UNIT - II

Packages:

A package in Java is used to group related classes. Think of it as a folder in a file directory. We use packages to avoid name conflicts, and to write a better maintainable code.

Creating Packages and importing:

The directory structure on your computer is related to the package name.

Suppose we have these two classes:

E:\vdemo\bindu\Bank.java

```
package  BankProject;
public class  Bank
{
   private float  balance;
   public Bank(float balance)
   {
      this.balance=balance;
   }
   public void displayBalance()
   {
     System.out.println("Balance="+ balance);
   }
}
```

Compile:

```
E:\vdemo\bindu> javac   –d   .   Bank.java
```

Add the following path to the **"classpath"** envornment variable

E:\vdemo\bindu,

which consists the package BankProject. So that the class **Bank** from the package BankProject can be accessed from anywhere.

C:\sudheer\Test.java

```
import  BankProject.Bank;

class Test

{

 public static void main(String[] args)

   {

       Bank b1=new Bank(2000);

       b1.displayBalance();

   }

}
```

Compile:

C:\sudheer> javac   Test.java

To Run

java   Test

output:

Got it

CLASSPATH:

CLASSPATH is an environment variable which is used by the Application ClassLoader to locate and load the .class files.

The CLASSPATH defines the path, to find third-party and user-defined classes that are not extensions or part of Java platform.

Include all the directories which contain .class files and JAR files when setting the CLASSPATH.

You need to set the CLASSPATH if:

You need to load a class that is not present in the current directory or any sub-directories.

You need to load a class that is not in a location specified by the extensions mechanism.

There are two ways to ways to set CLASSPATH:

1. through Command Prompt:
   Example:
   Type the following command in the command prompt.
   set classpath= %classpath%;e:\vdemo\bindu;
   in the above line we added myjava folder to the class path, so the class loader can recognize the classes that are imported from 'myjava' folder(package).
2. setting Environment Variable.
   System->Advanced System Settings->Advanced->Environment variables
   Variable name: classpath
   Value: e:\vdemo\bindu

Interfaces

An interface in Java is similar to class but it has static constants and abstract methods.

There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple inheritance in Java.

Java Interface also represents the IS-A relationship.

It cannot be instantiated just like the abstract class.

Since Java 8, we can have default and static methods in an interface.

Since Java 9, we can have private methods in an interface.

Why use Java interface?

There are mainly three reasons to use interface. They are given below.

1. It is used to achieve abstraction.

2. By interface, we can support the functionality of multiple inheritance.

3. It can be used to achieve loose coupling.

Defining interface/ implementing interface/ Applying interface:

```
// interface - Runtime polymorphism/ full polymorphism/ Dynamic method dispatch mechanism

interface Bank

{

      int  getRateOfInterest();

}

class  SBI  implements  Bank

{

    public int  getRateOfInterest()
```

```
    {
      return 8;
    }
}
class  ICICI  implements  Bank
{
    public int  getRateOfInterest()
     {
       return  7;
     }
}
class  Test
{
public  static  void  main(String  args[])
     {
       Bank  ref1;
       ref1=new  SBI();
       System.out.println(ref1.getRateOfInterest());
       ref1=new  ICICI();
       System.out.println(ref1.getRateOfInterest());
     }
}
```

Multiple Inheritance using interfaces:

```
interface  Printable
```

```
{
  void   print();
}
interface  Showable
{
 void show();
}
class   A7   implements  Printable,Showable
{
    public  void   print()
      {
           System.out.println("Hello");
      }
     public void show()
      {
           System.out.println("Welcome");
       }
     public static void main(String args[])
      {
            A7 obj = new A7();
            obj.print();
            obj.show();
       }
}
```

Nested interfaces:

An interface i.e. declared within another interface or class is known as nested interface.

The nested interfaces are used to group related interfaces so that they can be easy to maintain.

The nested interface must be referred by the outer interface or class.

It can't be accessed directly.

Points to remember for nested interfaces

Nested interface must be public if it is declared inside the interface but it can have any access modifier if declared within the class.

Nested interfaces are declared static implicitly.

Example:

```
interface  Showable
{
   void show();
   interface Message
   {
      void msg();
    }
}
class    TestNestedInterface1   implements    Showable.Message
{
     public void msg()
     {
```

```
        System.out.println("Hello nested interface");
    }
    public static void main(String args[])
    {
        Showable.Message   message=new   TestNestedInterface1();
        message.msg();
    }
}
```

Variables in interface:

The variable in an interface is public, static, and final by default.

If any variable in an interface is defined without public, static, and final keywords then, the compiler automatically adds the same.

No access modifier is allowed except the public for interface variables.

Every variable of an interface must be initialized in the interface itself.

The class that implements an interface cannot modify the interface variable, but it may use as it defined in the interface.

Stream based I/O (java.io)

Java uses the concept of a stream to make I/O operation fast.

The java.io package contains all the classes required for input and output operations.

Stream:

A stream is a sequence of data. In Java, a stream is composed of bytes. It's called a stream because it is like a stream of water that continues to flow.

In Java, 3 streams are created for us automatically. All these streams are attached with the console.

1) System. Out: standard output stream

2) System.in: standard input stream

3) System.err: standard error stream

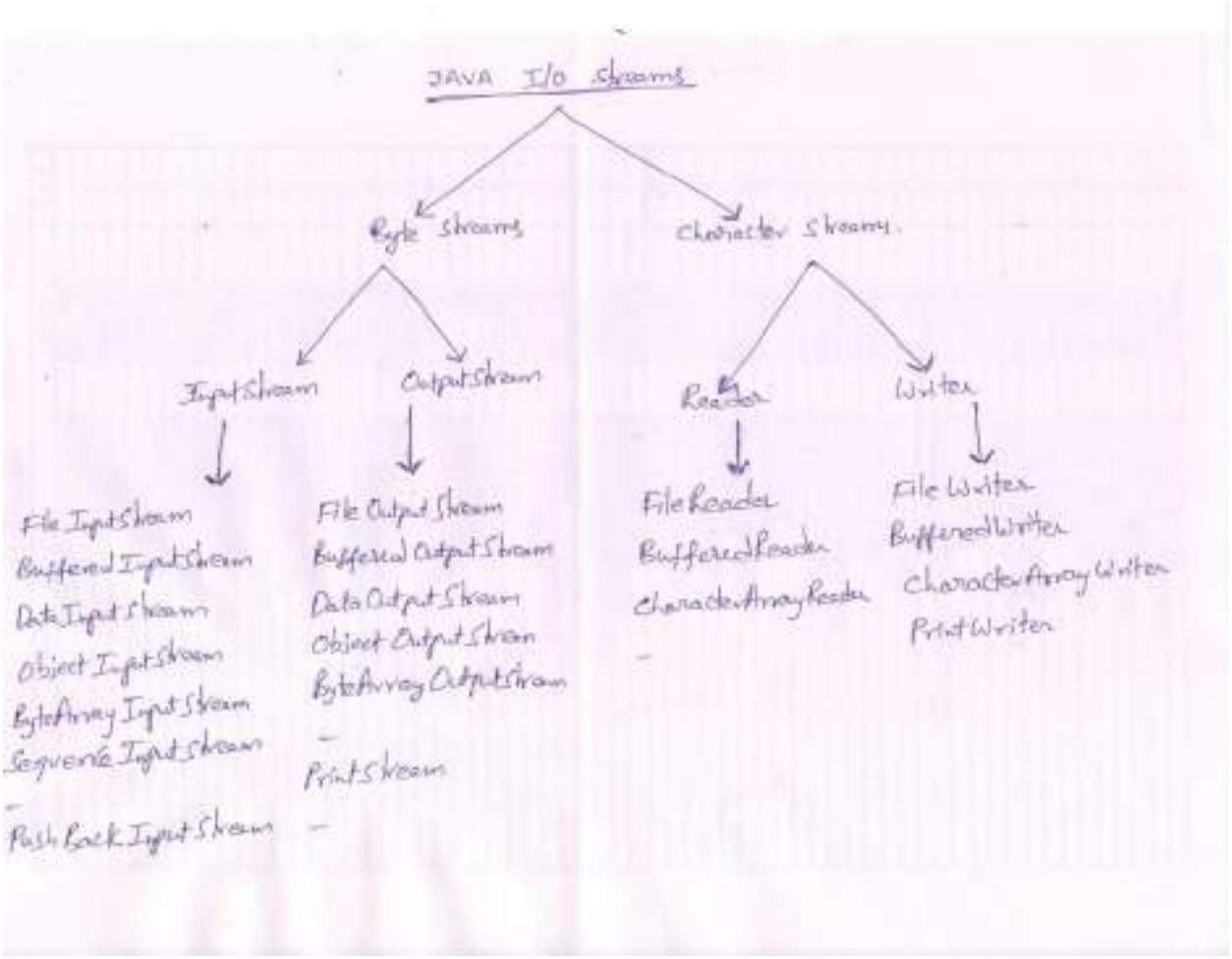
JAVA I/0 Classes:

1. Byte Streams:

- Read/Write the data as bytes.
- These classes are used to read/write primitive data types, objects.
- InputStream & OutputStream are the base classes for these classes.

2. Character Streams:

- Read/Write data as characters.
- These classes are used to read/write text data.
- Reader/Writer are the base classes for these classes.



Reading and writing Files:

Byte Streams:

FileInputStream class

- Helps us to read data(bytes) from the file.
- This class has different read() methods that helps to read data from the file.
- read() - reads one byte of data from the input stream
- read(byte[] array) - reads n bytes(n is the size of the array passed as a parameter to read method) from the stream and stores in the specified array.

//demonstrates how to read a byte at a time from the file and display on the monitor.

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

```
class ex1
{
  public static void main(String[] args) throws
IOException,FileNotFoundException
  {
   int  ch;
   FileInputStream   fis = new   FileInputStream("praveen.txt");
  while(true)
  {
     ch=fis.read();
     if(ch==-1)
         break;
    System.out.print((char)ch+" ");
  }
  fis.close();
}
}
```

The read() method used in the above program reads one byte of data from the current file pointer location, returns the equivalent int value and also advances the file pointer to the next byte.

If there is no more bytes to read from the file, the read() method returns -1.

FileOutputStream class

- Helps us to write data(bytes) to the file.
- This class has different write() methods that helps to write data to the file.
- write() - writes the specified byte to the output stream
- write(byte[] array) - writes the n bytes(n is the size of the array passes as a parameter to write method) from the specified array to the output stream.

//Retrieve characters from a string variable and write them a byte at a time onto a file.

```
import java.util.*;
import java.io.*;
class  ex2
{
 public static void main(String[] args) throws Exception
 {
   int ch, ch2;
   FileOutputStream   fos = new   FileOutputStream("Alekya.txt");
   String   s = "Welcome Alekya to IO Streams";
   byte   b[] = s.getBytes();
   for(int i=0;i<b.length;i++)
       fos.write(b[i]);
   fos.close();
   System.out.println("data written to the File.");
 }
}
```

BufferedInputStream class

- Used for Performace optimzation
- The read() method of FileInputStream class has to contact secondary storage everytime to fetch the data, which is very slow.
- We can convert a FileInputStream reference to BufferedInputStream reference, so that in creates an internal buffer in the main memory and brings blocks of data from secondary storage to main memory in advance.
- So the read() method of BufferedInputStream has to contact internal buffer in the main memory instead of secondary storage. And hence improves performance.

Example code:

```
import java.io.*;
class ex3
{
  public static void main(String[] args) throws
IOException,FileNotFoundException
  {
   int ch;
   FileInputStream  fis= new  FileInputStream("bindu.txt");
   BufferedInputStream   bis = new  BufferedInputStream(fis);
  while(true)
  {
     ch=bis.read();
     if(ch==-1)
         break;
    System.out.print((char)ch+" ");
  }
  bis.close();
}
}
```

DataOutputStream class

- This class has different type of Write() methods to directly write primitive data types to any output steam(example-File).
- Example - writeInt(), WriteFloat(), WriteDouble(),….etc. for all primitive data types.
- So we can treat the data to be written as a corresponding primitive type instead of counting number of bytes.

//To demonstrate how to read integer values from keyboard and write them onto a file.

```
import java.io.*;
import java.util.*;
class  ex4
{
 public static void main(String[] args) throws IOException
 {
    int  i, x;
    Scanner  s = new  Scanner(System.in);
    FileOutputStream  fos = new  FileOutputStream("preethi.dat");
    DataOutputStream  dos = new  DataOutputStream(fos);
    //reading 5 integers from keyboard & writing them onto a file.
   for(i=1;i<=5;i++)
    {
      System.out.println("enter the number:");
      x=s.nextInt();  // x=14
      dos.writeInt(x);
    }
    dos.close();
}
}
```

DataInputStream class

- This class has different type of read() methods to directly read primitive data types from any input stream.

- Example - readInt(), readFloat(), readDouble(),….etc. for all primitive data types.

//To demonstrate how to retrieve all integer values present in the file and display on the monitor.

```
import java.io.*;
import java.util.*;
class  ex5
{
 public static void main(String[] args) throws IOException
 {
   int  x;
   FileInputStream   fis = new  FileInputStream("preethi.dat");
   DataInputStream  dis = new  DataInputStream(fis);
   for(int i=0;i<5;i++)
  {
      x=dis.readInt();
      System.out.println(x);
   }
  dis.close();
}
}
```

Serialization

- Process of writing the state of an object to a byte stream.
- Saving the object in a persistent(permanent) storage.
- The objects of the any class can be serialized only if either the class must implement Serializable interface or any one of its super class must implement Serializable interface.

- The serializable interface has no methods, it only indicates that the objects of this class can be serialized.
- If a class is Serializable, all of its sub classes are also serializable.
- Transient attributes will not be serialized.
- Static attributes also will not be serialized.

To demonstrate how to write object onto a File.

```
import java.io.*;
import java.util.*;
public class Student implements Serializable
{
    public String rno;
    public String name;
    public Student(String rno,String name)
    {
       this.rno=rno;
       this.name=name;
    }
    public void display()
    {
        System.out.println("ROLL NO="+rno+" NAME="+name);
     }
}
class  FileWrite
{
 public static void main(String args[]) throws IOException,ClassNotFoundException
```

```
{
    FileOutputStream fos=new FileOutputStream("student.dat");
    ObjectOutputStream oos=new ObjectOutputStream(fos);
    Student s1=new Student("19891A1201","shiva");
    oos.writeObject(s1);
    Student s2=new Student("19891A1202","Younus");
    oos.writeObject(s2);
    Student s3=new Student("19891A1203","kiran");
    oos.writeObject(s3);
    oos.close();
 }
}
```

To demonstrate how to retrieve and display all the objects present in the file.

```
import java.io.*;
import java.util.*;
class FileRead
{
 public static void main(String args[]) throws IOException,ClassNotFoundException
 {
    FileInputStream fis=new FileInputStream("student.dat");
    ObjectInputStream ois=new ObjectInputStream(fis);
    Student s=null;
    for(int i=0;i<3;i++)
     {
```

```
        s=(Student)ois.readObject();
         s.display();
      }
   }
}
```

Character Streams

1. Character streams were built on top of byte stream to facilitate handling of character(text) data.

2. Reader and Writer are the abstract base classes for reading(input) and writing(output) related classes in character streams correspondingly.

Reader	Writer
FileReader	FileWriter
BufferedReader	BufferedWriter
	PrintWriter
CharacterArrayReader	CharacterArrayWriter

FileReader:

This class helps us to read data from a file as characters(text).

There are many overloaded read() methods that helps us to read a single character or specified number of characters at a time from a file.

BufferedReader:

This class is used for performance optimization.

This class has readLine() method, which helps us to read a line at a time from the input buffer corresponds to a file.

FileReader and Buffered Reader Example:

Reading from a file

```
//To demonstrate BufferedReader to read a line at a time.
import java.io.*;
class FileRead9
{
public static void main(String args[]) throws IOException
{
  FileReader   fr=new  FileReader("ex1.txt");
  BufferedReader br=new BufferedReader(fr);
  String s;

  System.out.println("The File Content is:");
  while((s=br.readLine())!=null)
     System.out.println(s);
  br.close();
}
}
```

FileWriter:

This class helps us to write data on the file as characters(text).

This class has many overloaded write() methods to write data on the file which will helps us to write one character at a time on the file, specified number of characters at a time onto a file…etc.

Writing on to a file:

```
// To demonstrate FileWriter,Read characters from keyboard and write them
onto a file.
```

```
import java.util.*;
import java.io.*;
class   FileWrite5
{
 public static void main(String[] args) throws IOException
 {
      char ch;
      Scanner sc=new Scanner(System.in);
       FileWriter   fw=new   FileWriter("ex1.txt");
     System.out.println("enter a Line of characters");
     String s=sc.nextLine();
     for(int i=0;i<s.length();i++)
             fw.write(s.charAt(i));
     fw.close();
  }
}
```

In the above program, the write method writes one character at a time onto the file.

File class:

- Abstract representation of File & Directory pathnames.
- Not used to actually read or write data.
- Work at higher level for performing the following: Making new empty files, Searching for files, Deleting files and Making directories.
- Helps us to know meta data information about a file.

Constructor:

File(String pathname)

Operations performed on a File

isFile() – To check whether the entry is a file or not.

length() – to get the size of the file

exists() – to know whether the the file with given name is present or not

lastModified()- to know when the file is last modified

getName() – to get the name of the file from the file reference.

createNewFile()- to create a new file using File reference if the file does not exist

getparent() – to get name of the parent folder

renameTo()- to rename the file

GetAbsolutePath() – to get the complete absolute path right from the drive name.

delete()- to delete a specified file

canRead() – to check whether the file is readable or not.

canWrite()- to check whether the file can be written or not.

Operations performed on a Directory File

isDirectory()-- to check whether the entry in the directory is a sub directoy(directory) or not(a file).

list()- to get name of all files in the directory in the form of string array.

listFiles()-to get name of all files in the directory in the form array of type File class.

listFiles(Filename Filter obj) – specifying filter so that we can list only the files with mathing filter(example, suppose I want to list files with extension ".java" onlye)

mkdir()—to create a new directory.

createNewFile()- to create a new file in the directory.

renameTo()- to rename a directory.

delete()- to delete a directory.

RandomAccessFile class

- This class is not derived from either InputStream or OutputStream.
- With this class we can open the file both for reading and writing.
- This class has Implemented DataInput & DataOutput interfaces.
- This class has implemented all the standard input & output methods which we can use for read/write operations with Random access files
- We can position the file pointer at a required position for read/write operations using seek() method.

Constructor:

RandomAccessFile(File fileobj,String access) throws FileNotFoundException;

Access modes:

r - for read

w - for write

rw - for read and write

rws - Any change to the file data or metadata will be immediately written to the

physical device.

rwd - Any change to the file data will be immediately written to the physical device.

Positioning File Pointer

void seek(long newpos) throws IOException;

It moves the file pointer by number of bytes specified by the argument 'newpos' from the beginning of the file.

The Console class:

- Console is a physical device with a keyboard and a display.
- Makes it easy to accept input from command line, both echoed and non echoed (password).
- Makes it easy to write formatted output to monitor.
- It has method like printf() and format() to write formatted output.

Method in Console class

String readLine()

String readLine(String fmt, Object args);

char[] readPassword()

char[] readPassword(String fmt, Object args);

Console printf(String fmt,Object args);

Console format(String fmt,Object args);

Reading a String

```
String name;
Console c = System.console();
name = c.readLine("Enter %s name", "employee");
c.format("Hello %s", name);
```

Reading Password

```
Console c = System.console();
char pw[] = c.readPassword("Enter password: ");
for(i=0;i<pw.length;i++)
System.out.println(pw[i]);
```

Writing formatted ouput(data) on monitor.

```
import java.io.*;
class ConsoleDemo
{
public static void main(String args[]) throws IOException
{
Console console = System.console();
String   formula = "Formula = ";
double   radius = 2.0;
console.format("%10s%12.10f * %3$4.2f * %3$4.2f%n",
formula, Math.PI, radius);
double area = Math.PI * radius * radius;
console.format("%10s%16.13f%n", "Result = ", area);
}
}
```

Autoboxing:

It is the process by which a primitive type is automatically encapsulated (boxed) into its equivalent wrapper whenever an object of that type is needed.

Auto Unboxing: It is the process by which the value of a boxed object is automatically extracted (unboxed) from a type wrapper when its value is needed.

Autoboxing and Auto Unboxing in Assignment statements

Ex1 Autoboxing

```
Integer iOb=100;
```

Ex2 Auto-Unboxing

```
int i=iOb;
```

Autoboxing With method parameters & Return values

```
class Ex1
{
 static int meth1(Integer v)
 { return v; }
 public static void main(String args[])
 {
 Integer Iob = meth1(100);
 System.out.println(iOb);
 }
}
```

Enumerations

- Is a list of named constants
- In C language, they are simply list of named integer constants.
- In Java, Enumeration defines a class type.
- Enumeration can have constructors, methods and instance variables.

Example

```
enum Apple
{
 Jonathan,GoldenDel,RedDel,Cortland
}
```

- Jonathan, GoldenDel, RedDel, Cortland are enumeration constants.Their type is Apple type.
- Implicitly declared as public, final, static members.
- We can not change the access specifier of these constants.
- Need not instantiate(can't instantiate), can treat like primitive type.

```
Apple ap;

ap = Apple.ReDel;

System.out.println(ap);
```

Methods

1)values()

Returns an array that contains a list of enumeration constants.

Example :

```
Apple a[] = Apple.values()

for(Apple  a1 :a)

 System.out.println(a1);
```

2) valueOf(String str)

Returns an enum constant whose value corresponds to the string passed.

Example :

```
Apple ap;

ap=Apple.valueOf(“RedDel”);

System.out.println(ap);
```

3)ordinal()

You can obtain a value (ordinal value) that indicates an enumeration constant position in the list of constants.

Ordinal value begins with zero.

Note:

- Each Enumeration constant is an object of its enumeration type.
- The constructor will be called when each enumeration constant is created.
- Each enum constant has its own copy of instance variable.
- Default value '0'.
- Constructors, instance var, .. etc all can be specified only after the declaration enumeration constants.

Example Program

```
enum Apple
{
 Jonathan(10), GoldenDel(9), RedDel(12);
 private int price;
 Apple() { price=-1; }
 Apple(int p) { price =p; } int getPrice() { return price; }
}
class Demo
{
public static void main(String args[])
{
Apple ap;
 System.out.println(Apple.GoldenDel.getPrice() );
```

```
 for(Apple a : Apple.values())
 System.out.println(a.getPrice());
 }
}
```

Generics:

The Java Generics programming is introduced in J2SE 5 to deal with type-safe objects.

It makes the code stable by detecting the bugs at compile time.

Before generics, we can store any type of objects in the collection, i.e., non-generic. Now generics force the java programmer to store a specific type of objects.

Advantage of Java Generics

There are mainly 3 advantages of generics. They are as follows:

1) Type-safety: We can hold only a single type of objects in generics. It doesn?t allow to store other objects.

Without Generics, we can store any type of objects.

```
List list = new ArrayList();
list.add(10);
list.add("10");
```

With Generics, it is required to specify the type of object we need to store.

```
List<Integer> list = new ArrayList<Integer>();
list.add(10);
list.add("10");// compile-time error
```

2) Type casting is not required: There is no need to typecast the object.

Before Generics, we need to type cast.

```
List list = new ArrayList();

list.add("hello");

String s = (String) list.get(0);//typecasting
```

After Generics, we don't need to typecast the object.

```
List<String> list = new ArrayList<String>();

list.add("hello");

String s = list.get(0);
```

3) Compile-Time Checking: It is checked at compile time so problem will not occur at runtime. The good programming strategy says it is far better to handle the problem at compile time than runtime.

```
List<String> list = new ArrayList<String>();

list.add("hello");

list.add(32);//Compile Time Error
```

Syntax to use generic collection

```
ClassOrInterface<Type>
```

Example to use Generics in java

```
ArrayList<String>
```