath = Paths.get("d://a1.txt");
```

```
        Path dPath = Paths.get("ab.txt");

        Files.createFile(dPath);

        List<String> list = Files.readAllLines(sourcePath);

        Files.write(dPath, list);

        System.out.println("done");


    }
}
```

Example: copying a file from one location to another

```
  Path source=Paths.get("somefile.zip");
  Path dest  =Paths.get("ab2.zip");


  Files.copy(source,dest);

  System.out.println("done");
```

Note : If we use move() method in place of copy() method then from the source the files will be removed(like cut and paste option).