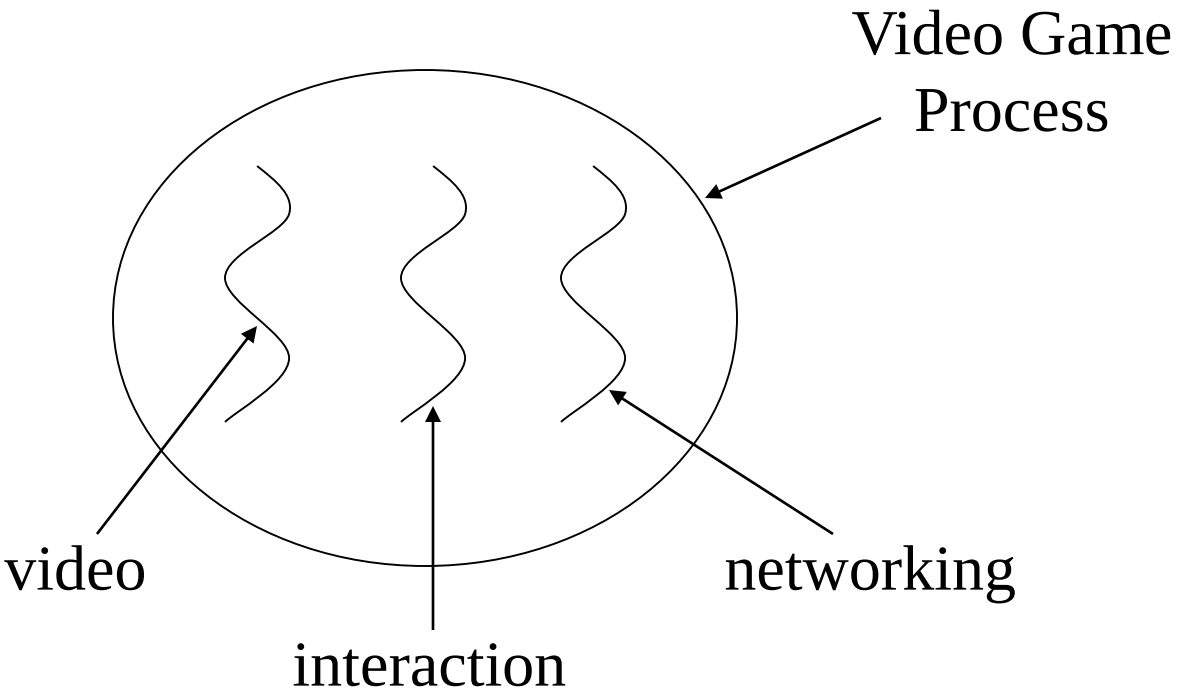

# Concurrency - Threads

Sisoft Technologies Pvt Ltd
SRC E7, Shipra Riviera Bazar, Gyan Khand-3, Indirapuram, Ghaziabad
Website: www.sisoft.in Email:info@sisoft.in
Phone: +91-9999-283-283

# What is a Thread?

- Individual and separate unit of execution that is part of a process
  - multiple threads can work together to accomplish a common goal
- Video Game example
  - one thread for graphics
  - one thread for user interaction
  - one thread for networking

# What is a Thread?



# Advantages

- easier to program
  - 1 thread per task
- can provide better performance
  - thread only runs when needed
  - no polling to decide what to do
- multiple threads can share resources
- utilize multiple processors if available

# Disadvantage

- multiple threads can lead to deadlock
  - much more on this later
- overhead of switching between threads

# Creating Threads (method 1)

- extending the Thread class
  - must implement the *run()* method
  - thread ends when *run()* method finishes
  - call *.start()* to get the thread ready to run

# Creating Threads (method 2)

- **Create thread by implementing Runnable interface:**
  - The simplest way to make a thread is to make a class that implements the runnable interface.
  - To implement the runnable, a class require only implement a single method called 'run( )', which is declared as following :
    ```
    public void run( )
    ```
  - After you make a class that implements 'Runnable', you will instantiate an Object of type 'Thread' from within that class. The thread defines various constructors. The one that we shall use is representing here: `Thread(Runnable threadObj, String threadname);`
  - After the new thread is made, it will not start running until you call it 'start( )' method, which is declared within the Thread. The 'start( )' method is representing here: `void start( );`

# Advantage of Using Runnable

- remember - can only extend one class
- implementing runnable allows class to extend something else

# Threads Class Reference

- *currentThread()*: Returns reference to the currently running thread
- *_.getName(): returns the Thread Name*
- *_.setName(): set the Thread Name*
- *getPriority()*
- *setPriority()*
- *getState()*

# Controlling Java Threads

- _.*start()*: begins a thread running
- *wait()* and *notify()*: for synchronization
  - more on this later
- _.*stop()*: kills a specific thread (deprecated)
- _.*suspend()* and *resume()*: deprecated
- _.*join()*: wait for specific thread to finish
- _.*setPriority()*: 0 to 10 (MIN_PRIORITY to MAX_PRIORITY); 5 is default (NORM_PRIORITY)

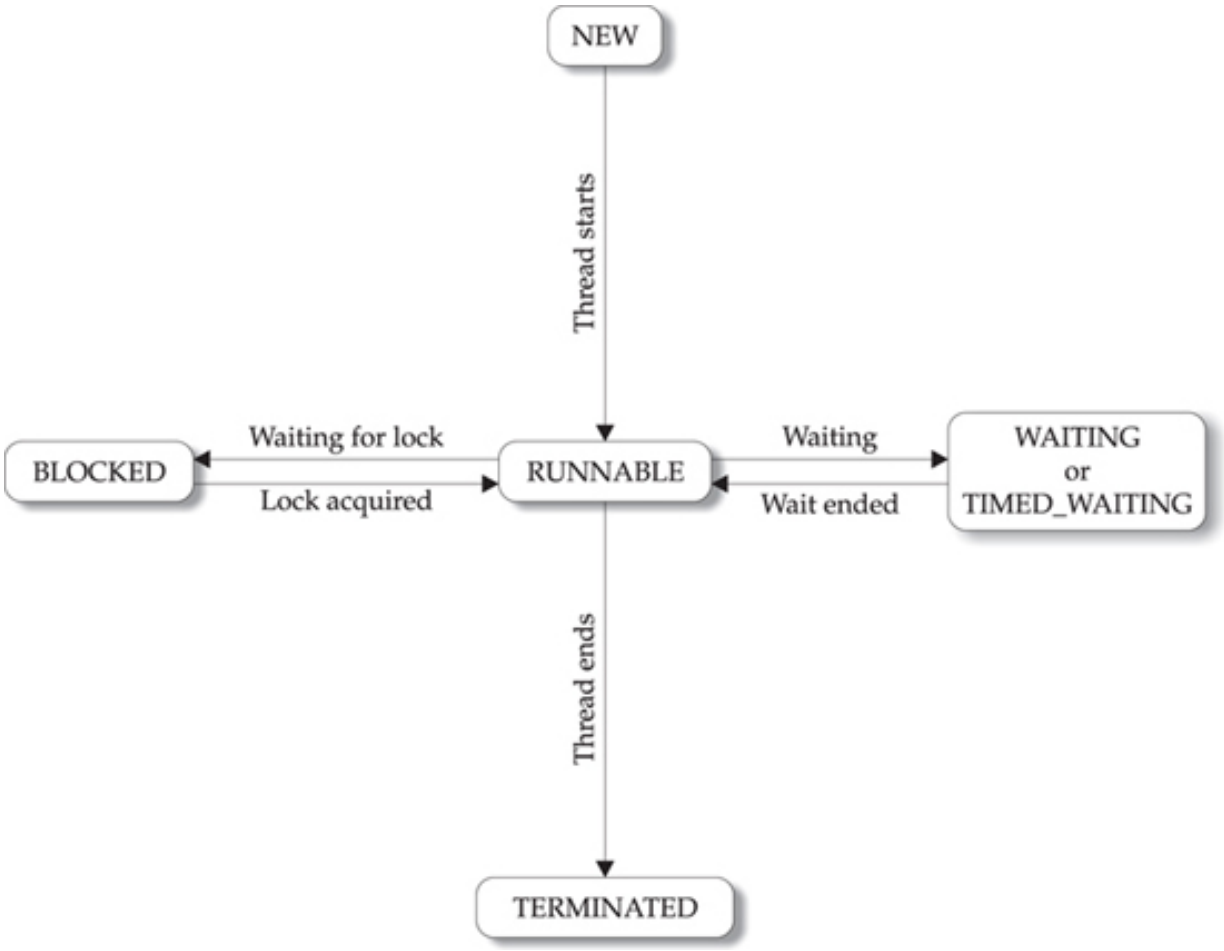
# Java Thread Scheduling

- highest priority thread runs
  - if more than one, arbitrary
- *yield()*: current thread gives up processor so another of equal priority can run
  - if none of equal priority, it runs again
- *sleep(msec)*: stop executing for set time
  - lower priority thread can run

# States of Java Threads

- new: just created but not started
- runnable: created, started, and able to run
- blocked: created and started but unable to run because it is waiting to acquire lock
- timed_waiting: waiting to complete sleep
- waiting: waiting for some action to finish
- terminated: thread has finished or been stopped

# States of Java Threads



# Java Thread Example 1

```
class Job implements Runnable {
    private static Thread [] jobs = new Thread[4];
    private int threadID;
    public Job(int ID) {
        threadID = ID;
    }
    public void run() { do something }
    public static void main(String [] args) {
        for(int i=0; i<jobs.length; i++) {
            jobs[i] = new Thread(new Job(i));
            jobs[i].start();
        }
        try {
            for(int i=0; i<jobs.length; i++) {
                jobs[i].join();
            }
        } catch(InterruptedException e) { System.out.println(e); }
    }
}
```

# Java Thread Example 2

```
class Schedule implements Runnable {
        private static Thread [] jobs = new Thread[4];
        private int threadID;
        public Schedule(int ID) {
                threadID = ID;
        }
        public void run() { do something }
        public static void  main(String [] args) {
                int nextThread = 0;
                setPriority(Thread.MAX_PRIORITY);
                for(int i=0; i<jobs.length; i++) {
                    jobs[i] = new Thread(new Job(i));
                    jobs[i].setPriority(Thread.MIN_PRIORITY);
                    jobs[i].start();
                }
                try {
                    for(;;) {
                        jobs[nextThread].setPriority(Thread.NORM_PRIORITY);
                        Thread.sleep(1000);
                        jobs[nextThread].setPriority(Thread.MIN_PRIORITY);
                        nextThread = (nextThread + 1) % jobs.length;
                    }
                } catch(InterruptedException e) { System.out.println(e); }
        }
}
```

# Thread Synchronization

- When two or more threads require access to the shared resource, they require some way to ensure that the resource will be used by one thread at a time.
- A process by which this synchronization is obtained is called the thread synchronization.
- The 'synchronized' keyword in Java makes a block of code referred to as the critical section. Each Java object with the critical section of code gets a lock associated with the object. To enter the critical section, a thread requires to obtain the corresponding object's lock.
- This is the general form of the 'synchronized' statement:

```
synchronized(object) {
// Statements to be synchronized
}
```

- Here, Object is a reference to the object being 'synchronized'. A 'synchronized' block ensures that the call to a method that is a member of object happens only after the current thread has successfully entered monitor of object.

# Executors

- Objects that encapsulate thread management and creation from the rest of the application are known as *executors .*
- The java.util.concurrent package defines three executor interfaces:
  - Executor, a simple interface that supports launching new tasks.
  - ExecutorService, a subinterface of Executor, which adds features that help manage the lifecycle, both of the individual tasks and of the executor itself.
  - ScheduledExecutorService, a subinterface of ExecutorService, supports future and/or periodic execution of tasks.