Programming in Java

Unit III

Unit III

Multithreaded Programming: Thread Class – Runnable Interface – Synchronization – Using Synchronized methods – Using Synchronized statement – Interthread Communication – Deadlock.

I/O Streams: Concepts of streams – Stream Classes – Byte and Character stream – Reading console Input and Writing Console output – File Handling

Multithreaded Programming

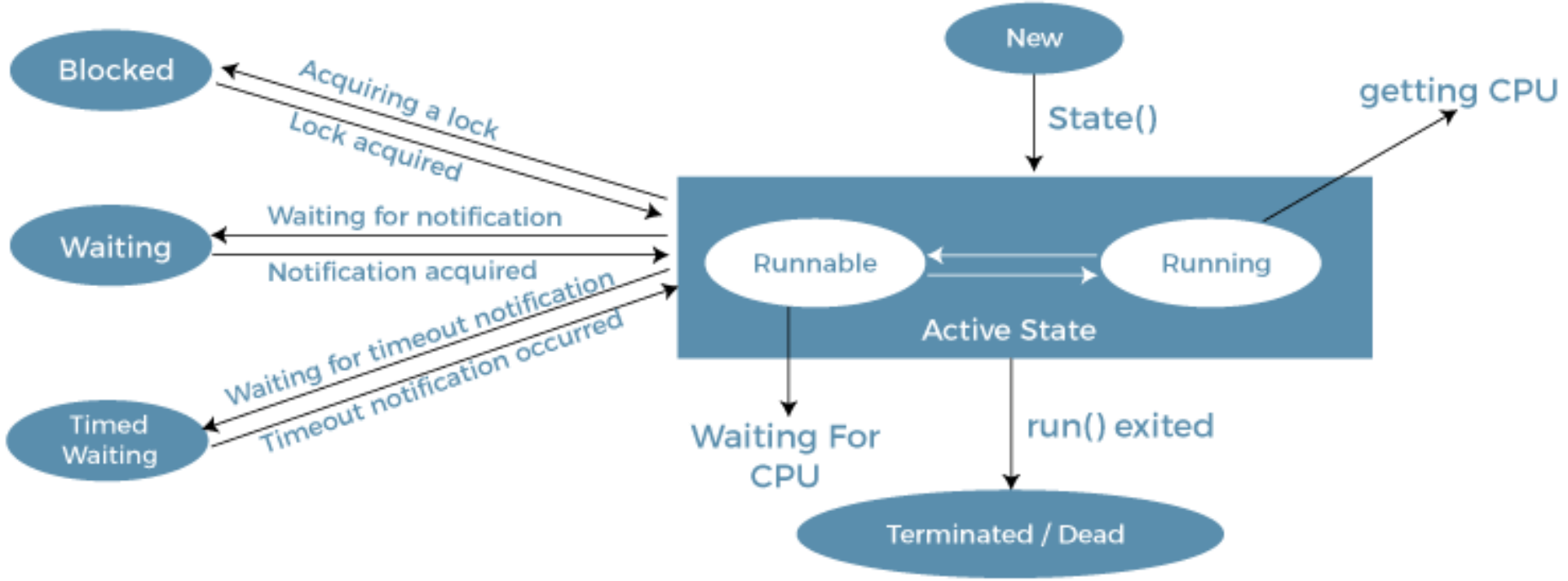
- Multithreading in Java is a process of executing multiple threads simultaneously.
- A thread is a lightweight sub-process, the smallest unit of processing.
- Multiprocessing and multithreading, both are used to achieve multitasking.
- However, we use multithreading than multiprocessing because threads 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.

Multithreading Vs Multiprocessing

- Multitasking is a process of executing multiple tasks simultaneously.
- It is used to utilize the CPU.
- Multitasking can be achieved in two ways:
  - Process-based multitasking – Multiprocessing
  - Thread-based multitasking - Multithreading

Multiprocessing	Multithreading
Each process has an address in memory. In other words, each process allocates a separate memory area.	Threads share the same address space.
A process is heavyweight.	A thread is lightweight.
Cost of communication between the process is high.	Cost of communication between the thread is low.
Switching from one process to another requires some time for saving and loading registers, memory maps, updating lists, etc.	

Life Cycle of Thread



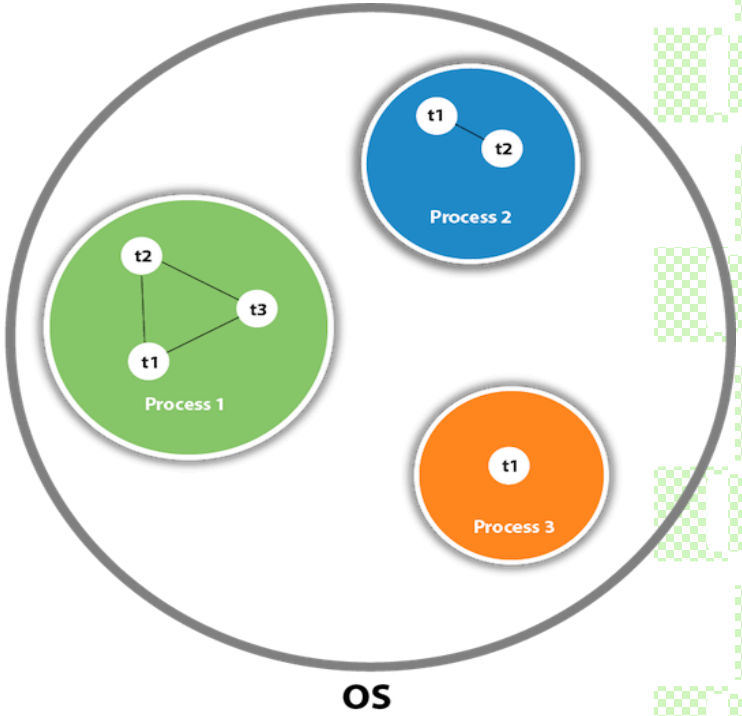
Life Cycle of a Thread

Life Cycle of a Thread

State	Description	Implementing Thread States
New	When a new thread is created, it is in the new state. The thread has not yet started to run when the thread is in this state. When a thread lies in the new state, its code is yet to be run and hasn't started to execute.	public static final Thread.State NEW
Runnable	The thread is ready for execution and is waiting for the availability of the processor. i.e in QUEUE manner. If all the thread have equal priority, it will be executed in round robin fashion i.e., FCFS manner	public static final Thread.State RUNNABLE
Blocked	The thread will be in blocked state when it is trying to acquire a lock but currently the lock is acquired by the other thread. The thread will move from the blocked state to runnable state when it acquires the lock	public static final Thread.State BLOCKED
Waiting	The thread will be in waiting state when it calls wait() method or join() method. It will move to the runnable state when other thread will notify or that thread will be terminated.	public static final Thread.State WAITING
Timed waiting	A thread lies in a timed waiting state when it calls a method with a time-out parameter. A thread lies in this state until the timeout is completed or until a notification is received.	public static final Thread.State TIMED_WAITING
Terminated	A thread terminates because of either of the following reasons: • Because it exits normally. This happens when the code of the thread has been entirely executed by the program. • Because there occurred some unusual erroneous event, like a segmentation fault or an unhandled exception.	public static final Thread.State TERMINATED

Thread Class

- 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.
- Constructors of Thread Class
  - Thread()
  - Thread(String name)
  - Thread(Runnable r)
  - Thread(Runnable r, String name)



Creating a Thread

- There are the following two ways to create a thread:
  - By Extending Thread Class
  - By Implementing Runnable Interface

1. By Extending Thread

Thread class:

- Thread class provide constructors and methods to create and perform operations on a thread.
- Thread class extends Object class and implements Runnable interface.

Commonly used Constructors of Thread class:

- Thread()
- Thread(String name)
- Thread(Runnable r)
- Thread(Runnable r, String name)

1. By Extending Thread

```
class Multi extends Thread                  //Extending thread class
{
public void run()                           // run() method declared
{
System.out.println("thread is running...");
}
public static void main(String args[])
{
 Multi t1=new Multi();                      //object initiated
t1.start();                                         // run() method called through start()
}
}
```

Output: thread is running…

2. By implementing Runnable interface

- Define a class that implements Runnable interface.
- The Runnable interface has only one method, run(), that is to be defined in the method with the code to be executed by the thread.

2. By implementing Runnable interface

```
class Multi3 implements Runnable   //Implementing Runnable interface
{
public void run()
{
System.out.println("thread is running...");
}
 public static void main(String args[])
{
Multi3 m1=new Multi3();                    // object initiated for class
Thread t1 =new Thread(m1);                 // object initiated for thread
t1.start();
}
 }
```

Output: thread is running…

Methods of Thread class:

Methods	Description
public void run()	used to perform action for a thread.
public void start()	starts the execution of the thread.JVM calls the run() method on the thread.
public void sleep(long miliseconds)	Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
public void join()	waits for a thread to die.
public void join(long miliseconds)	waits for a thread to die for the specified miliseconds.
public int getPriority()	returns the priority of the thread.
public int setPriority(int priority)	changes the priority of the thread.
public String getName()	returns the name of the thread.
public void setName(String name)	changes the name of the thread.
public Thread currentThread()	returns the reference of currently executing thread.
public int getId()	returns the id of the thread.

Methods of Thread class:

Methods	Description
public Thread.State getState()	returns the state of the thread.
public boolean isAlive():	tests if the thread is alive.
public void yield():	causes the currently executing thread object to temporarily pause and allow other threads to execute.
public void suspend():	used to suspend the thread(depricated).
public void resume():	used to resume the suspended thread(depricated).
public void stop():	used to stop the thread(depricated).
public boolean isDaemon():	tests if the thread is a daemon thread.
public void setDaemon(boolean b):	marks the thread as daemon or user thread.
public void interrupt():	interrupts the thread.
public boolean isInterrupted():	tests if the thread has been interrupted.
public static boolean interrupted():	tests if the current thread has been interrupted.

Static Methods in Thread Class

Syntax of the method	Purpose
static Thread currentThread()	To return a reference to the current thread
static void sleep(long m)	To cause the current thread to wait for m milliseconds
static sleep(long m , long n)	To cause the current thread to wait for m milliseconds plus n nanoseconds

Instance methods in Thread Class

Methods	Purpose
void start()	To start the specific thread
void run()	To encapsulate the functionality of a thread
boolean isAlive()	To return true if thread has been started and not yet died
void setName(String s)	To set the name of the thread to ‘s’
String getName()	To return the name of the specified thread
void setPriority(int p)	To Set the priority of the thread to ‘p’
int getPriority()	To return the priority of the thread

Priority of a Thread

- 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 (known as preemptive scheduling).
- But it is not guaranteed because it depends on JVM specification that which scheduling it chooses.
- 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

- **public final int getPriority():**
  - The java.lang.Thread.getPriority() method returns the priority of the given thread.
- **public final void 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.

```
// Importing the required classes
import java.lang.*;
public class ThreadPriorityExample extends Thread
{
                public void run()
        {
                // the print statement
                System.out.println("Inside the run() method");
        }
        // the main method
        public static void main(String argvs[])
        {
                // Creating threads with the help of ThreadPriorityExample class
                ThreadPriorityExample th1 = new ThreadPriorityExample();
                ThreadPriorityExample th2 = new ThreadPriorityExample();
                ThreadPriorityExample th3 = new ThreadPriorityExample();
                // We did not mention the priority of the thread.  Therefore, the priorities of the thread is 5, the
                default value
        // 1st Thread  Displaying the priority of the thread using the getPriority() method
                System.out.println("Priority of the thread th1 is : " + th1.getPriority());
        // 2nd Thread Display the priority of the thread
                System.out.println("Priority of the thread th2 is : " + th2.getPriority());
        // 3rd Thread  Display the priority of the thread
                System.out.println("Priority of the thread th2 is : " + th2.getPriority());
```

```
        // Setting priorities of above threads by passing integer arguments
        th1.setPriority(6);
        th2.setPriority(3);
        th3.setPriority(9);
        // 6
        System.out.println("Priority of the thread th1 is : " + th1.getPriority());
        // 3
        System.out.println("Priority of the thread th2 is : " + th2.getPriority());
        // 9
        System.out.println("Priority of the thread th3 is : " + th3.getPriority());
        // Main thread Displaying name of the currently executing thread
        System.out.println("Currently Executing The Thread : " + Thread.currentThread().getName()
);
        System.out.println("Priority of the main thread is : " + Thread.currentThread().getPriority());
        // Priority of the main thread is 10 now
        Thread.currentThread().setPriority(10);
        System.out.println("Priority of the main thread is : " + Thread.currentThread().getPriority());
        }
}
```

Output:

```
Priority of the thread th1 is : 5
Priority of the thread th2 is : 5
Priority of the thread th2 is : 5
Priority of the thread th1 is : 6
Priority of the thread th2 is : 3
Priority of the thread th3 is : 9
Currently Executing The Thread : main
Priority of the main thread is : 5
Priority of the main thread is : 10
```

Advantages of 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.

Synchronization

- Synchronization in Java is a critical concept in concurrent programming that ensures multiple threads can interact with shared resources safely.
- In a nutshell, synchronization prevents race conditions, where the outcome of operations depends on the timing of thread execution.
- It is the capability to control the access of multiple threads to any shared resource.
- Synchronization is a better option where we want to allow only one thread to access the shared resource.

Using synchronization method and statements

- The access to a method shall be synchronized by specifying the synchronized keyword as a modifier in the method declaration.
- When a thread starts executing a synchronized instance meyhod, it automatically gets a logical lock on the object that contains the synchronized method.
- This lock will remain till the thread is executing in that synchronized method.
- When the lock is present, no other thread will be allowed entry into that object.
- The lock will automatically released, when the thread completes its execution.

Example Program

```
class Account{
        private int balance=0;
        synchronized void credit(int amount){
                balance=balance+amount;
        }
        void displayBalance(){
                System.out.println(balance);
        }
}
class AccountHolder extends Thread{
        Account account;
        AccountHolder(Account account){
                this.account=account;
        }
        void run(){
                for(int j=0;j<5000;j++)
                        account.credit(100);
        }
}
```

```
class CreditDemo{
        static int numAccountHolders=5;
        public static void main(String args[])
        {
        Account account=new Account();
        AccountHolder accountholders[]=new AccountHolder[numAccountHolders];
        for(int k=0;k<numAccountHolders;k++)
        {
            accountholders[k]=new AccountHolders(account);
            accountholders[k].start();
        }
        for(int i=0;i<numAccountHolders;i++)
        {
        try
        {
                        accountholders[i].join();
        }
        catch(Exception e){}
        account.displayBalance();
        }
        }
        }
```

Output:

```
500000
2500000
2500000
2500000
2500000
```

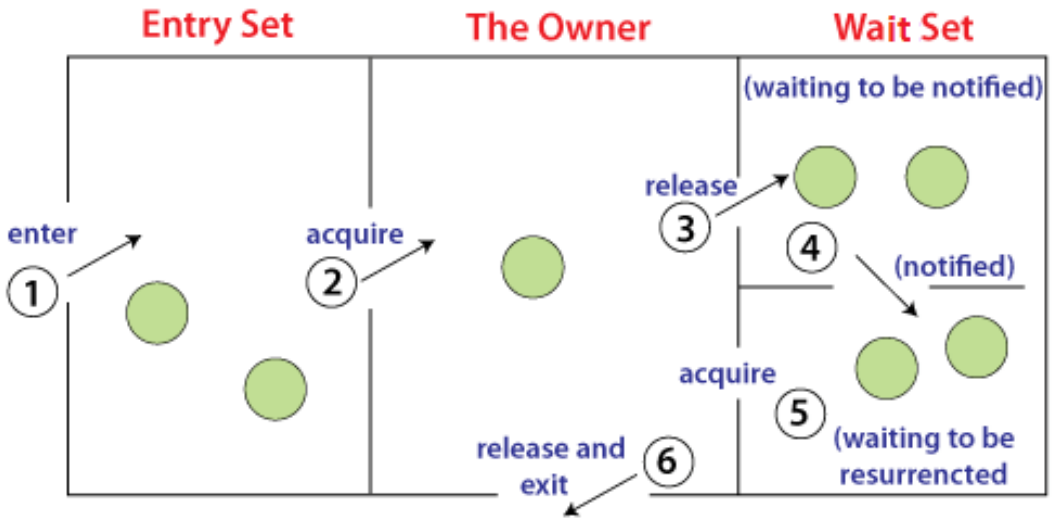
Explanation

- In the above program, the Account class is defined with synchronized credit() method and an ordinary method named displayBalance(). another class AccountHolder is defined with constructor and run() method.
- With in the main() method, single instance for Account class is created and three instances are created for AccountHolder class. i.e., three different threads are created and started.
- These three threads access the common method credit() that is declared with synchronized keyword.
- The main thread will wait until all these threads get executed with the help of join() method.
- Once these three threads complete their work, the final balance is retrieved and displayed.

Inter-thread Communication

- Inter-thread communication or Co-operation is all about allowing synchronized threads to communicate with each other.
- Cooperation (Inter-thread communication) is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed.It is implemented by following methods of Object class:
  - wait()
  - notify()
  - notifyAll()

Method	Syntax	Description
wait()	public final void wait()throws InterruptedException	It waits until object is notified.
	public final void wait(long timeout)throws InterruptedException	It waits for the specified amount of time.
notify()	public final void notify()	The notify() method wakes up a single thread that is waiting on this object's monitor.
notifyAll()	public final void notifyAll()	Wakes up all threads that are waiting on this object's monitor.



Understanding the process of inter-thread communication

1. Threads enter to acquire lock.
2. Lock is acquired by a thread.
3. Now thread goes to waiting state if you call wait() method on the object. Otherwise it releases the lock and exits.
4. If you call notify() or notifyAll() method, thread moves to the notified state (runnable state).
5. Now thread is available to acquire lock.
6. After completion of the task, thread releases the lock and exits the monitor state of the object.

Example Program

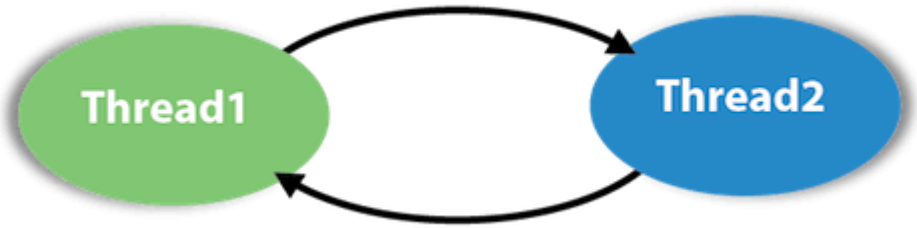
```
class Customer{
int amount=10000;
        synchronized void withdraw(int amount){
                System.out.println("going to withdraw...");
                if(this.amount<amount){
                System.out.println("Less balance; waiting for deposit...");
                try{wait();}catch(Exception e){}
                }
                this.amount-=amount;
                System.out.println("withdraw completed...");
        }
        synchronized void deposit(int amount){
                System.out.println("going to deposit...");
                this.amount+=amount;
                System.out.println("deposit completed... ");
                notify();
        }
}
class Test{
        public static void main(String args[]){
                final Customer c=new Customer();
                new Thread(){
                public void run(){c.withdraw(15000);}
                }.start();
                new Thread(){
                public void run(){c.deposit(10000);}
                }.start();
}}
```

Output:

going to deposit...
deposit completed...
going to withdraw...
withdraw completed...

Deadlock

- Deadlock in Java is a part of multithreading.
- Deadlock can occur in a situation when a thread is waiting for an object lock, that is acquired by another thread and second thread is waiting for an object lock that is acquired by first thread.
- Since, both threads are waiting for each other to release the lock, the condition is called deadlock.



Deadlock possibilities

- A deadlock may also include more than two threads. The reason is that it can be difficult to detect a deadlock.
- Here is an example in which four threads have deadlocked:
  - Thread 1 locks A, waits for B
  - Thread 2 locks B, waits for C
  - Thread 3 locks C, waits for D
  - Thread 4 locks D, waits for A
  - Thread 1 waits for thread 2, thread 2 waits for thread 3, thread 3 waits for thread 4, and thread 4 waits for thread 1.

Example Program

```
public class DeadlockSolved {
   public static void main(String ar[]) {
      DeadlockSolved test = new DeadlockSolved();
      final resource1 a = test.new resource1();
      final resource2 b = test.new resource2();
Runnable b1 = new Runnable() {     // Thread-1
   public void run() {
      synchronized (b) {
         try {    // Adding delay so that both threads can start trying to lock resources
            Thread.sleep(100);
         } catch (InterruptedException e) {
            e.printStackTrace();
         }  // Thread-1 have resource1 but need resource2 also
         synchronized (a) {
            System.out.println("In block 1");
         }   }   } };
Runnable b2 = new Runnable() { // Thread-2
   public void run() {
      synchronized (b) {  // Thread-2 have resource2 but need resource1 also
         synchronized (a) {
            System.out.println("In block 2");
         } } } };
      new Thread(b1).start();
      new Thread(b2).start();
   }
```

```
private class resource1 {  // resource1
   private int i = 10;
   public int getI() {
      return i;
   }
   public void setI(int i) {
      this.i = i;
   }  }
private class resource2 {    // resource2
   private int i = 20;
   public int getI() {
      return i;
   }
   public void setI(int i) {
      this.i = i;
   }  }  }
```

Output:

In block 1

In block 2

How to avoid deadlock?

- Deadlocks cannot be completely resolved. But we can avoid them by following basic rules mentioned below:

1. **Avoid Nested Locks**: We must avoid giving locks to multiple threads, this is the main reason for a deadlock condition. It normally happens when you give locks to multiple threads.
2. **Avoid Unnecessary Locks**: The locks should be given to the important threads. Giving locks to the unnecessary threads that cause the deadlock condition.
3. **Using Thread Join**: A deadlock usually happens when one thread is waiting for the other to finish. In this case, we can use join with a maximum time that a thread will take.

I/O Streams

- 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.
- Java I/O revolves around two primary concepts: streams and readers/writers

Stream

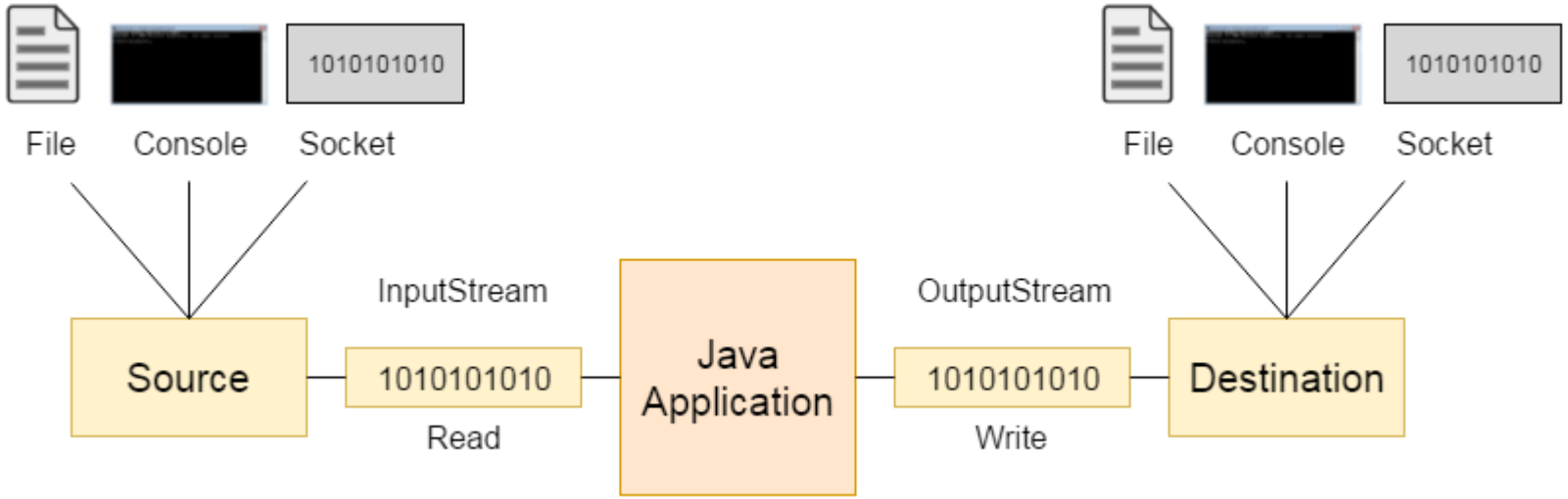
- A stream is a sequence of data. In Java, a stream is composed of bytes. It's called a stream because it is like a stream of water that continues to flow.
- In Java, 3 streams are created for us automatically. All these streams are attached with the console.
  1) System.out: standard output stream
  2) System.in: standard input stream
  3) System.err: standard error stream
- Depending upon the data a stream holds, it can be classified into:
  - Byte Stream
  - Character Stream

Byte Stream

- Byte stream is used to read and write a single byte (8 bits) of data.
- All byte stream classes are derived from base abstract classes called InputStream and OutputStream.

OutputStream Vs. InputStream

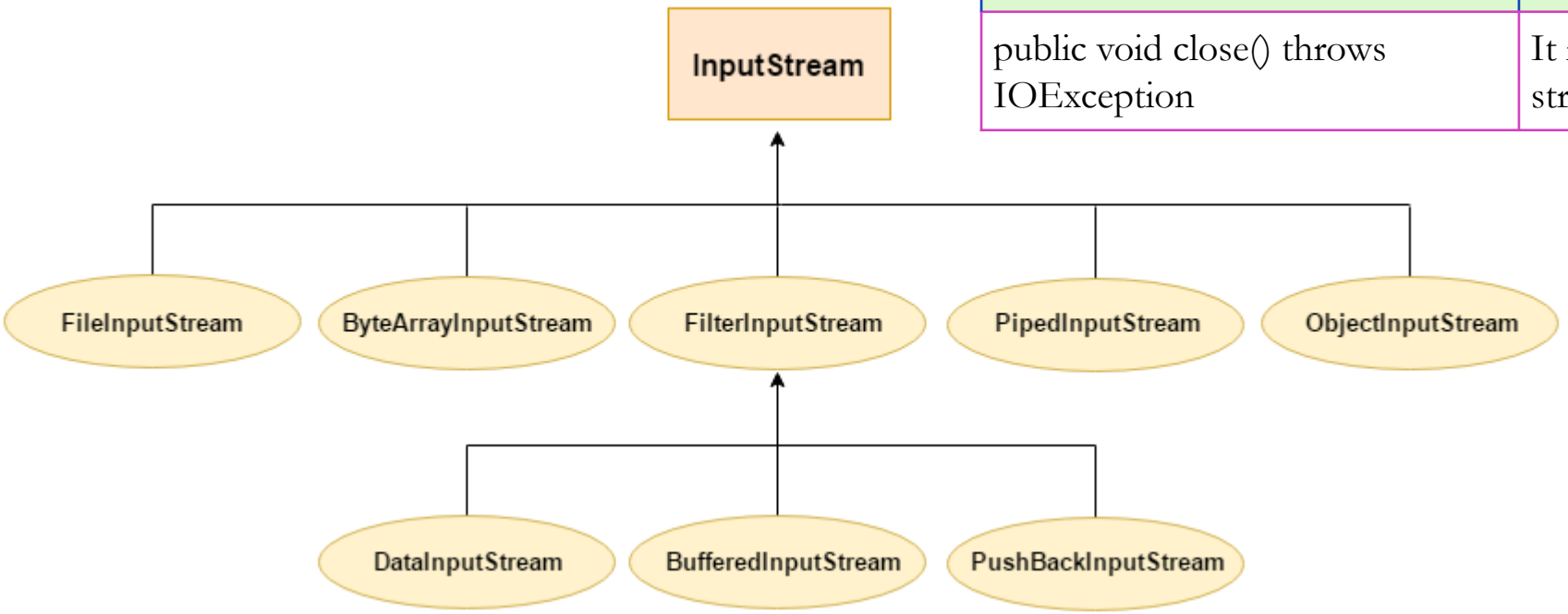
- OutputStream
  - Java application uses an output stream to write data to a destination; it may be a file, an array, peripheral device or socket.
- InputStream
  - Java application uses an input stream to read data from a source; it may be a file, an array, peripheral device or socket.



InputStream Class

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

Method	Description
public abstract int read() throws IOException	It reads the next byte of data from the input stream. It returns -1 at the end of the file.
public int available() throws IOException	It returns an estimate of the number of bytes that can be read from the current input stream.
public void close() throws IOException	It is used to close the current input stream.



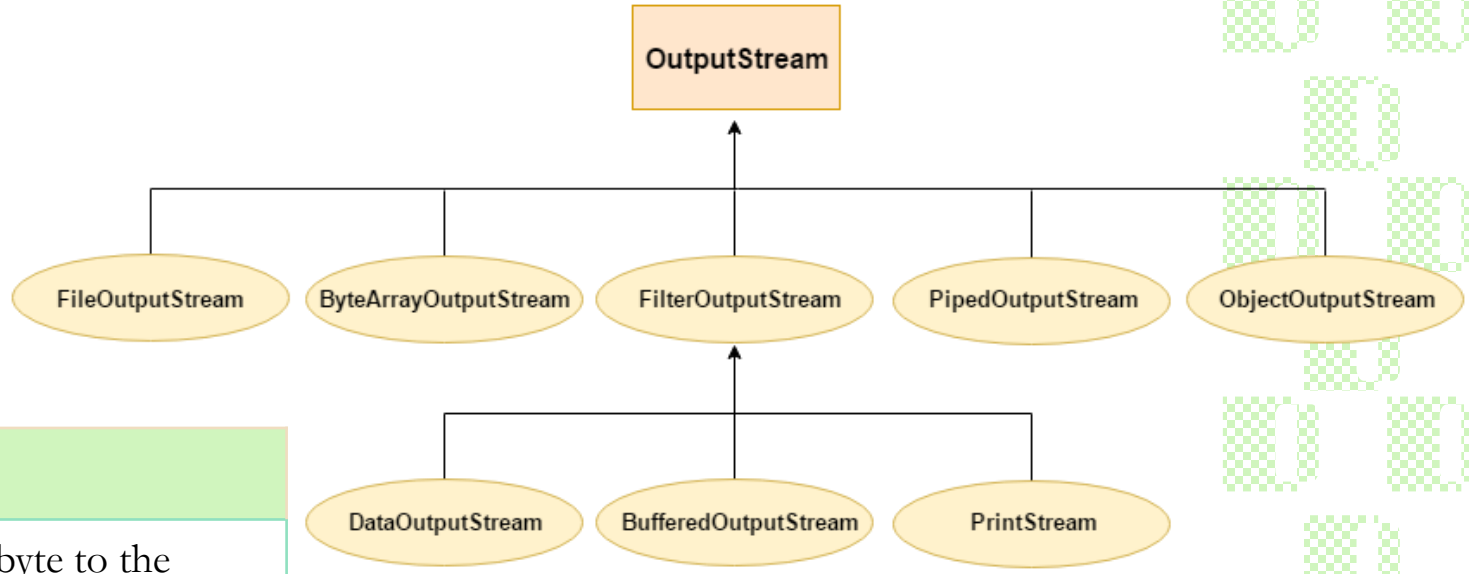
Example Program

```
import java.io.FileInputStream;
import java.io.InputStream;
class Main {
  public static void main(String args[]) {
    byte[] array = new byte[100];
    try {
      InputStream input = new FileInputStream("input.txt");
      System.out.println("Available bytes in the file: " + input.available());
      // Read byte from the input stream
      input.read(array);
      System.out.println("Data read from the file: ");
      // Convert byte array into string
      String data = new String(array);
      System.out.println(data);
      // Close the input stream
      input.close();
    } catch (Exception e) {
      e.getStackTrace();
    }
  }
}
```

Output:
Available bytes in the file: 39
Data read from the file:
This is a line of text inside the file

OutputStream Class

OutputStream class is an abstract class.
It is the superclass of all classes representing an output stream of bytes.
An output stream accepts output bytes and sends them to some sink.



Method	Description
public void write(int) throws IOException	It is used to write a byte to the current output stream.
public void write(byte[])throws IOException	It is used to write an array of byte to the current output stream.
public void flush() throws IOException	It flushes the current output stream.
public void close() throws IOException	It is used to close the current output stream.

Example Program

```
import java.io.FileOutputStream;
import java.io.OutputStream;
public class Main {
    public static void main(String args[]) {
        String data = "This is a line of text inside the file.";
        try {
            OutputStream out = new FileOutputStream("output.txt");
            // Converts the string into bytes
            byte[] dataBytes = data.getBytes();
            // Writes data to the output stream
            out.write(dataBytes);
            System.out.println("Data is written to the file.");
            // Closes the output stream
            out.close();
        }
        catch (Exception e) {
            e.getStackTrace();
        }
    }
}
```

Character Stream

- Character stream is used to read and write a single character of data.
- All the character stream classes are derived from base abstract classes Reader and Writer.

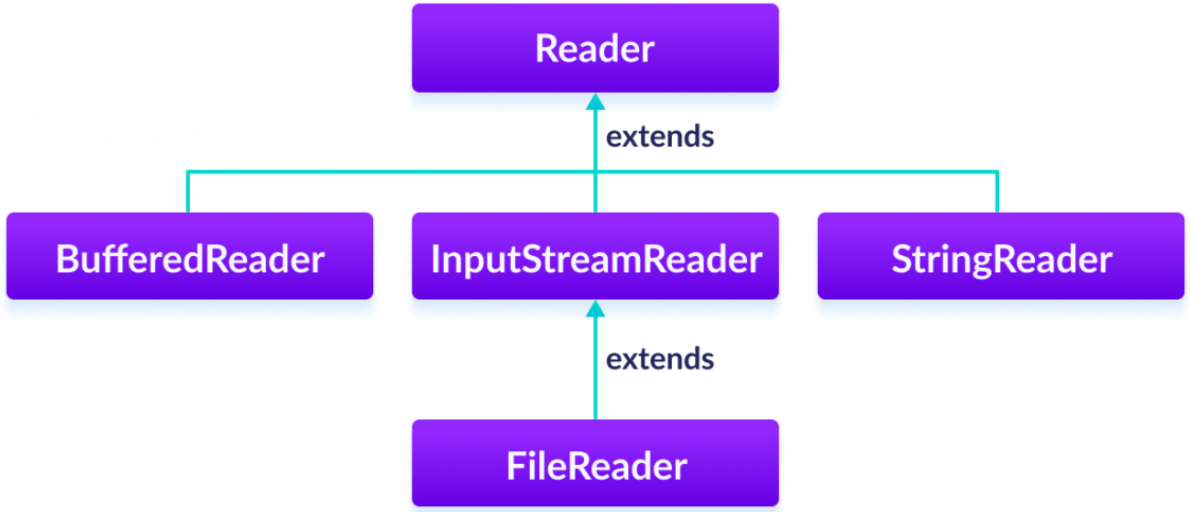
Reader Class

The Reader class of the java.io package is an abstract superclass that represents a stream of characters.
Since Reader is an abstract class, it is not useful by itself. However, its subclasses can be used to read data.

Subclasses of Reader

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

- BufferedReader
- InputStreamReader
- FileReader
- StringReader



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

Example Program

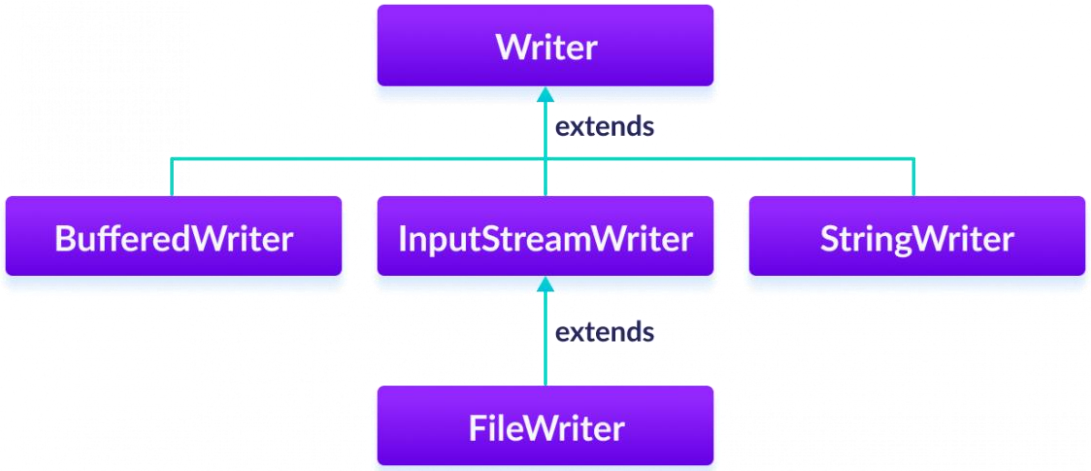
```
import java.io.Reader;
import java.io.FileReader;
class Main {
  public static void main(String[] args) {
    // Creates an array of character
    char[] array = new char[100];
    try {
      // Creates a reader using the FileReader
      Reader input = new FileReader("input.txt");
      // Checks if reader is ready
      System.out.println("Is there data in the stream?  " + input.ready());
      // Reads characters
      input.read(array);
      System.out.println("Data in the stream:");
      System.out.println(array);
      // Closes the reader
      input.close();
    }
    catch(Exception e) {
      e.getStackTrace();
    }
  }
}
```

Output:

```
Is there data in the stream?  true
Data in the stream:
This is a line of text inside the file.
```

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:
- In order to use the functionality of the Writer, we can use its subclasses. Some of them are:
  - BufferedWriter
  - OutputStreamWriter
  - FileWriter
  - StringWriter



Methods of Writer class

- 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

Example Program

```
import java.io.FileWriter;
import java.io.Writer;
public class Main {
   public static void main(String args[]) {
      String data = "This is the data in the output file";
      try {
         // Creates a Writer using FileWriter
         Writer output = new FileWriter("output.txt");
         // Writes string to the file
         output.write(data);
         // Closes the writer
         output.close();
      }
      catch (Exception e) {
         e.getStackTrace();
      }
   }
}
```

Reading console Input

- By default, to read from system console, we can use the Console class. This class provides methods to access the character-based console, if any, associated with the current Java process. To get access to Console, call the method System.console().
- Console gives three ways to read the input:
  - String readLine() – reads a single line of text from the console.
  - char[] readPassword() – reads a password or encrypted text from the console with echoing disabled
  - Reader reader() – retrieves the Reader object associated with this console. This reader is supposed to be used by sophisticated applications. For example, Scanner object which utilizes the rich parsing/scanning functionality on top of the underlying Reader.

Reading Input with *readLine()*

```
Console console =
System.console();
if(console == null) {
System.out.println("Console is not
available to current JVM process");
return; }
String userName =
console.readLine("Enter the
username: ");
System.out.println("Entered
username: " + userName);
```

```
Enter the username: lokesh
Entered username: lokesh
```

Reading Input with *readPassword()*

```
Console console =
System.console();
if(console == null) {
System.out.println("Console is not
available to current JVM process");
return; }
char[] password =
console.readPassword("Enter the
password: ");
System.out.println("Entered
password: " + new
String(password));
```

```
Enter the password: //input will not
visible in the console
Entered password: passphrase
```

Read Input with *reader()*

```
Console console =
System.console();
if(console == null) {
System.out.println("Console is not
available to current JVM process");
return; }
Reader consoleReader =
console.reader();
Scanner scanner = new
Scanner(consoleReader);
System.out.println("Enter age:");
int age = scanner.nextInt();
System.out.println("Entered age: "
+ age);
scanner.close();
```

```
Enter age:
12
Entered age: 12
```

Writing Console output

- The easiest way to write the output data to console is System.out.println() statements. Still, we can use printf() methods to write formatted text to console

1. **Writing with *System.out.println***
    - System.out.println("Hello, world!");

    Program output
    - Hello, world!

2. **Writing with *printf()***

- The printf(String format, Object... args) method takes an output string and multiple parameters which are substituted in the given string to produce the formatted output content. This formatted output is written in the console.

    String name = "Lokesh";

    int age = 38;

    console.printf("My name is %s and my age is %d", name, age);

- Program output

    My name is Lokesh and my age is 38

File Handling

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

```
// creates an object of File using the path
File file = new File(String pathName);
```

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

Creating a new File

- 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.

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

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.
```

Reading a file

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

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

Output:
Data in the file:
This is a line of text inside the file.

Writing to a file

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

```
import java.io.FileWriter;
 class Main {
   public static void main(String args[]) {
     String data = "This is the data in the output file";
     try {
       FileWriter output = new FileWriter("output.txt"); // Creates a Writer using FileWriter
       output.write(data); // Writes string to the file
       System.out.println("Data is written to the file.");
       output.close(); // Closes the writer
     }
     catch (Exception e) {
       e.getStackTrace();
     }
   }
}
```

Output:
Data is written to the file.

Deleting a file

- We can use the delete() method of the File class to delete the specified file or directory. It returns
  - true if the file is deleted.
  - false if the file does not exist.

```
import java.io.File;
class Main {
  public static void main(String[] args) {
    // creates a file object
    File file = new File("file.txt");
    // deletes the file
    boolean value = file.delete();
    if(value) {
      System.out.println("The File is deleted.");
    }
    else {
      System.out.println("The File is not deleted.");
    }
  }
}
```

Output:
The File is deleted.