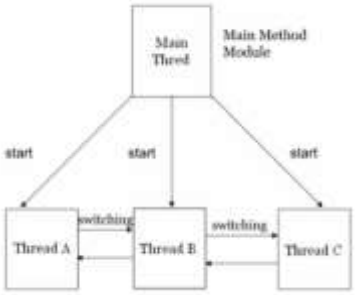
UNIT 4
MULTITHREADING, IO STREAMS

MULTITHREADING

- Multithreading in Java is a process of executing multiple threads simultaneously.
- A thread is a lightweight sub-process, the smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking.
- However, we use multithreading than multiprocessing because threads use a shared memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.
- Java Multithreading is mostly used in games, animation, etc.



Advantages of Java Multithreading

1) It doesn't block the user because threads are independent and you can perform multiple operations at the same time.
2) You can perform many operations together, so it saves time.
3) Threads are independent, so it doesn't affect other threads if an exception occurs in a single thread.

Main Thread

- The main thread in Java is a crucial component of any Java program. The thread is automatically created when a Java program starts, and maintains the main() method of the application. The main() method, which serves as the program's entrance point, is the initial method utilised at the start of the programme.
- The main thread works as a single thread, which means that each line of code is executed sequentially. Other names for this thread are "main application thread" and "main thread".
- Any additional threads that the programme may require must be created and started by the main thread. It is also in charge of organising how these threads are carried out and making sure they function effectively.

static Thread.currentThread():

The currentThread() method of thread class is used to return a reference to the currently executing thread object.

```
class EgThread
{
        public static void main(String args[])
        {
                Thread t=Thread.currentThread();
                System.out.println(t);
                t.setName("myfirstThread");
                System.out.println(t);
                System.out.println(t.getPriority());
                t.setPriority(10);
                System.out.println(t.getPriority());
        }
}
```

```
E:\JAVA\ANU>javac EgThread.java

E:\JAVA\ANU>java EgThread
Thread[#1,main,5,main]
Thread[#1,myfirstThread,5,main]
5
10
```

Thread:

- A thread is a lightweight subprocess, the smallest unit of processing. It is a separate path of execution.
- Threads are independent. If there occurs exception in one thread, it doesn't affect other threads. It uses a shared memory area.

Creating Threads

A thread can programmatically be created by:

- Implementing the java.lang.Runnable interface.
- Extending the java.lang.Thread class.

By implementing Runnable interface

The Runnable interface contains the run() method that is required for implementing the threads in our program. To do this we must perform the following steps:

- Declare a class as implementing the Runnable interface.
- Implement the run() method
- Create a Thread by defining an object that is instantiated from this "runnable" class as the target of the thread
- Call the thread's start() method to run the thread

```
class ExThrd implements Runnable
{
        public void run()
        {
                System.out.println("Thread is running");
        }

        public static void main(String args[])
        {
                ExThrd e=new ExThrd();
                Thread t=new Thread(e);
                t.start();
        }
}
```

```
E:\JAVA\ANU>javac ExThrd.java

E:\JAVA\ANU>java ExThrd
Thread is running
```

By extending Thread class

We can make our thread by extending the Thread class of java.lang.Thread class. This gives us access to all the methods of the Thread. It includes the following steps:

- Declare the class as Extending the Thread class.
- Override the "run()" method that is responsible for running the thread.
- Create a thread and call the "start()" method to instantiate the Thread Execution.

```
class EgThrd extends Thread
{
        public void run()
        {
                System.out.println("Thread is running");
        }

        public static void main(String args[])
        {
                EgThrd e=new EgThrd();
                e.start();
        }
}
```

```
E:\JAVA\ANU>javac EgThrd.java

E:\JAVA\ANU>java EgThrd
Thread is running
```

Life of a Thraed: (read from text book)

Thread Methods: (read from text book)

Thread Priority

Each thread has a priority. Priorities are represented by a number between 1 and 10. In most cases, the thread scheduler schedules the threads according to their priority. Note that not only JVM a Java programmer can also assign the priorities of a thread explicitly in a Java program.

Setter & Getter Method of Thread Priority

- **getPriority():** The java.lang.Thread.getPriority() method returns the priority of the given thread.
- **setPriority(int newPriority):** The java.lang.Thread.setPriority() method updates or assign the priority of the thread to newPriority. The method throws IllegalArgumentException if the value newPriority goes out of the range, which is 1 (minimum) to 10 (maximum).

3 constants defined in Thread class:

- public static int MIN_PRIORITY
- public static int NORM_PRIORITY
- public static int MAX_PRIORITY

Default priority of a thread is 5 (NORM_PRIORITY). The value of MIN_PRIORITY is 1 and the value of MAX_PRIORITY is 10.

```
class EgThread extends Thread
{
        public void run()
        {
                System.out.println("thread is running");
        }
        public static void main(String args[])
        {
                EgThread t1=new EgThread();
                EgThread t2=new EgThread();
                EgThread t3=new EgThread();

                System.out.println(t1.getPriority());
                System.out.println(t2.getPriority());
                System.out.println(t3.getPriority());

                t1.setPriority(1);
                t2.setPriority(10);
                t3.setPriority(6);

                System.out.println(t1.getName());
                System.out.println(t2.getName());
                System.out.println(t3.getName());

                System.out.println(t1.getPriority());
                System.out.println(t2.getPriority());
                System.out.println(t3.getPriority());

        }
}
```

```
E:\JAVA\ANU>javac EgThread.java

E:\JAVA\ANU>java EgThread
5
5
5
Thread-0
Thread-1
Thread-2
1
10
6
```

Synchronization

When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time. The process by which this is achieved is called synchronization.

The key to synchronization is the concept of the monitor (also called a semaphore). A monitor is an object that is used as a mutually exclusive lock, or mutex. Only one thread can own a monitor at a given time. When a thread acquires a lock, it is said to have entered the monitor. All other threads attempting to enter the locked monitor will be suspended until the first thread exits the monitor. These other threads are said to be waiting for the monitor. A thread that owns a monitor can renter the same monitor if it so desires.

```
class Counter
{
	int count;
	synchronized public void inc()
	{
		count++;
	}
}
class MyThread1 extends Thread
{
	Counter c;
	MyThread1(Counter c)
	{
		this.c=c;
	}
	public void run()
	{
		for(int i=1;i<=1000;i++)
		{
			c.inc();
		}
	}
}
class MyThread2 extends Thread
{
	Counter c;
	MyThread2(Counter c)
	{
		this.c=c;
	}
	public void run()
	{
		for(int i=1;i<=1000;i++)
		{
			c.inc();
		}
	}
}
public class Synch
{
	public static void main(String args[])
	{
		Counter c=new Counter();
		MyThread1 t1=new MyThread1(c);
		MyThread2 t2=new MyThread2(c);

		t1.start();
		t2.start();

		try{Thread.sleep(100);}catch(Exception e){System.out.println(e);}
		System.out.println("main : "+c.count);
	}
}
```

```
E:\JAVA\ANU>javac Synch.java

E:\JAVA\ANU>java Synch
main : 2000
```

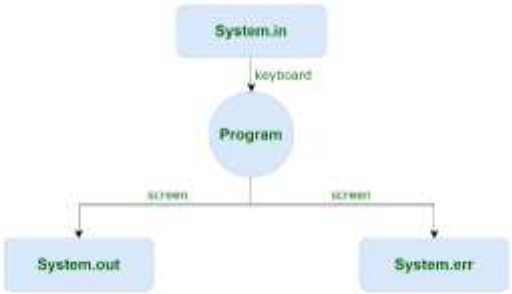
IO Streams

Stream

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

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

1) System.out: standard output stream
2) System.in: standard input stream
3) System.err: standard error stream



There are mainly 2 kinds of streams

OutputStream

Java application uses an output stream to write data to a destination; it may be a file, an array, peripheral device or socket.

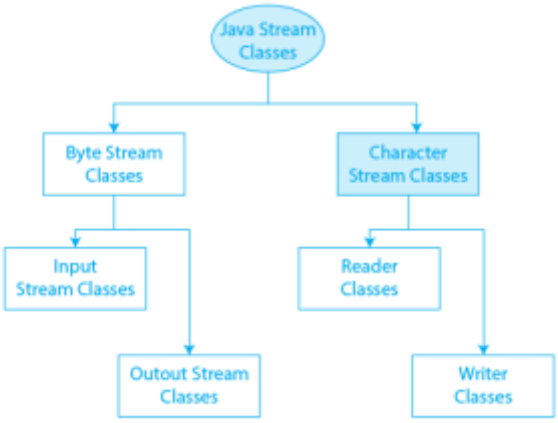
InputStream

Java application uses an input stream to read data from a source; it may be a file, an array, peripheral device or socket.

Depending on the **type of data** java Streams can be divided into two primary classes which can be further divided into other classes

ByteStream: This is used to process data byte by byte (8 bits). Though it has many classes, the FileInputStream and the FileOutputStream are the most popular ones. The FileInputStream is used to read from the source and FileOutputStream is used to write to the destination.

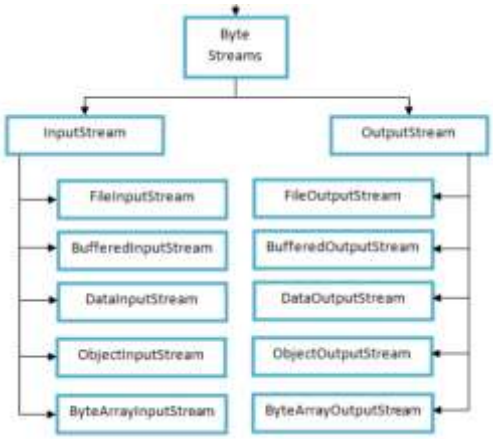
CharacterStream: In Java, characters are stored using Unicode conventions. Character stream automatically allows us to read/write data character by character. Though it has many classes, the FileReader and the FileWriter are the most popular ones. FileReader and FileWriter are character streams used to read from the source file and write to the destination file respectively.



ByteStreams:

Byte streams are designed to deal with raw binary data, which includes all kinds of data, including characters, pictues, audio, and video. These streams are represented through cclasses that cease with the word "InputStream" or "OutputStream" of their names,along with FileInputStream,BufferedInputStream, FileOutputStream and BufferedOutputStream.

Byte streams offer a low-stage interface for studying and writing character bytes or blocks of bytes. They are normally used for coping with non-textual statistics, studying and writing files of their binary form, and running with network sockets. Byte streams don't perform any individual encoding or deciphering. They treat the data as a sequence of bytes and don't interpret it as characters.



Java program on reading data from a source file and writing data to the destination file using FileInputStram and FileOutputStream

```
import java.io.*;
class ReadWriteFIS
{
	public static void main(String args[]) throws IOException,  FileNotFoundException
	{
		FileInputStream in=new FileInputStream("src.txt");
		FileOutputStream ot=new FileOutputStream("dstn.txt");
		int data;
		while((data=in.read()) !=-1)
		{
			ot.write(data);
		}
		System.out.println("Data Saved..!");
		in.close();
		ot.close();
	}
}
```

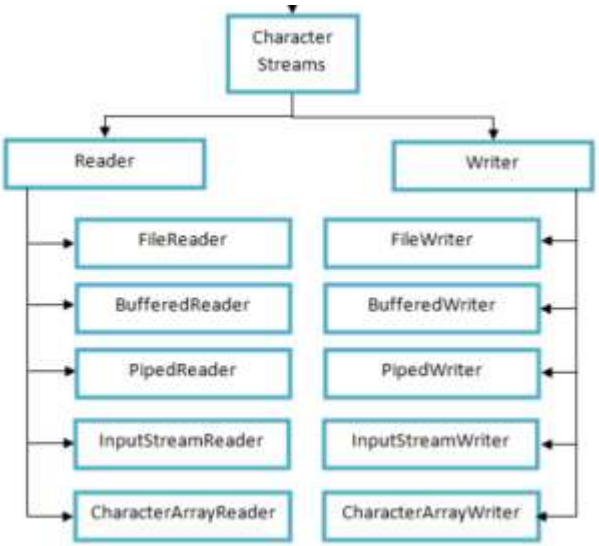
```
E:\JAVA\ANU>javac ReadWriteFIS.java

E:\JAVA\ANU>java ReadWriteFIS
Data Saved..!
```

Character Streams:

Character streams are designed to address character based records, which includes textual records inclusive of letters, digits, symbols, and other characters. These streams are represented by way of training that quit with the phrase "Reader" or "Writer" of their names, inclusive of FileReader, BufferedReader, FileWriter, and BufferedWriter.

Character streams offer a convenient manner to read and write textual content-primarily based information due to the fact they mechanically manage character encoding and decoding. They convert the individual statistics to and from the underlying byte circulation the usage of a particular individual encoding, such as UTF-eight or ASCII.It makes person streams suitable for operating with textual content files, analyzing and writing strings, and processing human-readable statistics.



Java program on reading data from a source file and writing data to the destination file using FileReader and FileWriter

```
import java.io.*;
class ReadWriteChar
{
	public static void main(String args[]) throws IOException,  FileNotFoundException
	{
		FileReader in=new FileReader("srcfile.txt");
		FileWriter ot=new FileWriter("dstnfile.txt");
		int data;
		while((data=in.read()) !=-1)
		{
			ot.write(data);
		}
		System.out.println("Data Saved..!");
		in.close();
		ot.close();
	}
}
```

```
E:\JAVA\ANU>javac ReadWriteChar.java

E:\JAVA\ANU>java ReadWriteChar
Data Saved..!
```

Byte Stream	Character Stream
It operates on raw binary data and data is processed byte by byte	It operates on text data and is processed character by character
Suitable for processing non-textual data such as images, videos, etc.	Suitable for processing textual data such as documents, HTML, etc.
Input and Output operations are performed using InputStream and OutputStream classes	Input and Output operations are performed using Reader and Writer classes
Suitable for low-level input and output operations	Suitable for high-level input and output operations
Examples include FileInputStream and FileOutputStream	Examples include FileReader and FileWriter

Console Class

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 java.io.Console class is attached with system console internally. The Console class is introduced since 1.5.

Java Console class methods

Method	Description
Reader reader()	It is used to retrieve the reader object associated with the console
String readLine()	It is used to read a single line of text from the console.
String readLine(String fmt, Object... args)	It provides a formatted prompt then reads the single line of text from the console.
char[] readPassword()	It is used to read password that is not being displayed on the console.
char[] readPassword(String fmt, Object... args)	It provides a formatted prompt then reads the password that is not being displayed on the console.
Console format(String fmt, Object... args)	It is used to write a formatted string to the console output stream.
Console printf(String format, Object... args)	It is used to write a string to the console output stream.
PrintWriter writer()	It is used to retrieve the PrintWriter object associated with the console.
void flush()	It is used to flushes the console.

Reading Console Input and Writing Console Output

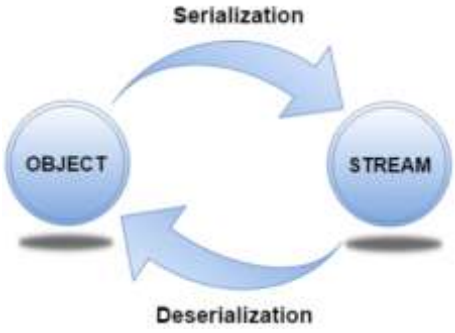
```
import java.io.Console;
class StreamConsole
{
        public static void main(String args[])
        {
                Console c=System.console();
                System.out.println("Enter your name: ");
                String n=c.readLine();
                System.out.println("Welcome "+n);
   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

```
E:\JAVA>javac StreamConsole.java

E:\JAVA>java StreamConsole
Enter your name:
maxi
Welcome maxi
```

Serializaation

- Serialization is a mechanism of converting the state of an object into a byte stream. Deserialization is the reverse process where the byte stream is used to recreate the actual Java object in memory. This mechanism is used to persist the object.
- Serialization and deserialization are crucial for saving and restoring the state of objects in Java.
- The byte stream created is platform independent. So, the object serialized on one platform can be deserialized on a different platform. To make a Java object serializable we implement the java.io.Serializable interface. The ObjectOutputStream class contains writeObject() method for serializing an Object.
- The ObjectInputStream class contains readObject() method for deserializing an object.



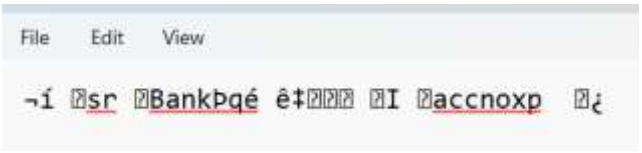
java.io.Serializable interface

Serializable is a marker interface (has no data member and method). It is used to "mark" Java classes so that the objects of these classes may get a certain capability.
The Serializable interface must be implemented by the class whose object needs to be persisted.

```
import java.io.*;
class SerialDemo
{
        public static void main(String args[]) throws IOException,  FileNotFoundException, ClassNotFoundException
        {
                Bank b=new Bank();
                b.accno=5823;
                FileOutputStream fos=new FileOutputStream("filedemo.txt");
                ObjectOutputStream os=new ObjectOutputStream(fos);
                os.writeObject(b);
        }
}
class Bank implements Serializable
{
        int accno;
}
```

```
E:\JAVA\ANU>javac SerialDemo.java

E:\JAVA\ANU>java SerialDemo
```



Deserialization

- Deserialization is the process of reconstructing the object from the serialized state. It is the reverse operation of serialization. Let's see an example where we are reading the data from a deserialized object.
- Deserialization is the process of reconstructing the object from the serialized state. It is the reverse operation of serialization. Let's see an example where we are reading the data from a deserialized object.

```
import java.io.*;
class DeSerialDemo
{
	public static void main(String args[]) throws IOException,  FileNotFoundException, ClassNotFoundException
	{
		FileInputStream fis=new FileInputStream("filedemo.txt");
		ObjectInputStream is=new ObjectInputStream(fis);
		Bank b1=(Bank)is.readObject();
		System.out.println(b1.accno);


	}
}
class Bank implements Serializable
{
	int accno;
}
```

```
E:\JAVA\ANU>javac DeSerialDemo.java

E:\JAVA\ANU>java DeSerialDemo
5823
```