**Java IO Basics**

Java I/O stream is the flow of data that you can either read from, or you can write to.  
It is used to perform read and write operations in file permanently. Java uses streams to perform these tasks. **Java I/O stream** is also called **File Handling**, or **File I/O.**It is available in **java.io** package.  
  
Java.io package provides classes for system input and output through files, network streams, memory buffers, etc.  
  
Some input-output stream will be initialized automatically by the JVM and these streams are available in System class as **in**, **out**, and **err** variable.

* **In** reference refers to the default input device, i.e. keyboard.
* **Out**and**err**refers to the default output device, i.e. console.

**Streams:**

Streams are the sequence of bits (data).  
There are two types of streams:

* Input Streams
* Output Streams

**Input Streams:**

Input streams are used to read the data from various input devices like keyboard, file, network, etc.

**Output Streams:**

Output streams are used to write the data to various output devices like monitor, file, network, etc.

**Streams based on data**

There are two types of streams based on data:

* **Byte Stream**: used to read or write byte data.
* **Character Stream**: used to read or write character data.

**Byte Input Stream:**

* These are used to read byte data from various input devices.
* Input Stream is an abstract class and it is the super class of all the input byte streams.

List of Byte Input Streams:

**Byte Output Stream:**

* These are used to write byte data to various output devices.
* Output Stream is an abstract class and it is the superclass for all the output byte streams.

List of Byte Output Streams:

**Character Input Stream:**

* These are used to read char data from various input devices.
* Reader is an abstract class and is the super class for all the character input streams.

List of Character Input Streams:

**Character Output Stream:**

* These are used to write char data to various output devices.
* Writer is an abstract class and is the super class of all the character output streams.

**List of Character Output Stream:**

**Read File:**

 package JavaIO;  
  
import java.io.FileInputStream;  
import java.io.IOException;  
  
public class ReadFile  
{  
 public static void main(String args[]) throws IOException  
 {  
 FileInputStream fi = new FileInputStream("src/JavaIO/Names.txt") ;  
 int i =0;  
 while((i = fi.read())!= -1)  
 {  
 System.*out*.print((char)i);  
 }  
 fi.close();  
 }  
}

**Read Line:**

package JavaIO;  
  
import java.io.BufferedReader;  
import java.io.FileReader;  
import java.io.IOException;  
  
public class ReadLine  
{  
 public static void main(String args[]) throws IOException  
 {  
 BufferedReader br = new BufferedReader(new FileReader("src/JavaIO/Names.txt"));  
 String line = br.readLine();  
 while(line != null)  
 {  
 System.*out*.println(line+ " "+ line.length());  
 line = br.readLine();  
 }  
 }  
}

**Write File:**

package JavaIO;  
  
import java.io.FileNotFoundException;  
import java.io.FileOutputStream;  
import java.io.IOException;  
  
public class WriteFile  
{  
 public static void main(String args[]) throws IOException {  
 FileOutputStream fo = new FileOutputStream("src/JavaIO/Names.txt");  
 String names = "Anuj,Nitin,Amey,Hitesh,Amit,Amar";  
 byte arr[] = names.getBytes();  
 fo.write(arr);  
 fo.close();  
 }  
}

**What is Serialization in Java?**

Serialization in Java is a process that allows objects to be converted into a stream of bytes, which can be easily stored, transmitted, or reconstructed later. It facilitates the persistence of object data and enables communication between different [Java](https://www.shiksha.com/online-courses/what-is-java-st619-tg17) applications. The serialized data can be saved in files, sent over networks, or stored in databases, making it a crucial mechanism for data exchange and storage in Java programs.

**Why do we use Serialization?**

In Java, to run a program, first, we create a class, and then we create functions or objects to fulfill our requirements for the program. We can execute a program in Java without creating an object, but in this case, we will assume that we have created an object for our program.

Now, when our program executes with an object, it takes some disk storage to run the program, but after execution, the whole disk storage got destroyed by itself. As we know, Java has a garbage collector by default. It clears the memory, and if we have to execute the same object again, it recreates the disk memory for the object. To overcome this problem, we use Serialization, convert the data into a byte stream, and store that on disk storage.

**Benefits of Serialization**

Serialization has several benefits, so let us check these out,

I. **Persistence**: Serialization makes it simple to store objects in a file or in a database and retrieve them later, which makes it easy to implement persistent storage in applications. This allows the application to store the object for long-term purposes, such as enterprise applications and data-intensive systems.

II. **Interoperability**: As we read above, Serialization is JVM-independent, which means it is a platform-independent way of transmitting objects between different systems. This is because serialized objects are represented as a stream of bytes, which can be transmitted over a network and then deserialized on the receiving end to create an identical object.

III. **Security**: Serialization also gives our program more security as it can be used to implement secure communication between different networks.

IV.**Performance & Flexibility**: As we read above it helps us to reuse the objects we need repeatedly and save time to run programs. It also gives us flexibility in the development and design of Java applications.

**To make a Java object serializable we implement** the **java.io.Serializable** interface. The ObjectOutputStream class contains **writeObject()** method for serializing an Object. 

**public final void writeObject(Object obj) throws IOException**

The ObjectInputStream class contains **readObject()** method for deserializing an object. 

**public final Object readObject() throws IOException, ClassNotFoundException**

package JavaIO;  
  
import java.io.Serializable;  
  
public class SerializeEmployee implements Serializable  
{  
 private int id;  
 private String name;  
 private int age;  
  
 public SerializeEmployee(int id, String name, int age) {  
 this.id = id;  
 this.name = name;  
 this.age = age;  
 }  
  
 public int getId() {  
 return id;  
 }  
  
 public void setId(int id) {  
 this.id = id;  
 }  
  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
  
 public int getAge() {  
 return age;  
 }  
  
 public void setAge(int age) {  
 this.age = age;  
 }  
  
 @Override  
 public String toString() {  
 return "Employee{" +  
 "id=" + id +  
 ", name='" + name + '\'' +  
 ", age=" + age +  
 '}';  
 }  
}

package JavaIO;  
  
import java.io.FileNotFoundException;  
import java.io.FileOutputStream;  
import java.io.IOException;  
import java.io.ObjectOutputStream;  
  
public class Serialization  
{  
 public static void main(String args[]) throws IOException {  
 SerializeEmployee emp1 = new SerializeEmployee(111,"Ram",21);  
 FileOutputStream fo = new FileOutputStream("src/JavaIO/Employee.ser");  
 ObjectOutputStream out = new ObjectOutputStream(fo);  
 out.writeObject(emp1);  
 System.*out*.println("Serialization done for Employee in the file Employee.ser ");  
 }  
}

package JavaIO;  
  
import java.io.FileInputStream;  
import java.io.IOException;  
import java.io.ObjectInputStream;  
  
public class Deserialization  
{  
 public static void main(String args[]) throws IOException, ClassNotFoundException {  
 FileInputStream fi = new FileInputStream("src/JavaIO/Employee.ser");  
 ObjectInputStream oi = new ObjectInputStream(fi);  
 SerializeEmployee e =(SerializeEmployee) oi.readObject();  
 System.*out*.println("Deserialized Objects Value "+ e.getId()+" "+e.getName()+" "+e.getAge());  
 }  
}

**New IO (NIO)**

[Java IO](https://www.geeksforgeeks.org/java-io-packag/)(Input/Output) is used to perform read and write operations. The [java.io package](https://www.geeksforgeeks.org/java-io-packag/) contains all the classes required for input and output operation. Whereas, Java NIO (New IO) was introduced from JDK 4 to implement high-speed IO operations. It is an alternative to the standard IO API’s.

In JDK4, NIO is introduced, which can meet the needs of Java program I/O to the greatest extent, and there are three core components in NIO: Channel, Buffer, Selector, traditional IO is stream-oriented, you can read one or more bytes from the stream at a time, you can only read backwards, not forward, NIO is buffer-oriented, and the data is read into a buffer, which can be moved forward/backward in the buffer.

NIO has three core parts: Channel, Buffer, and Selector. While traditional IO operates based on byte and character streams, NIO operates on Channel and Buffer, where data is always read from the channel into the buffer or written from the buffer to the channel. The Selector is used to listen for events from multiple channels (e.g., connection opened, data arrived). As a result, a single thread can listen on multiple data channels.

**Channels, Buffers, Selectors**

**Channel**

First of all, let's talk about Channel, which is mostly translated as "channel" in China. Channels and streams in IO are about the same level. However, the stream is one-way, for example: InputStream, OutputStream, and the channel is bidirectional, which can be used for both read and write operations.  
The main implementations of Channels in NIO are:

* FileChannel
* DatagramChannel
* SocketChannel
* ServerSocketChannel

You can guess the reason by looking at the name here: it can correspond to file IO, UDP and TCP (Server and Client) respectively. The example presented below basically revolves around these four types of channels.

**Buffer**

The key buffer implementations in NIO are: ByteBuffer, CharBuffer, DoubleBuffer, FloatBuffer, IntBuffer, LongBuffer, and ShortBuffer, which correspond to the basic data types: byte, char, double, float, int, long, short. Of course, there are also MappedByteBuffer, HeapByteBuffer, DirectByteBuffer, etc. in NIO.

**Selector**

Selectors run a single thread for multiple channels, and it's handy to use a selector if your app has multiple channels open, but traffic is low for each connection. For example, in a chat server. To use a Selector, you need to register a Channel with the Selector and then call its select() method. This method blocks until an event is ready for a registered channel. Once the method returns, the thread can process these events, such as a new connection coming in, data being received, and so on.

package JavaIO;  
  
import java.io.IOException;  
import java.io.RandomAccessFile;  
import java.nio.ByteBuffer;  
import java.nio.channels.FileChannel;  
  
public class ReadNIO  
{  
 public static void main(String args[]) throws IOException  
 {  
 RandomAccessFile file = new RandomAccessFile("src/JavaIO/Names.txt","r");  
 FileChannel channel = file.getChannel();  
 ByteBuffer buf = ByteBuffer.*allocate*(1024);  
 int r ;  
 while((r = channel.read(buf))!= -1)  
 {  
 buf.flip();  
 while(buf.hasRemaining())  
 {  
 System.*out*.print((char)buf.get());  
 }  
 buf.clear();  
 }  
 }  
}

**Java 8 Date and Time API**

package JavaIO;  
  
import java.sql.Date;  
import java.time.LocalDate;  
import java.time.chrono.ChronoLocalDate;  
import java.time.temporal.ChronoUnit;  
  
public class DaysBetweenDate {  
 public static void main(String[] args)  
 {  
 LocalDate firstDate = LocalDate.*now*();  
 LocalDate SecondDate = LocalDate.*of*(2027,11,2);  
 long d = ChronoUnit.*DAYS*.between(firstDate,SecondDate);  
 System.*out*.println("First Date : "+firstDate + " Second Date : "+SecondDate);  
 System.*out*.println("Days Between : "+d);  
 }  
}

**Java Networking**

**URL:**

package JavaIO;  
  
import java.io.BufferedReader;  
import java.io.IOException;  
import java.io.InputStreamReader;  
import java.net.HttpURLConnection;  
import java.net.URL;  
  
public class NetworkingURL  
{  
 public static void main(String[] args) throws IOException {  
 URL ur = new URL("https://www.simplilearn.com/java-networking-article");  
 String protocol = ur.getProtocol();  
 System.*out*.println(protocol);  
 int p = ur.getPort();  
 String f = ur.getFile();  
 System.*out*.println("PORT : " + p + "File :" + f);  
 HttpURLConnection con = (HttpURLConnection) ur.openConnection();  
 /\* con.setRequestMethod("Get"); \*/  
 int res = con.getResponseCode();  
 System.*out*.println(res);  
 BufferedReader br = new BufferedReader(new InputStreamReader(con.getInputStream()));  
 String line = br.readLine();  
 System.*out*.println(line);  
 }  
}

**Client-Server Application:**

package JavaIO;  
  
import java.io.Serializable;  
  
public class NetworkingCalculation implements Serializable {  
 private int x;  
 private int y;  
 private String Operation;  
  
 public int getX() {  
 return x;  
 }  
  
 public void setX(int x) {  
 this.x = x;  
 }  
  
 public int getY() {  
 return y;  
 }  
  
 public void setY(int y) {  
 this.y = y;  
 }  
  
 public String getOperation() {  
 // *TODO Auto-generated method stub* return null;  
 }  
  
 public void setOperation(String operation) {  
 this.Operation = operation;  
 }  
  
 public NetworkingCalculation(int x, int y, String string) {  
 super();  
 this.x = x;  
 this.y = y;  
 }  
}

package JavaIO;  
  
import java.io.IOException;  
import java.io.ObjectInputStream;  
import java.io.ObjectOutputStream;  
import java.net.ServerSocket;  
import java.net.Socket;  
  
public class NetworkingServer {  
 public static void main(String[] args) throws IOException, ClassNotFoundException {  
 ServerSocket serverSocket = new ServerSocket(9999);  
 while(true) {  
 Socket client = serverSocket.accept();  
 System.*out*.println("Connected");  
 ObjectInputStream oi = new ObjectInputStream(client.getInputStream());  
 /\*  
 \* DataInputStream di = new DataInputStream(client.getInputStream()); String s =  
 \* di.readUTF(); System.out.println("Message Received from Client "+s);  
 \* di.close();  
 \*/  
 NetworkingCalculation ob= (NetworkingCalculation) oi.readObject();  
 serverSocket.close();  
 int result;  
 int a = ob.getX();  
 int b = ob.getY();  
 String operation = ob.getOperation();  
 switch(operation) {  
 case "+":  
 result=a+b;  
 break;  
  
 case"-":  
 result=a-b;  
 break;  
 case "\*" :  
 result=a\*b;  
 break;  
  
 case "/" :  
 result=a/b;  
 break;  
  
 default:  
 result = -1;  
 }  
 ObjectOutputStream o = new ObjectOutputStream(client.getOutputStream());  
 o.write(result);  
 o.flush();  
 client.close();  
 }  
 }  
}

package JavaIO;  
  
import java.io.IOException;  
import java.io.ObjectInputStream;  
import java.io.ObjectOutputStream;  
import java.net.Socket;  
  
public class NetworkingClient  
{  
 public static void main(String[] args) throws IOException {  
 try {  
 Socket socket = new Socket("local host", 9999);  
 ObjectOutputStream out = new ObjectOutputStream(socket.getOutputStream());  
 String s = "Calculate";  
 NetworkingCalculation obj = new NetworkingCalculation(20, 30, "\*");  
 System.*out*.println("X: "+obj.getX()+"Y: "+obj.getY()+"Operation: "+obj.getOperation());  
 out.writeObject(obj);  
 /\* out.flush(); \*/  
 ObjectInputStream oi = new ObjectInputStream(socket.getInputStream());  
  
 int responseResult = oi.readInt();  
 System.*out*.println("The result is: "+responseResult);  
  
 } catch (IOException e)  
 {  
 e.printStackTrace();  
 }  
 }  
}