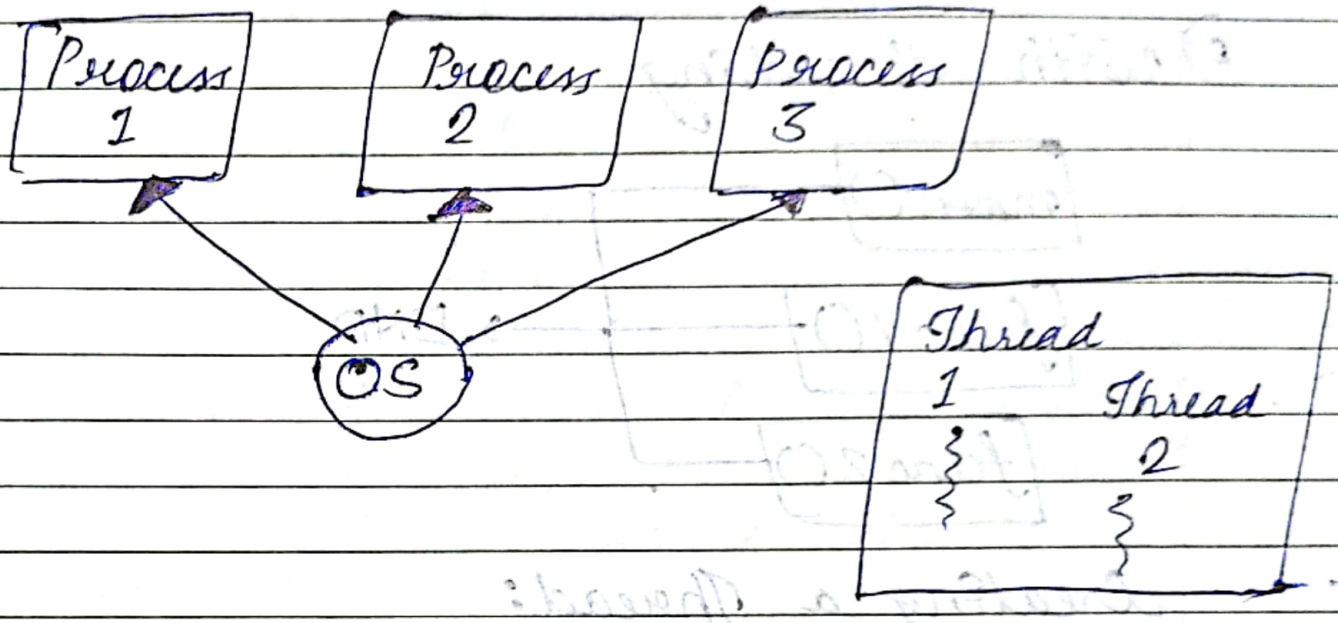
Multithreadings in Java

Multiprocessing & multithreading both are used to achieve multitasking



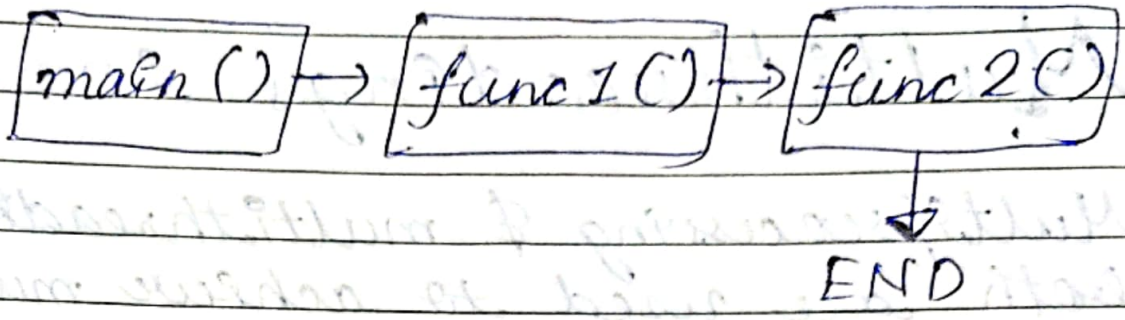
In a nutshell:

- Threads use shared memory area
- Threads ⇒ Faster context switching
- A thread is light-weight whereas a process is heavyweight

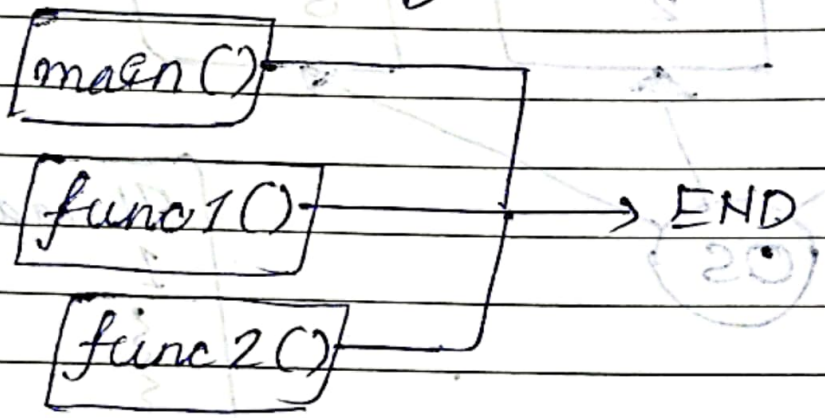
For eg: A word processor can have one thread running in foreground as an editor & another in the background auto saving the document.

Flow of control in Java:

1) Without threading



2) With threading



Creating a Thread:

There are two ways to create a thread in Java:

- By extending Thread class
- By implementing Runnable interface

Multithreading in Java:

- Used to maximize the CPU utilization.
- We don't want our CPU to be in a free state, for eg; Func1() comes into the memory & demands any i/p/o/p process. The CPU will need to wait for until Func1() to complete its i/p/o/p

operation on such a condition.
But, while Func1() completes its I/O operation, the CPU is free & not executing any thread. So, the ~~effecien~~ efficiency of the CPU is decreased in the absence of multithreading.

→ In the case of multithreading, if a thread demands any I/O operation, then the CPU will ~~need~~ let the thread perform its I/O operation, but it will start the execution of a new thread parallelly. So, in this case, two threads are executing at the same time.

~~Extending~~
Lets see how to create a thread by extending the thread class:

```
class MyThread extends Thread{
    @Override

    public void run(){
        // code that we want to get
    }// executed on running thread

}
```

→ The code you want to execute on the thread's execution goes inside the run() method.

```
class MyThread1 extends Thread {
    @Override

public void run() {
    int i=0;
    while (i<40000) {
    System.out.println("My cooking Thread is
                                Running");
    System.out.println("I am happy");
        i++;
    }
  }
}

public class X {
    p s v m ( ) {
    MyThread t1 = new MyThread();
      t1.start();
    }
}
```

In order to execute the thread, the start() method is used. start() is called on the object of the MyThread class. It automatically calls the run() method, & a new stack is provided to the thread. So, that's how you easily create threads by extending the thread class in Java.

```
class MyThread2 extends Thread{
  @Override
  public void run(){
      int i=0;
    while(i<40000){
      sout("I am chatting");
    sout("I am sad");
      i++;
    }
  }
}

public class cwh{
  p s v m( ){
  MyThread1 t1=new MyThread1();
  MyThread2 t2= " " " 2();
    t1.start();
    t2.start();
  }
}
```

O/P

My cooking thread . .

- - - -

I am chatting - -

Steps to create a Java thread using runnable Interface

1) Create a class & implement the Runnable Interface by using implements keyword
2) Override the run method inside the implementer class
3) Create an object of the implementer class in the main() method.
4) Instantiate the Thread class & pass the object to the thread ~~class~~ constructor
5) Call start() on the thread. It will call the run() method.

E.g.

```
class t1 implements Runnable {
    @Override
    public void run() {
        System.out.println("Thread is running");
    }
}

public class Classname {
    p s vm ( ) {
        t1 obj1 = new t1();
        Thread t = new Thread(obj1);
        t.start();
    }
}
```

o/p

1) Class t1 implements the runnable interface
2) Overriding of the run method is done inside the t1 class.
3) In the main method, obj1, an object of the t1 class, is created.
4) The constructor of the Thread class accepts the ~~ta~~ runnable instance as an argument, so obj1 is passed to the constructor of the Thread class.
5) Finally, the start() method is called on the thread that will call the run() method internally, & the thread's execution will begin.

Runnable Interface vs. Extending Thread class

Since, we've discussed both ways to create Thread in Java.
The Runnable Interface is preferred over extending the Thread class for the

foll. reasons.

→ As multiple inheritance is not supported in Java, it is impossible to extend the Thread class if your class had already extended some other class.

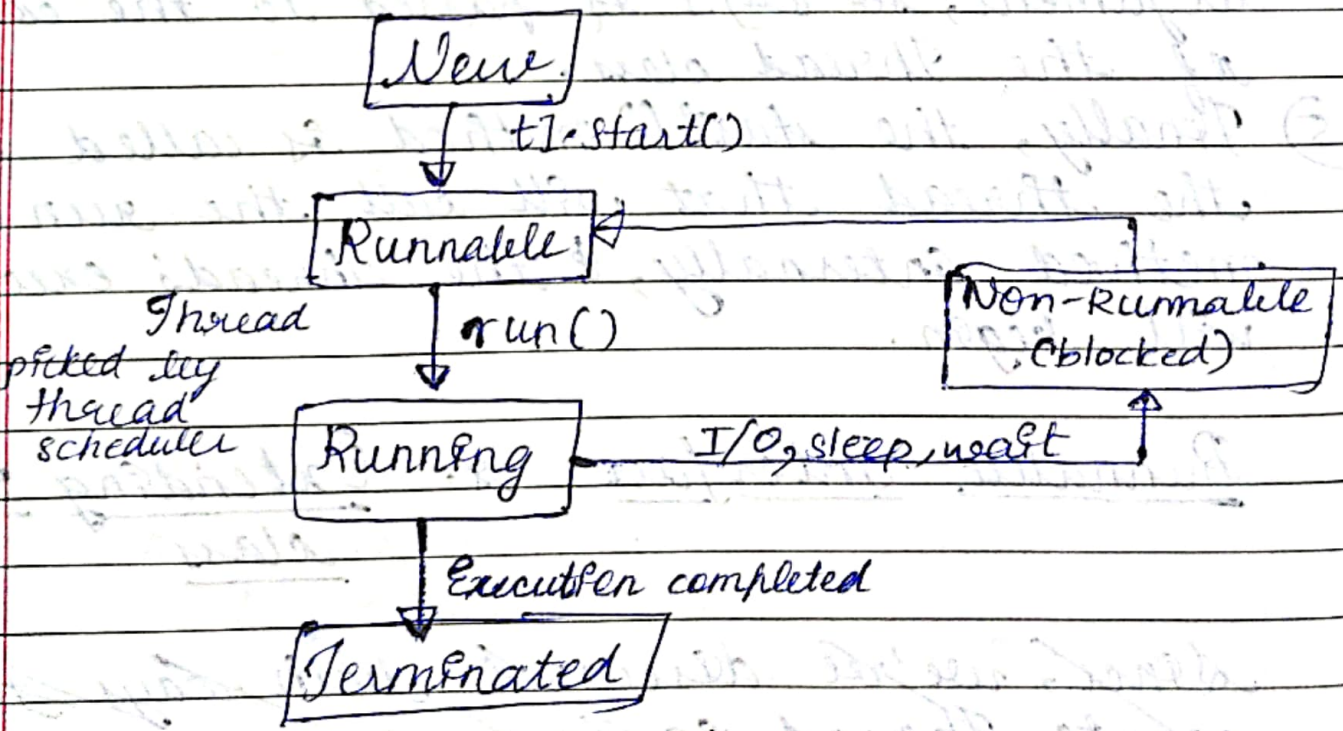
→ While implementing Runnable, we don't modify or change the thread's behaviour.

→ More memory is required while extending the Thread class because each thread creates a unique object.

→ Less memory is req. while implementing Runnable because multiple Threads share the same object.

Java Thread Life cycle.

Thread t1 = new Thread()



1) New - Instance of thread created which is not yet started by invoking start(). In this

states, the thread is also known as the born thread.

2) Runnable - After invocation of start() & before it is selected to be run by the schedules.

3) Running : After the thread schedules has selected it.

4) Non-runnable : Thread alive, not eligible to run.

5) Terminated : run() method has exited

Constructors from Thread class

The Thread class

1) Thread()
2) Thread(string)
3) Thread(Runnable r)
4) Thread(Runnable r, string name)

```
class MyThr extends Thread{
    public MyThr(String name){
        super(name);
    }
public void run(){
    int i=34;
    sout(n("Thank you");
    }
}
```

```
public class Thread{

    p s v m ( ) {
    MyThr t1 = new MyThr("Harry");
    MyThr t2 = new MyThr("Ram Gandi");
     t1.start();
     t2.start();
    sout("The id of thread t1 "+ t1.getId()
    sout("The name of thread t1 "+t1.getName()
    sout("The id of thread t2 "+t2.getId());
```

Java Thread Properties

In Multithreading environment, all the threads which are ready & waiting to be executed are present in the Ready queue. The thread scheduler is responsible for assigning the processor to a thread. But the question is on what basis the thread scheduler decides that a particular thread will run before other threads. Here comes the concept of priority in the picture.

→ Every single thread in Java has some priority associated with it.
→ The programmer can explicitly assign some priority to the thread. Otherwise, JVM automatically assigns priority while creating the thread.
→ In Java, we can specify the priority of each thread relative to other threads. Those threads having higher priorities get greater access to the available resources than the lower priorities threads.
→ Thread scheduler will use priorities ~~while~~ while allocating processor.
→ The valid range of thread priorities is 1 to 10 (but not 0 to 10)
→ If there is more than one thread of same priority in the queue, then the thread scheduler picks any one of them to execute.
→ The following static final integer constants are defined on the Thread class.

- MIN_PRIORITY: Min. priority that a thread can have is value 1.

- NORM_PRIORITY: This is the default priority ~~that~~ automatically assigned by JVM to a thread if a programmer does not explicitly set the priority of that thread. Value is 5.

- MAX_PRIORITY: Maximum priority that a thread can have, value is 10.

Priority methods in Java

1) setPriority():
- This method is used to set Priority of of thread. ~~Illegal argument~~ IllegalArgumentException is thrown if the priority given by user is out of the range [1, 10];

Syntax:

public final void setPriority(int x);

// x is the priority [1, 10] that is to be
// set for the thread

2) getPriority():
This method is used to display the priority of a given thread.

Syntax:
t1.getPriority()
// Will return the priority of the
// t1 thread

Eg:

```
class Priority{
    public void run(){
        System.out.println("I'm a thread: "+
                Thread.currentThread().getName());

System.out.println("I'm a thread: "+
 Thread.currentThread().getPriority());

}

p s v m ( ){

Priority t1 = new Priority();
Priority t2 = new Priority();
t1.setPriority(Thread.MIN_PRIORITY);
// setting priority of t1 thread to
//MIN_PRIORITY(1)
t2.setPriority(Thread.MAX_PRIORITY);
// setting priority of t2 thread to
// MAX_PRIORITY(10)

t1.start();
t2.start();
    }
}
```

o/p

I'm a thread: Thread-0
——— : ——— 1
——— : 10
——— : 1

Thread Methods

→ Join() method in Java allows one thread to wait until the execution of some other specified thread is completed.

→ If t is a thread object whose thread is currently executing, then t.join() causes the current thread to pause execution until t's thread terminates.

→ Join() method puts the current thread on wait until the thread on which it is called is dead.

Syntax:

public final void join()

You can also specify the time for which you need to wait for the execution of a particular thread by using the join() method.

Syntax:

public final void join(long millis)

Thread: Main — Thread: t

Boyfriend — t.join() → Girlfriend

Main: Hey, t, I would wait for you ~~until~~ until you get ready

Thread: Main

Thread: t

t.join(3000);

Boyfriend → Girlfriend

Main: Hey 't', get ready in 3s or I'm leaving

Sleep() Method:

→ The sleep method in Java is useful to put a thread to sleep for a specified amount of time.
→ When we put a thread to sleep, the thread scheduler picks & executes another thread in queue.
→ sleep method returns void.
→ sleep() method can be used for any thread, including main thread.

Syntax:
→ public static void sleep(long millisecond)
→ " (long milli, int nano)
→ throws InterruptedException

Parameters passed to sleep method.

1) long millisecond: Time in milliseconds for which thread will sleep.
2) nano: Ranges from [0, 999999]. Additional time in nanoseconds.

E.g.

```
import java.lang*;
import java.lang.Thread;

public class cwh {
    p s v m ( ) {
        try {
        for(int i=1; i<=5; i++) {
            Thread.sleep(2000);
            System.out.println(i);
        }
    }
    catch(Exception e){
        System.out.println(e);
    }
}
}
```

In the above example, the main() method will be put to sleep for 2 seconds every time the for loop executes

o/p

1
2
3
4
5

Interrupt() method:

→ A thread in sleeping or waiting state can be interrupted by another thread with the help of interrupt method of Thread class.

→ The interrupt method shows interrupted exception

→ The interrupt method won't throw the interrupted exception if the Thread is not in the sleeping/blocked state, but the interrupt flag will be changed to true.

Syntax:

Public void interrupt()

Different scenarios where interrupt method can be used

Case 1: Interrupting a thread doesn't stop working

```
class CWH1 extends Thread{
    @Override public void run(){
        try{
            for(int i=0; i<5; i++){
                sout("Child Thread");
                Thread.sleep(4000); /* Child thread is put to sleep for 4000 ms */
            }
        }
```

```
catch (InterruptedException e)
{
System.out.println("Child Thread Interrupted.");
}
System.out.println("Thread is running");
}
}
```

```
public class CWH extends Thread{
    p s v m ( ){
    CWHI t = new CWHI();
        t.start();
        t.interrupt();
        System.out.println("Main Thread");
    }
}
```

In the above code, for loop runs for the first time, but the child thread is put to sleep after that. So, the main method interrupts the child thread & InterruptedException is generated. Here, the child thread comes out of the sleeping state, but it does not stop working.

Output:

```
Main Thread
Child Thread
Child Thread Interrupted
Thread is running
```

Case 2: Interrupting a thread that works normally:

```
class CWHI extends Thread{
    public void run(){
        for(int i=0; i<10; ++i){
            System.out.println(i);
        }
    }
}

// public class CWH extends Thread{
    p s v m( ){
        CWHI t = new CWHI();
        t.start();
        t.interrupt();
        System.out.println("Main Thread");
    }
}
```

Here the thread works normally because no exception occurred during the thread's execution, so the interrupt() only sets the value of the thread flag to true:

```
o/p
0
1
2
3
4
5
6
7
8
9
```

→ next line