UNIT - II

PACKAGES: A **java package** is a group of similar types of classes, interfaces and sub-packages.

Advantage of Java Package

1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.

2) Java package provides access protection.

3) Java package removes naming collision.

In java, the packages have divided into two types.

- Built-in Packages
- User-defined Packages

1: Built-in packages: These are the packages which are already available in java language. These Packages provide all most all necessary classes, interfaces and methods for the programmer to Perform any task in his programs.

Some of the important packages are.

1 : java.lang : lang stands for language. This package got primary classes and interfaces for Developing a basic java program. It consists of wrapper classes which are useful to convert Primitive data types into objects. These are classes like String, StringBuffer, threads and Exceptions.

2 : java.util : util stands for utility. This package contains useful classes and interfaces like Stack, LinkedList, Hashtable, Vector,Arrays,Random, etc.

3 : java.io : io stands for input and output. This package contains streams. Astram represents flow of data from one place to another place. This package contains input-output related tasks.

4 : java.awt : awt stands for abstract window toolkit. This package helps to develop GUI (Graphical User Interface).

5 : java.swing : These package to develop GUI like java.awt. The x in javax represents it is an extend package.

6 : java.applet : applets are programs which come from a server into a client and get extended on a client machine on a network.

7 : java.text : These package has two important classes.

1 : DateFormate : to perform dates and times.
2: Number Format: Which is useful to format numeric values?

2. User-defined Packages: The package which is defined by the user is called a User-defined package. It contains user-defined classes and interfaces.

Rules to create user defined package

- package statement should be the first statement of any package program.
- Choose an appropriate class name or interface name and whose modifier must be public.
- Any package program can contain only one public class or only one public interface but it can contain any number of normal classes.
- Package program should not contain any main class (that means it should not contain any main())
- modifier of constructor of the class which is present in the package must be public. (This is not applicable in case of interface because interface have no constructor.)
- The modifier of method of class or interface which is present in the package must be public (This rule is optional in case of interface because interface methods by default public)
- Every package program should be save either with public class name or public Interface name

Defining a Package in java

We use the *package* keyword to create or define a package in java programming language.

Syntax: package packageName;

Example: package myPackage;

- The package statement must be the first statement in the program.
- The package name must be a single word.

Compile package programs:

For compilation of package program first we save program with public className.java and it Compile using below syntax:

Syntax : javac -d . className.java
javac –d . Add.java

Explanation: In above syntax " –d "is a specific tool which is tell to java compiler create a separate folder for the given package in given path. When we give specific path then it create a new folder at that location and when we use . (dot) then it crate a folder at current working directory.

Note: Any package program can be compile but can not be execute or run. These program can be executed through user defined program which are importing package program.

Example of package program

Package program which is save with Add.java and compile by javac -d . Add.java

```
package myPackage; public
class Add
{
        int x,y,z;
        public void display()
        {
x=30;           y=40;
        z=x+y;
                System.out.println("Sum is:"+z);
        }
}
```

IMPORTING PACKAGE:

- When a package has imported, we can refer to all the classes of that package using their name directly.
- A program may contain any number of import statements.
- Using an importing statement, we can import a specific class

  Syntax : import packageName.ClassName;
- To import all the classes of the package, we use * symbol.

  Syntax : import packageName.*;

Sample.java

```
import myPackage.Add; public
class Sample
{
        public static void main(String args[])
        {
                Add a=new Add();
a.display();
        }
}
```

Compile : javac Sample.java

Run : Java Sample

CLASSPATH

CLASSPATH can be set by any of the following ways:

- CLASSPATH can be set permanently in the environment:
- In Windows, choose control panel
- System
- Advanced
- Environment Variables
- choose "System Variables" (for all the users) or "User Variables" (only the currently login user)
- choose "Edit" (if CLASSPATH already exists) or "New"
- Enter "CLASSPATH" as the variable name
- Enter the required directories and JAR files (separated by semicolons) as the value (e.g., ".;c:\javaproject\classes;d:\tomcat\lib\servlet-api.jar").
- Take note that you need to include the current working directory (denoted by '.') in the CLASSPATH.

To check the current setting of the CLASSPATH, issue the following command:

- > SET CLASSPATH
- CLASSPATH can be set temporarily for that particular CMD shell session by issuing the following command:
- > SET CLASSPATH=.;c:\javaproject\classes;d:\tomcat\lib\servlet-api.jar
- Instead of using the CLASSPATH environment variable, you can also use the command-line option -classpath or -cp of the javac and java commands, for example,

> java –classpath c:\javaproject\classes com.abc.project1.subproject2.MyClass3

ACCESS PROTECTION

In Java, Access modifiers help to restrict the scope of a class, constructor, variable, method, or data member. It provides security, accessibility, etc to the user depending upon the access modifier used with the element.

Java has four access modifiers

1. private
2. default
3. protected
4. public

1. Private: The **private** members can be accessed only inside the same class. It cannot be accessed from outside the class. The scope of the private access specifier is class level.

Example:

```
class A
{
        private int data=40;
        private void msg()
        {
                System.out.println("Hello java");
        }
}
class Simple
{
        public static void main(String args[])
        {
                A obj=new A();
                System.out.println(obj.data); //Compile Time Error   obj.msg();
                //Compile Time Error
        }
}
```

2. Default: If you don't use any modifier, it is treated as **default** by default. The default modifier is accessible only within package. It cannot be accessed from outside the package. It provides more accessibility than private. But, it is more restrictive than protected, and public. **Example:** **Save by A.java**

```
package pack;
class A
{
        void msg()
        {
                System.out.println("Hello");
```

Compile : javac –d . A.java

```
        }
}
```

Save by B.java

```
package mypack;
import pack.*;   class
B
{
        public static void main(String args[])
        {
                A obj = new A();//Compile Time Error   obj.msg();//Compile
                Time Error
        }
}
```

Compile : javac B.java

Run : java B

3. Protected: protected members of a class are accessible outside the class. But generally within the same package and another package in a sub class. **Example:** **Save by A.java**

```
package pack;
public class A
{
        protected void msg()
        {
                System.out.println("Hello");
        }
}
```

Compile : javac –d . A.java

Save by B.java

```
package mypack;
import pack.*;   class
B extends A
{
        public static void main(String args[])
        {
                B obj = new B();   obj.msg();
        }
}
```

Compile : javac B.java

Run : java B

4. Public: public members of a class are accessible everywhere out-sides the class. The scope of the public access specifier is global.

Example :

```
package pack;
public class A
{
        public void msg()
        {
                System.out.println("Hello");
        }
}
```

Compile : javac –d . A.java

Save by B.java

```
package mypack;
import pack.*;   class
B
{
        public static void main(String args[])
        {
                A obj = new A(); obj.msg();
        }
}
```

Compile : javac B.java

Run : java B

Let's understand the access modifiers in Java by a simple table.

Accessibility / Location / Access Specifier	Same Class	Same Package		Other Package	
		Child class	Non-child class	Child class	Non-child class
Public	Yes	Yes	Yes	Yes	Yes
Protected	Yes	Yes	Yes	Yes	No
Default	Yes	Yes	Yes	No	No
Private	Yes	No	No	No	No

INTERFACES

- An interface should be declared by using the keyword interface, interface is like a class.
- An interface contains only abstract methods, but not provides the body.
- All the methods in an interface must be public and abstract.
- Public and abstract keywords may be omitted in the code and java treats all the methods of the Interface as the public and abstract implicitly.
- When an interface method is implemented in a class, it must be declared as public.
- Whenever a class is implementing an interface this class should override and must provide body of each and every method in the interface.
- We cannot create an object to interface, but we can create a reference.

Why do we use an Interface?

- It is used to achieve total abstraction.
- Since java does not support multiple inheritances in the case of class, by using an interface it can achieve multiple inheritances.
- Any class can extend only 1 class but can any class implement infinite number of interface.
- Interfaces are used to implement abstraction. So the question arises why use interfaces when we have abstract classes?
- The reason is, abstract classes may contain non-final variables, whereas variables in the interface are final, public and static.

DEFINING AN INTERFACE

Defining an interface is similar to that of a class. We use the keyword interface to define an interface. All the members of an interface are public by default.

Syntax:

```
interface interfacename
{
        datatype variablename=value;
        returntype methodname(list of parameters or no parameters);
}
```

Where,

- **Interface** is a keyword interface name can be user defined name.
- The default signature of variable is public static final and for method is public abstract.

- JVM will be added implicitly public static final before data members and public abstract before method.

Example:

```
interface  A
{
        intsrno=101;
        void Adisplay();
        void Ashow();

}
```

IMPLEMENTING AN INTERFACE

A class can implement any number of interfaces. When a class wants to implement more than one interface, we use the **implements** keyword is followed by a comma-separated list of the interfaces implemented by the class.

Syntax:

```
class className implements InterfaceName
        {
                boby-of-the-class
    }
```

Example :

```
interface A
{
        void Adisplay();
void Ashow();
}
class Demo implements A
{
        public void Adisplay()
        {
                System.out.println(" A display ");
        }
        public void Ashow()
        {
                System.out.println(" A show");
        }
}
class Interface3
{
        public static void main(String arg[])
        {
                Demo obj=new Demo();
obj.Adisplay();
```

OUTPUT :

A display
A show

```
        obj.Ashow();
    }
}
```

IMPLEMENTING MULTIPLE INTERFACES

When a class wants to implement more than one interface, we use the ***implements*** keyword is followed by a comma-separated list of the interfaces implemented by the class.

Syntax:

```
class className implements InterfaceName1, InterfaceName2, ...
{   boby-of-the-class
}
```

Example :

```
interface A
{
    void Adisplay();
void Ashow();
}
interface B
{
    void Bdisplay();
void Bshow();
}
class Demo implements A,B
{
    public void Adisplay()
    {
        System.out.println(" A display ");
    }
    public void Ashow()
    {
        System.out.println(" A show");
    }
    public void Bdisplay()
    {
        System.out.println(" B display ");
    }
    public void Bshow()
    {
        System.out.println(" B show");
```

```
        }
}
class Interface2
{

        public static void main(String arg[])
        {
                Demo obj=new Demo();
        obj.Adisplay();
obj.Ashow();            obj.Bdisplay();
        obj.Bshow();


                 A ob=new Demo();
                ob.Adisplay();
ob.AShow();


               B ob1=new Demo();
                ob1.Bdisplay();
ob1.BShow();


        }
}
```

OUTPUT :

```
A display A show
B display
B show

A display
A show

B display
B show
```

EXTENDING AN INTERFACE

- In java, an interface can extend another interface.
- When an interface wants to extend another interface, it uses the keyword ***extends***.
- The interface that extends another interface has its own members and all the members defined in its parent interface too.
- The class which implements a child interface needs to provide code for the methods defined in both child and parent interfaces, otherwise, it needs to be defined as abstract class.

Syntax: interface ChildInterfaceName extends ParentInterfaceName

```
{
        boby-of-the-class
}
```

Example : interface

```
A
{
    void Adisplay();
void Ashow();
}
interface B extends A
{
    void Bdisplay();
void Bshow();
}
class Demo implements B
{
    public void Adisplay()
    {
        System.out.println(" A display ");
    }
    public void Ashow()
    {
        System.out.println(" A show”);
    }
    public void Bdisplay()
    {
        System.out.println(" B display ");
    }
    public void Bshow()
    {
        System.out.println(" B show”);
```

```
        }
}
class Interface4
{

        public static void main(String arg[])
        {
                Demo obj=new Demo();
        obj.Adisplay();
obj.Ashow();            obj.Bdisplay();
        obj.Bshow();

                B ob=new Demo();
                ob.Adisplay();
ob.Ashow();             ob.Bdisplay();
        ob.Bshow();

                A ob1=new Demo();
                ob1.Adisplay();
ob1.AShow();
        }
}
```

OUTPUT :

```
A display A show
B display
B show

A display A show
B display
B show

A display
A show
```

VARIABLES IN JAVA INTERFACES

- An interface is a container of abstract methods and static final variables.
- The interface contains the static final variables.

  Example: public static final int PI=3.14;
- The variables defined in an interface cannot be modified by the class that implements the interface, but it may use as it defined in the interface.

Example:

```
interface Marks
{
        int internal=25;
        int external=75;
        int total=100;
}
class Demo implements Marks
{
        void display()
        {
                System.out.println("Internal Marks:"+internal);
                System.out.println("External Marks:"+external);
                System.out.println("Total Marks:"+total);
        }
}
class InterfaceVariable
{
        public static void main(String args[])
        {
                Demo d=new Demo();
        d.display();
        }
}
```

OUTPUT:

```
Internal Marks: 25
External Marks: 75
Total Marks: 100
```

NESTED INTERFACES

- In java, an interface may be defined inside another interface, and also inside a class. The interface that defined inside another interface or a class is known as nested interface.
- The nested interface declared within an interface is public by default.
- The nested interface declared within a class can be with any access modifier.
- Every nested interface is static by default.
- The nested interface cannot be accessed directly.
- The nested interface that defined inside another interface must be accessed as **OuterInterface.InnerInterface**.

Example: Nested interface inside another interface interface

```
OuterInterface
{
    void outerMethod();
    interface InnerInterface
    {
        void innerMethod();
    }
}
class OnlyOuter implements OuterInterface
{
    public void outerMethod()
    {
        System.out.println("This is OuterInterface method");
    }
}
class OnlyInner implements OuterInterface.InnerInterface
{
    public void innerMethod()
    {
        System.out.println("This is InnerInterface method");
    }
}
class NestedInterfaceExample
{
    public static void main(String[] args)
    {
        OnlyOuter obj_1 = new OnlyOuter();
    OnlyInner obj_2 = new OnlyInner();
        obj_1.outerMethod();
    obj_2.innerMethod();
```

```
    }
}
```

Nested interface inside a class

The nested interface that defined inside a class must be accessed as ClassName.InnerInterface.

Example :

```
class OuterClass
{

    interface InnerInterface
    {
        void innerMethod();
    }
}

class ImplementingClass implements OuterClass.InnerInterface
{
    public void innerMethod()
    {
        System.out.println("This is InnerInterface method");
    }
}

public class NestedInterfaceExample
{

    public static void main(String[] args)
    {
        ImplementingClass obj = new ImplementingClass();

        obj.innerMethod();
    }

}
```

Difference between Abstract class and Interface

	Abstract class	Interface
1	It is collection of abstract methods and concrete methods.	It is collection of abstract methods.
2	There properties can be reused commonly in a specific application.	There properties commonly usable in any application of java environment.
3	It does not support multiple inheritances.	It supports multiple inheritances.
4	Abstract class is preceded by abstract keyword.	Interface is preceded by Interface keyword.
5	Which may contain either variable or constants?	Which should contain only constants?
6	The default access specifier of abstract class methods are default.	There default access specifier of interface method are public.
7	These class properties can be reused in other class using extend keyword.	These properties can be reused in any other class using implements keyword.
8	Inside abstract class we can take constructor.	Inside interface we cannot take any constructor.
9	For the abstract class there is no restriction like initialization of variable at the time of variable declaration.	For the interface it should be compulsory to initialization of variable at the time of variable declaration.
10	There are no any restriction for abstract class variable.	For the interface variable cannot declare variable as private, protected, transient, volatile.
11	There are no any restriction for abstract class method modifier that means we can use any modifiers.	For the interface method cannot declare method as strictfp, protected, static, native, private, final, synchronized.

ABSTRACT CLASS: An abstract class should be declared by using the keyword abstract. An abstract class is a class that contains zero or more abstract methods. If any class extends the abstract class, the subclass should overrode or give anybody for all the method of the super class.

Rule 1: An abstract class can have all abstract methods and concrete methods or both.

Rule 2: The subclass should override all the abstract methods of the abstract super class.

Rule 3: An abstract class can contain instance variables.

Rule 4: An abstract class we cannot create object, but we can create reference only.

Syntax:

```
abstract class className
{
        ......
}
```

Abstract method

An abstract method contains only declaration or prototype but it never contains body or definition. In order to make any undefined method as abstract whose declaration is must be predefined by abstract keyword.

Syntax: abstract returntype methodName(List of formal parameter);

Example:

```
abstract class Test
{
        abstract void calculate(double x);
}
class Sub1 extends Test
{
        void calculate(double x)
        {
                System.out.println("Square ="+(x*x));
        }
}
class Sub2 extends Test
{
        void calculate(double x)
        {
                System.out.println("Square Root ="+Math.sqrt(x));
        }
```

```
}
class Sub3 extends Test
{
        void calculate(double x)
        {
                System.out.println("Cube ="+(x*x*x));
        }
}
class Abstract1
{
        public static void main(String args[])
        {
                Sub1 obj1 = new Sub1();
                Sub2 obj2 = new Sub2();
Sub3 obj3 = new Sub3();
obj1.calculate(3);
obj2.calculate(4);
obj3.calculate(5);
        }
}
```

OUTPUT:

javac Abstract.java

java Abstract1 Square

=9.0

Square Root =2.0

Cube =125.0

Types of Streams

Depending on the type of operations, streams can be divided into two primary classes:

- **InPutStream** – The InputStream is used to read data from a source.
- **OutPutStream** – The OutputStream is used for writing data to a destination.

Depending upon the data a stream can be classified into:

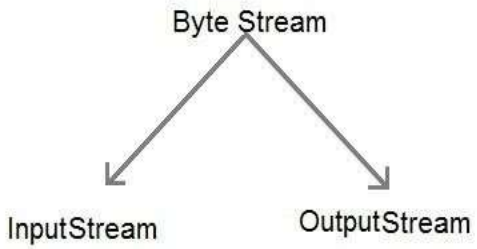
1. Byte Stream
2. Character Stream

1. Byte Stream

Java byte streams are used to perform input and output of 8-bit bytes.

Byte Stream Classes

All byte stream classes are derived from base abstract classes called **InputStream** and **OutputStream**.



InputStream Class

InputStream class is an abstract class. It is the superclass of all classes representing an input stream of bytes.

Subclasses of InputStream

In order to use the functionality of InputStream, we can use its subclasses. Some of them are:

Stream class	Description
BufferedInputStream	Used for Buffered Input Stream.
DataInputStream	Contains method for reading java standard datatype
FileInputStream	Input stream that reads from a file

Methods of InputStream

The InputStream class provides different methods that are implemented by its subclasses. Here are some of the commonly used methods:

- **read()** - reads one byte of data from the input stream
- **read(byte[] array)** - reads bytes from the stream and stores in the specified array
- **available()** - returns the number of bytes available in the input stream
- **mark()** - marks the position in the input stream up to which data has been read
- **reset()** - returns the control to the point in the stream where the mark was set
- **close()** - closes the input stream

OutputStream class

OutputStream class is an abstract class. It is the superclass of all classes representing an output stream of bytes.

Subclasses of OutputStream

In order to use the functionality of OutputStream, we can use its subclasses. Some of them are:

Stream class	Description
BufferedOutputStream	Used for Buffered Output Stream.
DataOutputStream	An output stream that contain method for writing java standard data type
FileOutputStream	Output stream that write to a file.
PrintStream	Output Stream that contain print() and println() method

Methods of OutputStream

The OutputStream class provides different methods that are implemented by its subclasses. Here are some of the methods:

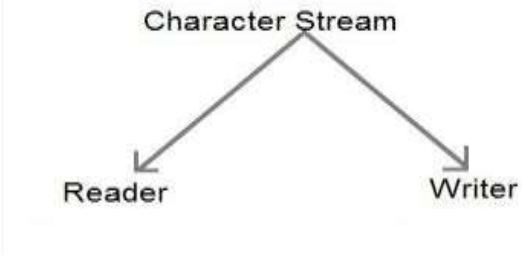
- **write()** - writes the specified byte to the output stream
- **write(byte[] array)** - writes the bytes from the specified array to the output stream
- **flush()** - forces to write all data present in output stream to the destination
- **close()** - closes the output stream

2. Character Stream

Character stream is used to read and write a single character of data.

Character Stream Classes

All the character stream classes are derived from base abstract classes Reader and Writer.



Reader Class

The Reader class of the java.io package is an abstract superclass that represents a stream of characters.

Sub classes of Reader Class

In order to use the functionality of Reader, we can use its subclasses. Some of them are:

Stream class	Description
BufferedReader	Handles buffered input stream.
FileReader	Input stream that reads from file.
InputStreamReader	Input stream that translate byte to character

Methods of Reader

The Reader class provides different methods that are implemented by its subclasses. Here are some of the commonly used methods:

- **ready() -** checks if the reader is ready to be read
- **read(char[] array) -** reads the characters from the stream and stores in the specified array
- **read(char[] array, int start, int length) -** reads the number of characters equal to length from the stream and stores in the specified array starting from the start
- **mark() -** marks the position in the stream up to which data has been read
- **reset() -** returns the control to the point in the stream where the mark is set
- **skip() -** discards the specified number of characters from the stream

Writer Class

The Writer class of the java.io package is an abstract superclass that represents a stream of characters.

Since Writer is an abstract class, it is not useful by itself. However, its subclasses can be used to write data.

Subclasses of Writer

Stream cl ıss	Description
BufferedWriter	Handles buffered output stream.
FileWriter	Output stream that writes to file.
PrintWriter	Output Stream that contain print() and println() method.

Methods of Writer

The Writer class provides different methods that are implemented by its subclasses. Here are some of the methods:

- **write(char[] array) -** writes the characters from the specified array to the output stream
- **write(String data) -** writes the specified string to the writer
- **append(char c) -** inserts the specified character to the current writer
- **flush() -** forces to write all the data present in the writer to the corresponding destination
- **close() -** closes the writer

Reading Console Input

There are times when it is important for you to get input from users for execution of programs. To do this you need Java Reading Console Input Methods.

Java Reading Console Input Methods

1. Using BufferedReader Class
2. Using Scanner Class 3. Using Console Class

Using BufferedReader Class

- Reading input data using the BufferedReader class is the traditional technique. This way of the reading method is used by wrapping the System.in (standard input stream) in an InputStreamReader which is wrapped in a BufferedReader, we can read input from the console.
- The BufferedReader class has defined in the java.io package.

- We can use read() method in BufferedReader to read a character. **int read() throws IOException**

Reading Console Input Characters Example:

```
import java.io.*;
class ReadingConsoleInputTest
{
        public static void main(String args[])
        {
            char ch;
            BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
           System.out.println("Enter characters, char 'x' to exit.");          do
            {       ch = (char) br.read();
                  System.out.println(ch);
            } while(ch != 'x');
        }
}
```

How to read a string input in java?

readLine() method is used to read the string in the BufferedReader.

Program to take String input from Keyboard in Java import java.io.*; class MyInput

```
{   public static void main(String[] args)
  {
        String text;
        InputStreamReader isr = new InputStreamReader(System.in);
   BufferedReader br = new BufferedReader(isr);      text =
   br.readLine();      //Reading String      System.out.println(text);
  }
}
```

Using the Scanner Class

Scanner is one of the predefined class which is used for reading the data dynamically from the keyboard.

Import Scanner Class in Java

java.util.Scanner

Constructor of Scanner Class

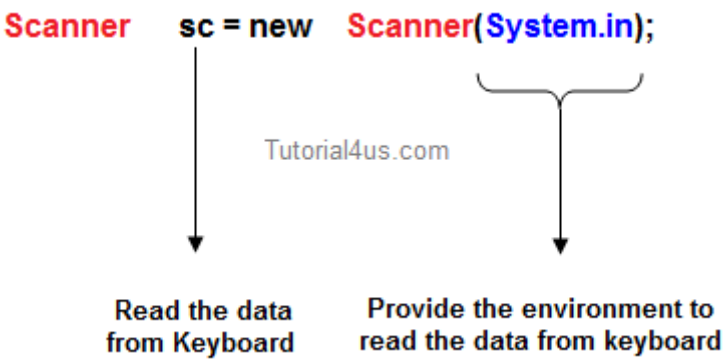
Scanner(InputStream)

This constructor create an object of Scanner class by talking an object of InputStream class. An object of InputStream class is called in which is created as a static data member in the System class.

Syntax of Scanner Class in Java

Scanner sc=new Scanner(System.in);

Here the object 'in' is use the control of keyboard



Instance methods of Scanner Class

S.No	Method	Description
1	public byte nextByte()	Used for read byte value
2	public short nextShort()	Used for read short value
3	public int nextInt()	Used for read integer value
4	public long nextLong()	Used for read numeric value
5	public float nextLong()	Used for read numeric value
6	public double nextDouble()	Used for read double value
7	public char nextChar()	Used for read character
8	public boolean nextBoolean()	Used for read boolean value
9	public String nextLine()	Used for reading any kind of data in the form of String data.

Example of Scanner Class in Java import java.util.Scanner public class ScannerDemo

```
{
    public static void main(String args[])
    {
        Scanner s=new Scanner(System.in);
        System.out.println("Enter first no= "); int
        num1=s.nextInt();
```

```
        System.out.println("Enter second no= "); int
        num2=s.nextInt();
        System.out.println("Sum of no is= "+(num1+num2));
    }
}
```

Using the Console Class

- This is another way of reading user input from the console in Java.
- The Java Console class is be used to get input from console. It provides methods to read texts and passwords.
- If you read password using Console class, it will not be displayed to the user.
- The Console class is defined in the java.io class which needs to be imported before using the console class.

Example

```
import java.io.*; class
consoleEg
{
    public static void main(String args[])
    {
        String name;
        System.out.println ("Enter your name: ");
Console c = System.console();          name =
c.readLine();
        System.out.println ("Your name is: " + name);
    }
}
```

Writing Console Output

- print and println methods in System.out are mostly used for console output.
- These methods are defined by the class PrintStream which is the type of object referenced by System.out.
- System.out is the byte stream.
- PrintStream is the output derived from OutputStream. write method is also defined in PrintStream for console output.

void write(int byteval)

//Java code to Write a character in Console Output

```
import java.io.*;
class WriteCharacterTest
```

```
{
        public static void main(String args[])
        {                               int
            byteval;
            byteval = 'J';
              System.out.write(byteval);
              System.out.write('\n');
        }
}
```

Java File Class

The File class of the java.io package is used to perform various operations on files and directories.

File and Directory

A file is a named location that can be used to store related information. For example, **main.java** is a Java file that contains information about the Java program.
A directory is a collection of files and subdirectories. A directory inside a directory is known as subdirectory.

Create a Java File Object

To create an object of File, we need to import the java.io.File package first. Once we import the package, here is how we can create objects of file.

```
File f = new File(String pathName);
```

Here, we have created a file object named file. The object can be used to work with files and directories.
Note: In Java, creating a file object does not mean creating a file. Instead, a file object is an abstract representation of the file or directory pathname (specified in the parenthesis).

Java File Operation Methods

Operation	Method	Package
To create file	createNewFile()	java.io.File
To read file	read()	java.io.FileReader
To write file	write()	java.io.FileWriter

To delete file	delete()	java.io.File

Java create files

To create a new file, we can use the createNewFile() method. It returns

- true if a new file is created.
- false if the file already exists in the specified location.

Example: Create a new File

```
// importing the File class
import java.io.File; class
MainDemo
{
        public static void main(String[] args)
        {
                // create a file object for the current location
                File f = new File("newFile.txt");
                try
                {
                        // trying to create a file based on the object
                    boolean value = f.createNewFile();      if (value)
                    {
                       System.out.println("The new file is created.");
                    }      else
                    {
                       System.out.println("The file already exists.");
                    }
                }
                catch(Exception e)
                {
                    e.getStackTrace();
                }
        }
}
```

In the above example, we have created a file object named f. The file object is linked with the specified file path.

File f = new File("newFile.txt");

Here, we have used the file object to create the new file with the specified path.

If newFile.txt doesn't exist in the current location, the file is created and this message is shown.

The new file is created.

However, if newFile.txt already exists, we will see this message. The file already exists.

Java read files

To read data from the file, we can use subclasses of either InputStream or Reader.

Example: Read a file using FileReader

Suppose we have a file named **input.txt** with the following content.

This is a line of text inside the file.

Now let's try to read the file using Java FileReader. import java.io.FileReader; class MainDemo2

```
{
        public static void main(String[] args)
        {
                char[] a = new char[100];
                try     {
                        // Creates a reader using the FileReader
                        FileReader input = new FileReader("input.txt");
                        // Reads characters      input.read(a);
                        System.out.println("Data in the file:");
                        System.out.println(a);      //
                   Closes the reader
                   input.close();
                }
                catch(Exception e)
                {
                 e.getStackTrace();
                }
        }
}
```

Java write to files

To write data to the file, we can use subclasses of either OutputStream or Writer.

Example: Write to file using FileWriter import java.io.FileWriter; class Main

```
{
            public static void main(String args[])
            {
                String data = "This is the data in the output file";
               try      {
                        // Creates a Writer using FileWriter
```

```
                FileWriter output = new FileWriter("output.txt");
                // Writes string to the file       output.write(data);
                System.out.println("Data is written to the file.");
                // Closes the writer       output.close();
            }
            catch (Exception e)
            {
                e.getStackTrace();
            }
        }
}
```

Random access file operations

The **Java.io.RandomAccessFile** class file behaves like a large array of bytes stored in the file system.

Constructor

Constructor	Description
RandomAccessFile(File file, String mode)	Creates a random access file stream to read from, and optionally to write to, the file specified by the File argument.
RandomAccessFile(String name, String mode)	Creates a random access file stream to read from, and optionally to write to, a file with the specified name.

Methods

Modifier and Type	Method	Description
void	close()	It closes this random access file stream and releases any system resources associated with the stream.
FileChannel	getChannel()	It returns the unique FileChannel object associated with this file.
int	readInt()	It reads a signed 32-bit integer from this file.

String	readUTF()	It reads in a string from this file.
void	seek(long pos)	It sets the file-pointer offset, measured from the beginning of this file, at which the next read or write occurs.
void	writeDouble(double v)	It converts the double argument to a long using the doubleToLongBits method in class Double, and then writes that long value to the file as an eight-byte quantity, high byte first.
void	writeFloat(float v)	It converts the float argument to an int using the floatToIntBits method in class Float, and then writes that int value to the file as a four-byte quantity, high byte first.
void	write(int b)	It writes the specified byte to this file.
int	read()	It reads a byte of data from this file.
long	length()	It returns the length of this file.
void	seek(long pos)	It sets the file-pointer offset, measured from the beginning of this file, at which the next read or write occurs.

Example

```
import java.io.IOException;   import
java.io.RandomAccessFile;    public class
RandomAccessFileExample
{
        static final String FILEPATH ="myFile.TXT";      public
    static void main(String[] args)
      {       try
              {
                  System.out.println(new String(readFromFile(FILEPATH, 0, 18)));
          writeToFile(FILEPATH, "I love my country and my people", 31);
              }
             catch (IOException e)
```

```
                {
                    e.printStackTrace();
                }
        }
    private static byte[] readFromFile(String filePath, int position, int size) throws IOException
    {
                    RandomAccessFile file = new RandomAccessFile(filePath, "r");
                file.seek(position);        byte[] bytes = new byte[size];
                file.read(bytes);        file.close();        return bytes;
    }
    private static void writeToFile(String filePath, String data, int position) throws IOException
    {
                    RandomAccessFile file = new RandomAccessFile(filePath, "rw");
    file.seek(position);        file.write(data.getBytes());        file.close();   }
}
```

Serialization

- In java, the **Serialization** is the process of converting an object into a byte stream so that it can be stored on to a file, or memory, or a database for future access.
- The reverse operation of serialization is called *deserialization* where byte-stream is converted into an object.
- Using serialization and deserialization, we can transfer the Object Code from one Java Virtual machine to another.
- For serializing the object, we call the writeObject() method *ObjectOutputStream*, and for deserialization we call the readObject() method of *ObjectInputStream* class.
- We must have to implement the *Serializable* interface for serializing the object.

Let's see the example given below:

```
import java.io.Serializable;
public class Student implements Serializable
{   int id;
        String name;    public Student(int id,
        String name)
        {
            this.id = id;
            this.name = name;
        }
}
```

In the above example, Student class implements Serializable interface. Now its objects can be converted into stream.

ObjectOutputStream class

The ObjectOutputStream class is used to write primitive data types, and Java objects to an OutputStream. Only objects that support the java.io.Serializable interface can be written to streams.

Constructor

1) public ObjectOutputStream(OutputStream out) throws IOException {}	creates an ObjectOutputStream that writes to the specified OutputStream.

Important Methods

Method	Description
1) public final void writeObject(Object obj) throws IOException {}	writes the specified object to the ObjectOutputStream.
2) public void flush() throws IOException {}	flushes the current output stream.
3) public void close() throws IOException {}	closes the current output stream.

Example of Java Serialization

In this example, we are going to serialize the object of Student class.
The writeObject() method of ObjectOutputStream class provides the functionality to serialize the object.
We are saving the state of the object in the file named f.txt.

```
import java.io.*;   class
Persist
{
        public static void main(String args[])
        {    try
         {
               //Creating the object
               Student s1 =new Student(211,"ravi");
               //Creating stream and writing the object
               FileOutputStream fout=new FileOutputStream("f.txt");
              ObjectOutputStream out=new ObjectOutputStream(fout);
              out.writeObject(s1);     out.flush();
               //closing the stream     out.close();
               System.out.println("success");
```

```
        }
      catch(Exception e)
      {
          System.out.println(e);}
      }
}
```

Enumeration

- The Enum in Java is a data type which contains a fixed set of constants.

- It can be used for days of the week (SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, and SATURDAY) , directions (NORTH, SOUTH, EAST, and WEST), season (SPRING, SUMMER, WINTER, and AUTUMN or FALL), colors (RED, YELLOW, BLUE, GREEN, WHITE, and BLACK) etc.
- According to the Java naming conventions, we should have all constants in capital letters. So, we have enum constants in capital letters.
- Java Enums can be thought of as classes which have a fixed set of constants (a variable that does not change).
- The Java enum constants are static and final implicitly.
- Enums are used to create our own data type like classes.
- The enum data type (also known as Enumerated Data Type) is used to define an enum in Java. Unlike C/C++, enum in Java is more *powerful*. Here, we can define an enum either inside the class or outside the class.
- Java Enum internally inherits the *Enum class*, so it cannot inherit any other class, but it can implement many interfaces. We can have fields, constructors, methods, and main methods in Java enum.

Points to remember for Java Enum

- Enum improves type safety
- Enum can be easily used in switch
- Enum can be traversed
- Enum can have fields, constructors and methods
- Enum may implement many interfaces but cannot extend any class because it internally extends Enum class

Simple Example of Java Enum class

```
EnumExample1
{
   public enum Season { WINTER, SPRING, SUMMER, FALL}   public
   static void main(String[] args)
   {
```

```
        for (Season s : Season.values())   System.out.println(s);
    }
}
```

Autoboxing

The automatic conversion of primitive data types into its equivalent Wrapper type is known as boxing.

(OR)

Converting a primitive value into an object of the corresponding wrapper class is called autoboxing.

For example, converting int to Integer class.

No need of conversion between primitives and Wrappers manually so less coding is required.

The following table lists the primitive types and their corresponding wrapper classes, which are used by the Java compiler for autoboxing:

Primitive type	Wrapper Class
boolean	Boolean
byte	Byte
char	Character
float	Float
int	Integer
long	Long
short	Short
double	Double

Simple Example of Autoboxing in java:

```
class BoxingExample
{
    public static void main(String args[])
    {
        int a=50;
        Integer a2=new Integer(a);//Boxing
        Integer a3=5;//Boxing
        System.out.println(a2+" "+a3);
    }
}
```

Output: 50 5

Generics in Java

The Java Generics programming is introduced 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:

Type-safety: We can hold only a single type of objects in generics. It doesnt 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
```

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);
```

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>
```