UNIT - II

Packages in Java

A package is a collection of similar types of classes, interfaces and sub-packages.

Types of packages

Package are classified into two type which are given below.

1. Predefined or built-in package
2. User defined package

1.Predefined or built-in package

These are the package which are already designed by the Sun Microsystem and supply as a part of java API, every predefined package is collection of predefined classes, interfaces and sub-package.

Following are the list of predefined packages in java

- **java.lang** – This package provides the language basics.
- **java.util** – This packages provides classes and interfaces (API's) related to collection frame work, events, data structure and other utility classes such as date.
- **java.io** – This packages provides classes and interfaces for file operations, and other input and output operations.
- **java.awt** – This packages provides classes and interfaces to create GUI components in Java.
- **java.time** – The main API for dates, times, instants, and durations.

2.User defined package

If any package is design by the user is known as user defined package. User defined package are those which are developed by java programmer and supply as a part of their project to deal with common requirement.

Defining a package

To create a package include a package command as the first statement in a Java source file. Any classes declared within that file will belong to the specified package.

If you omit the package statement, the class names are put into the default package, which has no name.

Syntax

```
package packagename;
```

Example

```
package mypack;
```

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
```

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 A.java and compile by javac -d . A.java

```
package mypack;
public class A
{
    public void show()
    {
        System.out.println("Sum method");
    }
}
```

Importing Packages

Import above class in below program using import packageName.className

Example

```
import mypack.A;
public class Hello
{
    public static void main(String args[])
    {
```

```
            A a=new A();
            a.show();
            System.out.println("show() class A");
        }
    }
```

Explanation: In the above program first we create Package program which is save with A.java and compiled by "javac -d . A.java". Again we import class "A" in class Hello using **"import mypack.A;"** statement.

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

Packages act as containers for classes and other subordinate packages. Classes act as containers for data and code. The class is Java's smallest unit of abstraction.

Because of the interplay between classes and packages, Java addresses four categories of visibility for class members:

- Subclasses in the same package
- Non-subclasses in the same package
- Subclasses in different packages
- Classes that are neither in the same package nor subclasses.

Let us see how access specifiers are applied here:

- Anything declared **public** can be accessed from anywhere.
- Anything declared **private** cannot be seen outside of its class.
- When a member does not have an explicit access specification, it is visible to subclasses as well as to other classes in the same package. This is the **default access**.
- If you want to allow an element to be seen outside your current package, but only to classes that subclass your class directly, then declare that element **protected.**

Table: class member access

	private	No-modifier	protected	public
Same class	yes	yes	yes	yes
Same package subclass	No	yes	yes	yes
Same package non-subclass	No	yes	yes	yes
Different package subclass	No	No	yes	yes
Different package non-subclass	No	No	No	yes

Interfaces

Interface is similar to class which is collection of public static final variables (constants) and abstract methods.

The interface is a mechanism to achieve fully abstraction in java. There can be only abstract methods in the interface. It is used to achieve fully abstraction and multiple inheritance in Java.

Difference between Abstract class and Interface

Abstract class	Interface
It is collection of abstract method and concrete methods.	It is collection of abstract method.
There properties can be reused commonly in a specific application.	There properties commonly usable in any application of java environment.
It does not support multiple inheritance.	It support multiple inheritance.
Abstract class is preceded by abstract	It is preceded by Interface keyword.

keyword.	
Which may contain either variable or constants.	Which should contains only constants.
The default access specifier of abstract class methods are default.	There default access specifier of interface method are public.
These class properties can be reused in other class using extend keyword.	These properties can be reused in any other class using implements keyword.
Inside abstract class we can take constructor.	Inside interface we can not take any constructor.
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.
There are no any restriction for abstract class variable.	For the interface variable can not declare variable as private, protected, transient, volatile.
There are no any restriction for abstract class method modifier that means we can use any modifiers.	For the interface method can not declare method as strictfp, protected, static, native, private, final, synchronized.

Defining Interfaces:

The **interface** keyword is used to declare an interface.

Syntax

```
interface interface_name
{
   declare constant fields
   declare methods that abstract
}
```

Example

```
interface A
 {
      Public static final int a = 10;
```

```
        void display();
}
```

Implementing Interfaces

A class uses the **implements** keyword to implement an interface.

Example

```
interface A
{
    Public static final int a = 10;
    void display();
}
class B implements A
{
  public void display()
  {
    System.out.println("Hello");
  }
}
class InterfaceDemo
{
  public static void main (String[] args)
  {
    A obj= new A();
    obj.display();
    System.out.println(a);
  }
}
```

Applying Interfaces

To understand the power of interfaces, let's look at a more practical example.
Here is the interface that defines an integer stack. Put this in a file called *IntStack.java.*

Example

```
interface IntStack
{
    void push(int item);
    int pop();
}
```

```
class FixedStack implements IntStack
{
     private int stck[];
     private int top;
     FixedStack(int size)
     {
         stck = new int[size];
         top = -1;
     }
     public void push(int item)
     {
           if(top==stck.length-1)
           System.out.println("Stack is full.");
           else
           stck[++top] = item;
    }
    public int pop()
    {
       if(top ==-1)
       {
            System.out.println("Stack underflow.");
            return 0;
       }
      else
     return stck[top--];
   }
}
class InterfaceTest
{
           public static void main(String args[])
           {
                  FixedStack mystack1 = new FixedStack(5);
                  FixedStack mystack2 = new FixedStack(8);
                  for(int i=0; i<5; i++)
                   mystack1.push(i);
                  for(int i=0; i<8; i++)
                   mystack2.push(i);
                  for(int i=0; i<5; i++)
```

```
                System.out.println(mystack1.pop());
                for(int i=0; i<8; i++)
                System.out.println(mystack2.pop());
            }
        }
```

Variables in interface

Variables can be declared inside of interface declarations. They are implicitly final and static, meaning they cannot be changed by the implementing class.

You can use interfaces to import shared constants into multiple classes by simply declaring an interface that contains variables that are initialized to the desired values.

Example

```
interface SharedConstants
{
        int NO = 0;
        int YES = 1;
        int MAYBE = 2;
        int LATER = 3;
        int NEVER = 4;
}
class Question implements SharedConstants
{
    BufferedReader br=new BufferedReader(new InputStreamreader(System.in));
    int ask()
    {
            System.out.println("would u like to have a cup of coffee?)
            String ans=br.readLine();
            if (ans= ="no")
            return NO;
            else if (ans=="yes")
            return YES;
            else if (ans=="notnow")
            return LATER;
            else
            return NEVER;
     }
}
```

```
class AskMe
{
        public static void main(String args[])
        {
              Question q = new Question();
              System.out.println(q.ask());
        }
}
```

Extending interfaces

One interface can inherit another by use of the keyword **extends**.

When a class implements an interface that inherits another interface, it must provide implementations for all methods defined within the interface inheritance chain.

Example

```
 interface A
 {
   void meth1();
   void meth2();
 }
 interface B extends A
 {
    void meth3();
 }
class MyClass implements B
{
     public void meth1()
     {
          System.out.println("Implement meth1().");
     }
     public void meth2()
     {
         System.out.println("Implement meth2().");
     }
     public void meth3()
     {
          System.out.println("Implement meth3().");
     }
 }
```

```
class InterfaceDemo
{
    public static void main(String arg[])
    {
          MyClass ob = new MyClass();
          ob.meth1();
          ob.meth2();
          ob.meth3();
     }
 }
```

Nested Interface

- 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.
- 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 implicitely.

Syntax of nested interface which is declared within the interface

```
interface interface_name
{
    ..........
    interface nested_interface_name
    {
        ..........
    }
}
```

Syntax of nested interface which is declared within the class

```
class class_name
{
      ...
      interface nested_interface_name
      {
          ...
```

```
    }
}
```

Example of nested interface which is declared within the interface

In this example, we are going to learn how to declare the nested interface and how we can access it.

```
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();
        }
}
```

Example of nested interface which is declared within the class

Let's see how can we define an interface inside the class and how can we access it.

```
class A
{
        interface Message
        {
           void msg();
        }
}
class TestNestedInterface2 implements A.Message
```

```
{
    public void msg()
    {
        System.out.println("Hello nested interface");
    }
    public static void main(String args[])
    {
        A.Message message=new TestNestedInterface2();
        message.msg();
    }
}
```

Stream Based I/O (java.io)

Java I/O (Input and Output) is used *to process the input* and *produce the output.*
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.

We can perform **file handling in Java** by Java I/O API.

Stream

In Java, streams are the sequence of data that are read from the source and written to the destination.

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

1.System.in: This is the **standard input stream** that is used to read characters from the keyboard or any other standard input device.
2.System.out: This is the **standard output stream** that is used to produce the result of a program on an output device like the computer screen.
3.System.err: This is the **standard error stream** that is used to output all the error data that a program might throw, on a computer screen or any standard output device.

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:

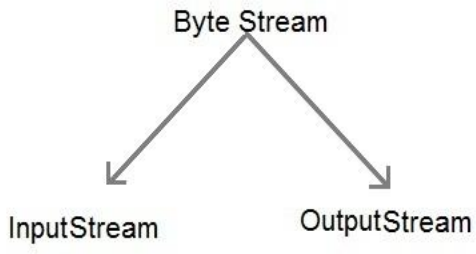
1. Byte Stream
2. Character Stream

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:

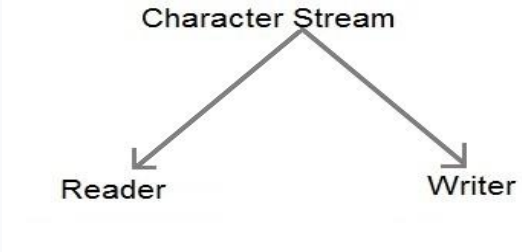
- **write()** - writes the specified byte to the output stream
- **write(byte[] array)** - writes the bytes from the specified array to the output stream
- **flush()** - forces to write all data present in output stream to the destination
- **close()** - closes the output stream

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 class	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

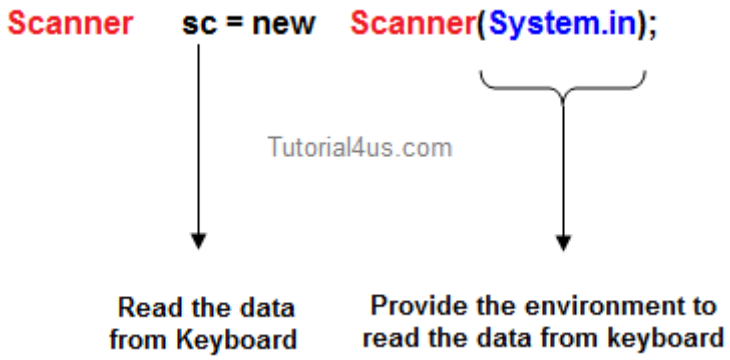
Scanner(InputStream)

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>
```