MULTITHREADING IN JAVA

Vijay Kumar Meena
Assistant Professor
KIIT University

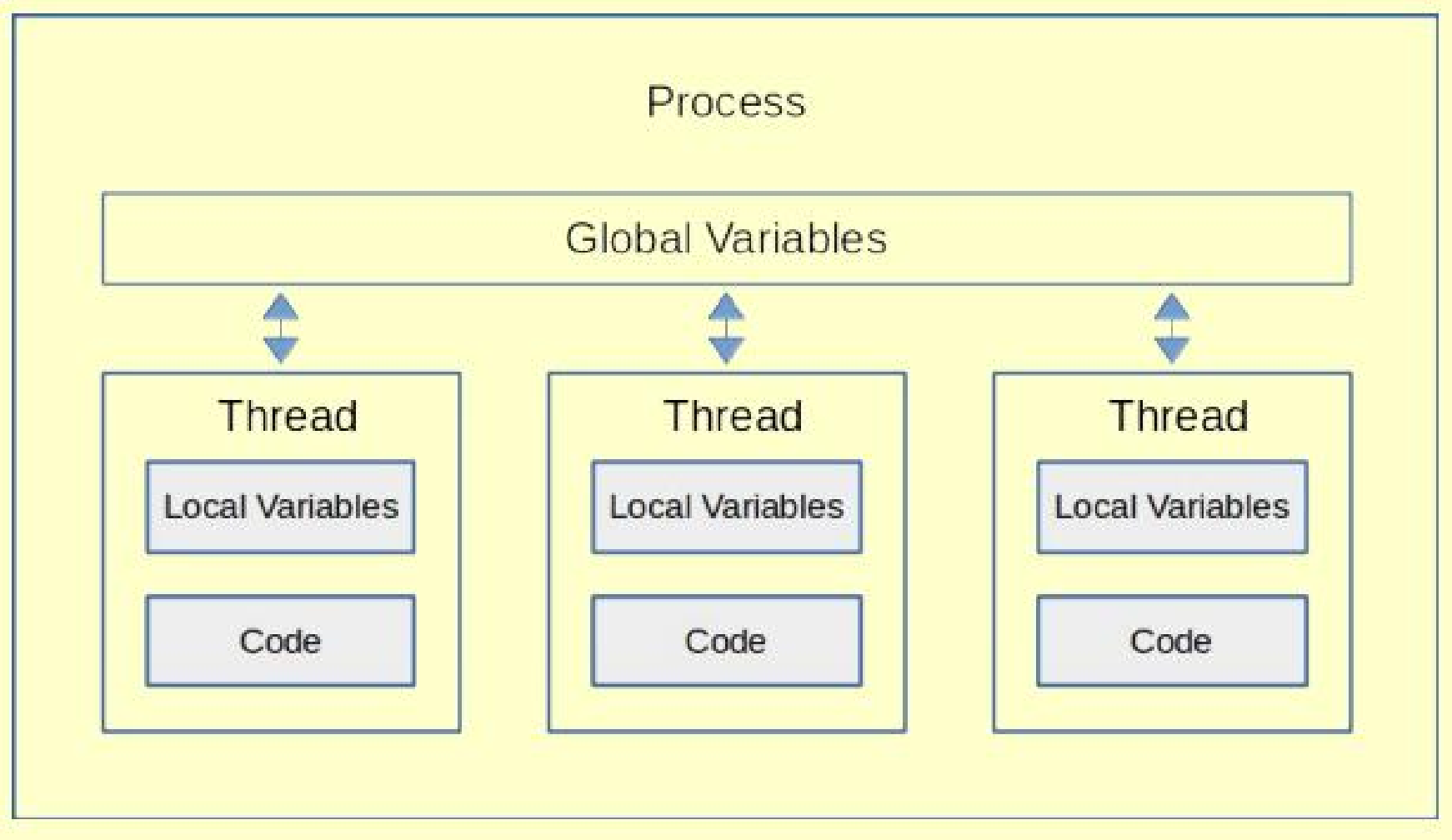
WHAT ARE THREADS?

- A thread is a single sequential (separate) flow of control within program.
- A thread is also called a lightweight process. A process is a program in execution.
- Threads share the address space of a process but has their own program counter and stack for local variable initialization.

MULTI-TASKING

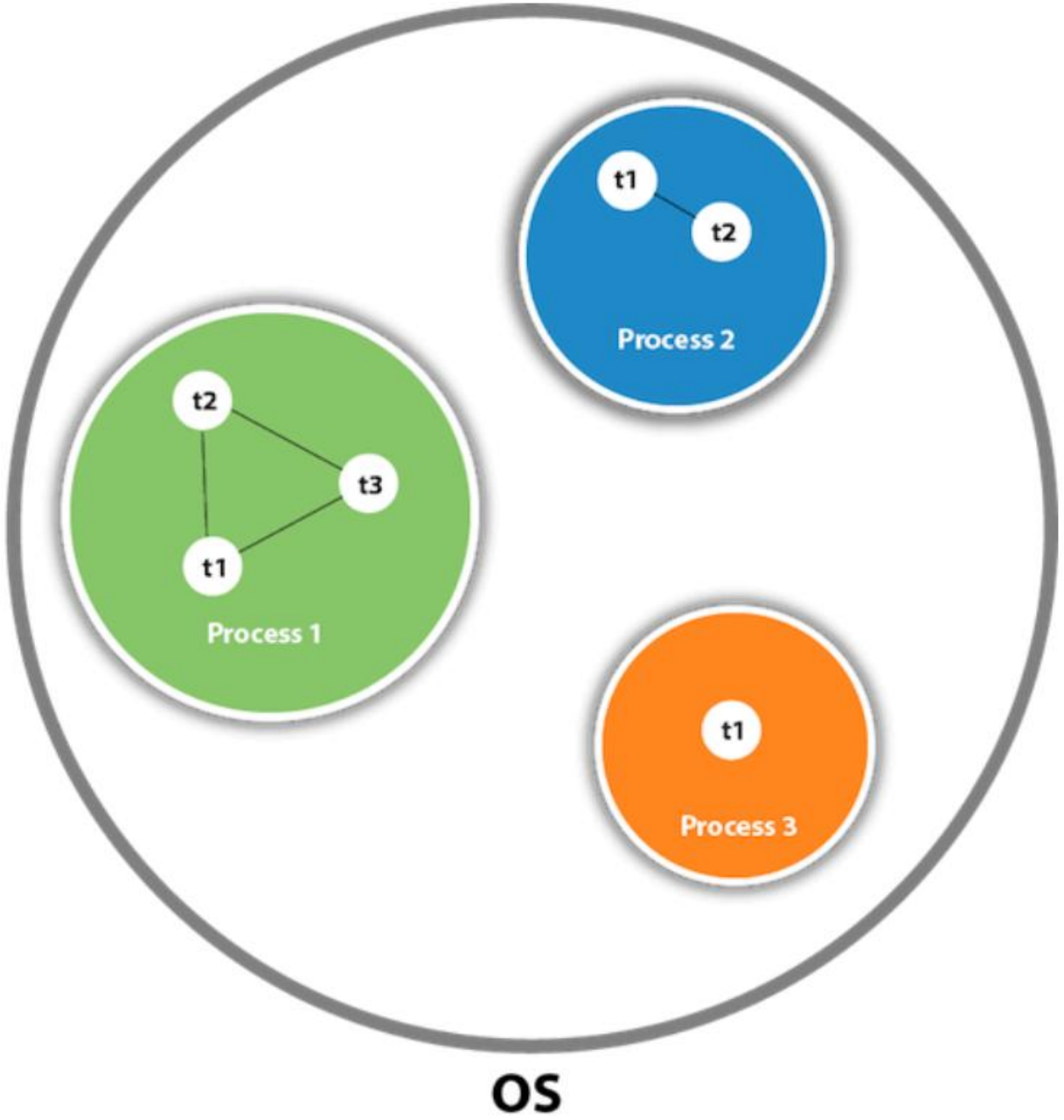
- Executing several tasks simultaneously is called multi-tasking. This can be achieved in two ways -
  - Process based multi-tasking (Multiprocessing)
    - Running several processes together. For example running browser and text editor together.
    - Each process has their own address space in memory.
    - CPU Scheduler executes processes on CPU one at a time. It switches among process to be scheduled on processor, this is known as context switch.
  - Thread based multi-tasking (Multithreading)
    - A process can have multiple threads which can be scheduled simultaneously on CPU for execution.
    - For instance, a text editor can format text at the same time that it is printing, as long as these two actions are being performed by two separate threads.

MULTITHREADING



ADVANTAGES OF MULTITHREADING

- Multitasking threads require less overhead than multitasking processes.
- Processes are heavyweight tasks that require their own separate address spaces. Interprocess communication is expensive and limited. Context switching from one process to another is also costly.
- Threads, on the other hand, are lighter weight. They share the same address space and cooperatively share the same heavyweight process.
- Interthread communication is cheap, and context switching from one thread to the next is lower in cost as it only involves switching out only identities and resources such as the program counter, registers and stack pointers.

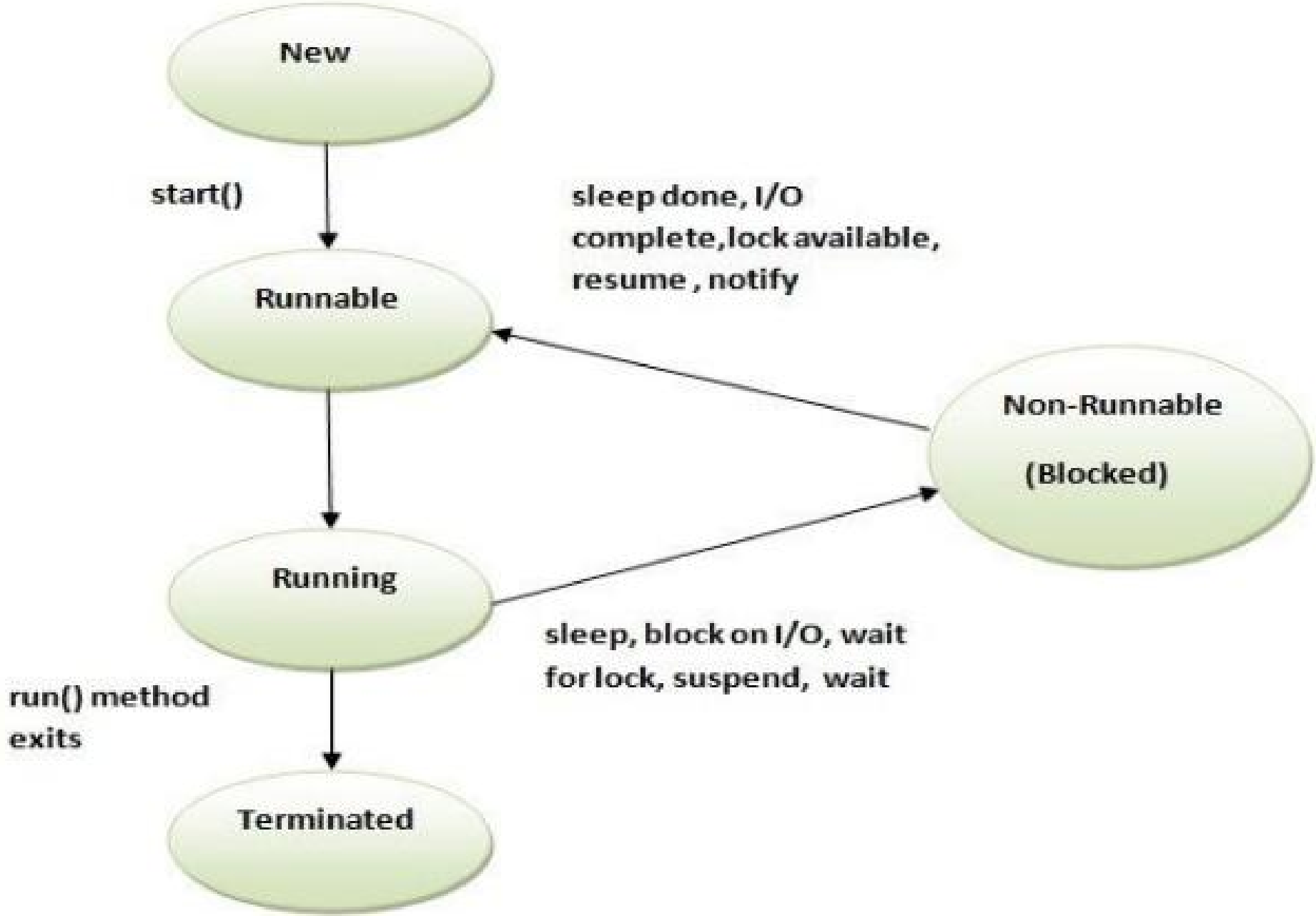

t1
t2
Process 2
t2
t3
t1
Process 1
t1
Process 3
OS

JAVA THREAD MODEL

- Single-threaded systems use an approach called an event loop with polling. In this model, a single thread of control runs in an infinite loop, polling a single event queue to decide what to do next.
- The benefit of Java's multithreading is that the main loop/polling mechanism is eliminated. One thread can pause without stopping other parts of your program.
- In a single core system, concurrently executing threads share the CPU, with each thread receiving a slice of CPU time. Therefore, in a single-core system, two or more threads do not actually run at the same time, but idle CPU time is utilized.
- However, in multi-core systems, it is possible for two or more threads to actually execute simultaneously. In many cases, this can further improve program efficiency and increase the speed of certain operations.

LIFE CYCLE OF A THREAD

- A thread can be any of the five following states -
  - Newborn State
  - Runnable State
  - Running State
  - Blocked State
  - Dead State


New
start()
Runnable
sleep done, I/O
complete,lock available,
resume , notify
Non-Runnable
(Blocked)
Running
sleep, block on I/O, wait
for lock, suspend, wait
run() method
exits
Terminated

THREAD CLASS

- Java's multithreading system is built upon the **Thread class** and its companion interface **Runnable.**
- Thread encapsulates a thread of execution. Since you can't directly refer to the ethereal state of a running thread, you will deal with it through its proxy, the Thread instance that spawned it.

Method	Meaning
getName	Obtain a thread's name.
getPriority	Obtain a thread's priority.
isAlive	Determine if a thread is still running.
join	Wait for a thread to terminate.
run	Entry point for the thread.
sleep	Suspend a thread for a period of time.
start	Start a thread by calling its run method.

THE MAIN THREAD

- When we run any java program, the program begins to execute its code starting from the main method.
- The JVM creates a thread to start executing the code present in main method. This thread is called as main thread.
- Although the main thread is automatically created, you can control it by obtaining a reference to it by calling ***currentThread()*** method which is a *public static* member of T*hread class*.
- All the other threads are created from main thread.
- Main thread must be always the last thread to finish execution.

THE MAIN THREAD

MainThread.java

```java
public class MainThread {
    public static void main(String[] args) {
        Thread mainThread = Thread.currentThread();

        System.out.println("Name: " + mainThread.getName());
        System.out.println("ID: " + mainThread.getId());
        System.out.println("Priority: " + mainThread.getPriority());
        System.out.println("Thread Group: " + mainThread.
        getThreadGroup().getName());
        System.out.println("State: " + mainThread.getState());

        try {
            System.out.println("Sleeping main thread for 3 seconds...
            ");
            Thread.sleep(3000); // Sleep for 3 seconds
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("After sleep, State: " + mainThread.
        getState());
    }
}
```

```
vijay@DESKTOP-58F6BI5:/mnt/d/OOPJ/Codes/Slides/Multithreading$ javac MainThread.java
vijay@DESKTOP-58F6BI5:/mnt/d/OOPJ/Codes/Slides/Multithreading$ java MainThread
Name: main
ID: 1
Priority: 5
Thread Group: main
State: RUNNABLE
Sleeping main thread for 3 seconds...
After sleep, State: RUNNABLE
vijay@DESKTOP-58F6BI5:/mnt/d/OOPJ/Codes/Slides/Multithreading$
```

CREATING THREADS

- In Java, you can create a thread by instantiating an object of type ***Thread***. You can this in two ways -
  - You can implement the **Runnable** interface.
  - You can extend the **Thread** class itself.

IMPLEMENTING RUNNABLE INTERFACE

- The easiest way to create a thread is to create a class that implements the **Runnable interface**. *Runnable* abstracts a unit of executable code. You can construct a thread on any object that implements Runnable.
- To implement **Runnable** interface, you need to provide implementation for a method *public void run()*. Inside the *run()*, you define the code that should be executed by the new thread.
- The *run()* method is the entry point for another concurrent thread of execution within your program. This thread will end when *run()* method returns.

IMPLEMENTING RUNNABLE INTERFACE

- After you create a class that implements Runnable, you will instantiate an object of type Thread from within that class. Thread defines several constructors. One of them is ***Thread(Runnable threadOb, String threadName).*** Here *threadOb* is the object of class which implements Runnable interface.
- After the new thread is created, it will not start running until you call its ***void start( )*** method, which is declared within *Thread class*. In essence, *start( )* executes a call to *run( )* method.

IMPLEMENTING RUNNABLE INTERFACE

```
class NewThread implements Runnable{
    Thread nthread;

    NewThread(){
        nthread = new Thread(this, "Custom Thread");

        System.out.println("Child thread: " + nthread);
        nthread.start();
    }

    public void run(){
        try{
            for(int i=1; i<=10; i++){
                System.out.println("Child thread: " + i);
                Thread.sleep(1000);
            }
        }
        catch(InterruptedException e){
            System.out.println("Child interrupted.");
        }

        System.out.println("Exiting child thread.");
    }
}

class Driver{
    public static void main(String[] args){
        NewThread nt = new NewThread();

        try{
            for(int i=1; i<=20; i++){
                System.out.println("Main Thread: " + i);
                Thread.sleep(1000);
            }
        }
        catch(InterruptedException e){
            System.out.println("Main thread interrupted.");
        }
        System.out.println("Main thread exiting.");
    }
}
```

EXTENDING THREAD CLASS

- Another way to create thread in java is to create a new class that extends the class **Thread.** And then create instance of that class.
- The child class must override the **run()** method, which is the entry point for the new thread.
- The child class must also call **start()** method to begin execution of the new thread.

EXTENDING THREAD CLASS

```
class NewThread extends Thread{
    NewThread(){
        super("New Thread");

        System.out.println("Child thread: " + this);
        start();
    }

    public void run(){
        try{
            for(int i=1; i<=10; i++){
                System.out.println("Child thread: " + i);
                Thread.sleep(1000);
            }
        }
        catch(InterruptedException e){
            System.out.println("Child interrupted.");
        }

        System.out.println("Exiting child thread.");
    }
}

class Driver{
    public static void main(String[] args){
        NewThread nt = new NewThread();

        try{
            for(int i=1; i<=20; i++){
                System.out.println("Main Thread: " + i);
                Thread.sleep(1000);
            }
        }
        catch(InterruptedException e){
            System.out.println("Main thread interrupted.");
        }
        System.out.println("Main thread exiting.");
    }
}
```

WHICH WAY IS BETTER?

- As we have, there are two ways to create new threads in java - By implementing Runnable interface or By extending thread class.
- So which one is the better way?
- Many Java programmers feel that classes should be extended only when they are being enhanced or modified in some way. So if you will not be overriding any of Thread class's other methods (except run() as it should always be overridden), then it is probably best simply to implement Runnable interface.
- Also, by implementing Runnable, your thread class does not need to inherit Thread, making it free to inherit a different class.

CREATING MULTIPLE THREADS

- You can create multiple threads by creating multiple objects of thread class.
- Main thread should end after all other threads has been finished.

```
class NewThread implements Runnable{
    String tname;
    Thread tObj;

    NewThread(String tname){
        this.tname = tname;
        tObj = new Thread(this, tname);

        System.out.println("New Thread: " + tObj);
        tObj.start();
    }

    public void run(){
        try{
            for(int i=0; i<5; i++){
                System.out.println(tname + ": " + i);
                Thread.sleep(1000);
            }
        }
        catch(InterruptedException e){
            System.out.println(tname + " Interrupted.");
        }

        System.out.println(tname + " exiting.");
    }
}

class Driver{
    public static void main(String[] args){
        new NewThread("One");
        new NewThread("Two");
        new NewThread("Three");

        try{
            // Put main thread on sleep, to wait for other threads to end
            Thread.sleep(10000);
        }
        catch(InterruptedException e){
            System.out.println("Main thread interrupted.");
        }
        System.out.println("Main thread exiting.");
    }
}
```

USING ISALIVE() AND JOIN() METHODS

- As mentioned before, often you will want the main thread to finish last. In the last example, this is accomplished by calling s*leep( )* within main( ), with a long enough delay to ensure that all child threads terminate prior to the main thread. But we can't always be sure how much time a child thread will take.
- The method **final boolean isAlive()** method returns *true* if the thread upon which it is called is still running. It returns *false* otherwise.
- The method **final void join() throws InterruptedException** waits until the thread on which it is called terminates.

USING ISALIVE() AND JOIN() METHODS

```
class NewThread implements Runnable{
    String tname;
    Thread tObj;
    NewThread(String tname){
        this.tname = tname;
        tObj = new Thread(this, tname);
        System.out.println("New Thread: " + tObj);
        tObj.start();
    }
    public void run(){
        try{
            for(int i=0; i<5; i++){
                System.out.println(tname + ": " + i);
                Thread.sleep(1000);
            }
        }
        catch(InterruptedException e){
            System.out.println(tname + " Interrupted.");
        }
        System.out.println(tname + " exiting.");
    }
}
class Driver{
    public static void main(String[] args){
        NewThread ob1 = new NewThread("One");
        NewThread ob2 = new NewThread("Two");
        NewThread ob3 = new NewThread("Three");
        System.out.println("Thread One is alive: " + ob1.tObj.isAlive());
        System.out.println("Thread Two is alive: " + ob2.tObj.isAlive());
        System.out.println("Thread Three is alive: " + ob1.tObj.isAlive());

        try{
            System.out.println("Waiting for threads to finish.");
            ob1.tObj.join();
            ob2.tObj.join();
            ob3.tObj.join();
        }
        catch(InterruptedException e){
            System.out.println("Main thread interrupted.");
        }
        System.out.println("Main thread exiting.");
    }
}
```

THREAD PRIORITIES

- Java assigns to each thread a priority that determines how that thread should be treated with respect to the others. Thread priorities are integers that specify the relative priority of one thread to another.
- As an absolute value, a priority is meaningless; a higher-priority thread doesn't run any faster than a lower-priority thread if it is the only thread running. Instead, a thread's priority is used to decide when to switch from one running thread to the next. This is called a *context switching*.
- ***A thread can voluntarily relinquish control.*** This is done by explicitly yielding, sleeping, or blocking on pending I/O. In this scenario, all other threads are examined, and the highest-priority thread that is ready to run is given the CPU.
- ***A thread can be preempted by a higher-priority thread***. In this case, a lower-priority thread that does not yield the processor is simply preempted—no matter what it is doing— by a higher-priority thread. Basically, as soon as a higher-priority thread wants to run, it does. This is called *preemptive multitasking or preemptive scheduling*.

THREAD PRIORITIES

- Thread priorities are used by the thread scheduler to decide when each thread should be allowed to run next.
- In theory, over a given period of time, higher-priority threads get more CPU time than lower-priority threads. In practice, the amount of CPU time that a thread gets often depends on several factors besides its priority like how an operating system implements multitasking can affect the relative availability of CPU time.
- A higher-priority thread can also preempt a lower-priority one. For instance, when a lower-priority thread is running and a higher-priority thread resumes (from sleeping or waiting on I/O, for example), it will preempt the lower-priority thread.
- In theory, threads of equal priority should get equal access to the CPU. But in practice, this might also differ in some manner because of internal implementation of underline environment.

THREAD PRIORITIES

- The method **final void setPriority(int level)** can be used to set the priority of a thread. The value of level must be within the range MIN_PRIORITY and MAX_PRIORITY. Currently, these values are 1 and 10, respectively.
- To return a thread to default priority, specify NORM_PRIORITY, which is currently 5. These priorities are defined as static final variables within Thread.
- The method **final int getPriority()** can be used to obtain the current priority level of a thread.

SYNCHRONIZATION

- Because multithreading introduces an asynchronous behavior to your programs, there must be a way for you to enforce synchronicity when you need it.
- For example, if you want two threads to communicate and share a complicated data structure, such as a linked list, you need some way to ensure that they don't conflict with each other. That is, you must prevent one thread from writing data while another thread is in the middle of reading it.
- Java uses something called ***monitor*** for synchronization purpose. You can think of a monitor as a very small box that can hold only one thread. Once a thread enters a monitor, all other threads must wait until that thread exits the monitor.
- In this way, a monitor can be used to protect a shared asset from being manipulated by more than one thread at a time.

SYNCHRONIZATION

- A *monitor* is an object that is used as a mutually exclusive lock. 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.
- Each Java class has their own monitor already implemented. To enter an object's monitor, just call a method that has been modified with the ***synchronized*** keyword.

NEED OF SYNCHRONIZATION

```
30
31  class Driver{
32      public static void main(String[] args){
33          Callme target = new Callme();
34          Caller ob1 = new Caller(target, "Hello");
35          Caller ob2 = new Caller(target, "Synchronized");
36          Caller ob3 = new Caller(target, "World");
37
38          //* wait for threads to end
39          try{
40              ob1.t.join();
41              ob2.t.join();
42              ob3.t.join();
43          }
44          catch(InterruptedException e){
45              System.out.println("Interrupted");
46          }
47      }
48  }
```

```
vijay@DESKTOP-58F6BI5:/mnt/d/OOPJ/Codes/Slides/Multithreading
$ java Driver
[Synchronized[Hello[World]
]
]
```

Asynchronized.java

```
1   class Callme {
2       void call(String msg){
3           System.out.print("[" + msg);
4           try{
5               Thread.sleep(1000);
6           }
7           catch(InterruptedException e){
8               System.out.println("Interrupted");
9           }
10          System.out.println("]");
11      }
12  }
13  //* Class to create threads
14  class Caller implements Runnable{
15      String msg;
16      Callme target;
17      Thread t;
18
19      Caller(Callme targ, String s){
20          target = targ;
21          msg = s;
22          t = new Thread(this);
23          t.start();
24      }
25
26      public void run(){
27          target.call(msg);
28      }
29  }
```

NEED OF SYNCHRONIZATION

- As you can see in last example, by calling sleep( ), the call( ) method allows execution to switch to another thread. This results in the mixed-up output of the three message strings.
- In this program, nothing exists to stop all three threads from calling the same method, on the same object, at the same time. This is known as a *race condition*, because the three threads are racing each other to complete the method. This example used sleep( ) to make the effects repeatable and obvious.
- To make sure that the program run correctly each time, we need to make sure that only one thread has access to **call()** method at a time. This can be done by adding **synchronized** keyword in definition of **call()** method.

USING SYNCHRONIZED METHODS

- Synchronization is easy in Java, because all objects have their own implicit monitor associated with them.
- To enter an object's monitor, just call a method that has been modified with the ***synchronized*** keyword.
- While a thread is inside a synchronized method, all other threads that try to call it (or any other synchronized method) on the same instance have to wait. To exit the monitor and relinquish control of the object to the next waiting thread, the owner of the monitor simply returns from the synchronized method.

SYNCHRONIZED METHOD

```
class Driver{
    public static void main(String[] args){
        Callme target = new Callme();
        Caller ob1 = new Caller(target, "Hello");
        Caller ob2 = new Caller(target, "Synchronized");
        Caller ob3 = new Caller(target, "World");

        //* wait for threads to end
        try{
            ob1.t.join();
            ob2.t.join();
            ob3.t.join();
        }
        catch(InterruptedException e){
            System.out.println("Interrupted");
        }
    }
}
```

```
vijay@DESKTOP-58F6BI5:/mnt/d/OOPJ/Codes/Slides/Multithreading
$ java Driver
[Hello]
[World]
[Synchronized]
```

Synchronized.java

```
class Callme {
    synchronized void call(String msg){
        System.out.print("[" + msg);
        try{
            Thread.sleep(1000);
        }
        catch(InterruptedException e){
            System.out.println("Interrupted");
        }
        System.out.println("]");
    }
}
//* Class to create threads
class Caller implements Runnable{
    String msg;
    Callme target;
    Thread t;

    Caller(Callme targ, String s){
        target = targ;
        msg = s;
        t = new Thread(this);
        t.start();
    }

    public void run(){
        target.call(msg);
    }
}
```

THE SYNCHRONIZED STATEMENT

- While creating synchronized methods within classes that you create is an easy and effective means of achieving synchronization, it will not work in all cases.
- Because you might be using object of class which is not defined by you and you don't have access to source code of the class. In this case, you can't make a method synchronized.
- In such cases, you can put calls to the methods defined by this class inside a *synchronized block*. Which has general form - **synchronized(objRef){** *//code that need to be synchronized* **}**
- Here, objRef is a reference to the object being synchronized. A synchronized block ensures that a call to a synchronized method that is a member of objRef's class occurs only after the current thread has successfully entered objRef's monitor.

THE SYNCHRONIZED STATEMENT

```
33
34  class Driver{
35      public static void main(String[] args){
36          Callme target = new Callme();
37          Caller ob1 = new Caller(target, "Hello");
38          Caller ob2 = new Caller(target, "Synchronized");
39          Caller ob3 = new Caller(target, "World");
40
41          //* wait for threads to end
42          try{
43              ob1.t.join();
44              ob2.t.join();
45              ob3.t.join();
46          }
47          catch(InterruptedException e){
48              System.out.println("Interrupted");
49          }
50      }
51  }
```

```
vijay@DESKTOP-58F6BI5:/mnt/d/OOPJ/Codes/Slides/Multithreading
$ java Driver
[Hello]
[World]
[Synchronized]
```

SynchronizedStatement.java

```
1   class Callme {
2       void call(String msg){
3           System.out.print("[" + msg);
4           try{
5               Thread.sleep(1000);
6           }
7           catch(InterruptedException e){
8               System.out.println("Interrupted");
9           }
10          System.out.println("]");
11      }
12  }
13  //* Class to create threads
14  class Caller implements Runnable{
15      String msg;
16      Callme target;
17      Thread t;
18
19      Caller(Callme targ, String s){
20          target = targ;
21          msg = s;
22          t = new Thread(this);
23          t.start();
24      }
25
26      public void run(){
27          //* Synchronized block
28          synchronized(target){
29              target.call(msg);
30          }
31      }
32  }
```

DEADLOCK

- A special type of error that you need to avoid that relates specifically to multitasking is *deadlock*, which occurs when two threads have a circular dependency on a pair of synchronized objects.
- For example, suppose one thread enters the monitor on object X and another thread enters the monitor on object Y. If the thread in X tries to call any synchronized method on Y, it will block as expected. However, if the thread in Y, in turn, tries to call any synchronized method on X, the thread waits forever, because to access X, it would have to release its own lock on Y so that the first thread could complete.
- Deadlock is a difficult error to debug for two reasons -
  - In general, it occurs only rarely, when the two threads time-slice in just the right way.
  - It may involve more than two threads and two synchronized objects. (That is, deadlock can occur through a more convoluted sequence of events than just described.)

GETTING THREAD STATE

- A thread can exist in a number of different states. You can obtain the current state of a thread by calling the **Thread.State getState( )** method defined by Thread.
- **State** is a static enumeration defined in the Thread class. An enumeration or enum is a special type of class in Java which defines a list of constant values.

```
enum Level {
  LOW,
  MEDIUM,
  HIGH
}
```

You can access `enum` constants with the **dot** syntax:

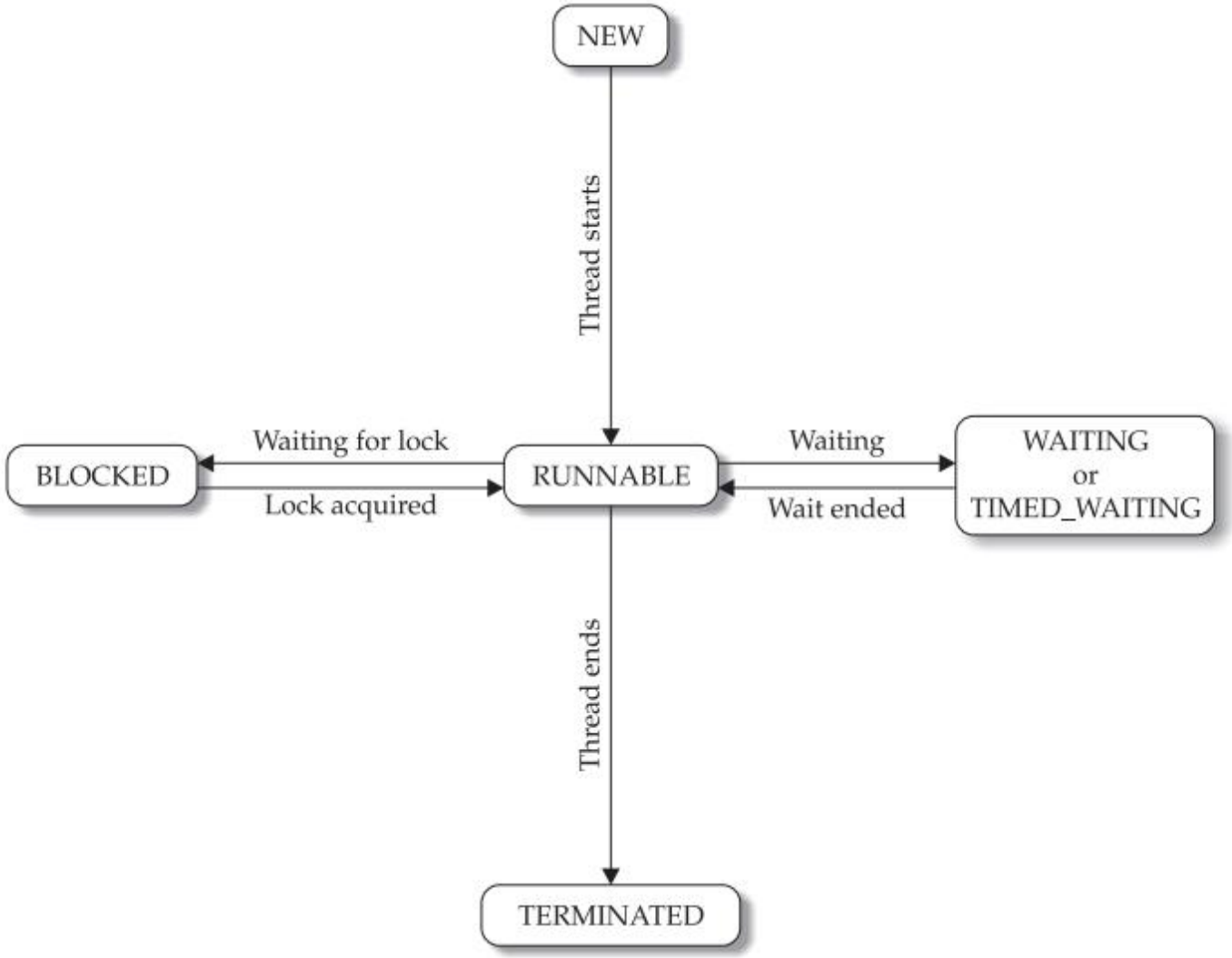
```
Level myVar = Level.MEDIUM;
```

GETTING THREAD STATE

Value	State
BLOCKED	A thread that has suspended execution because it is waiting to acquire a lock.
NEW	A thread that has not begun execution.
RUNNABLE	A thread that either is currently executing or will execute when it gains access to the CPU.
TERMINATED	A thread that has completed execution.
TIMED_WAITING	A thread that has suspended execution for a specified period of time, such as when it has called **sleep( )**. This state is also entered when a timeout version of **wait( )** or **join( )** is called.
WAITING	A thread that has suspended execution because it is waiting for some action to occur. For example, it is waiting because of a call to a non-timeout version of **wait( )** or **join( )**.

GETTING THREAD STATE

- It is important to understand that a thread's state may change after the call to *getState( )*. Thus, depending on the circumstances, the state obtained by calling *getState( )* may not reflect the actual state of the thread only a moment later.



```java
class NewThread extends Thread{
    String tname;
    NewThread(String name){
        tname = name;
    }

    public void run(){
        for(int i=0; i<5; i++){
            System.out.println(tname + " " + i);
            try{
                sleep(1000);
                System.out.println("t1 state: " + getState());
            }
            catch(InterruptedException e){
                System.out.println(e);
            }
        }
    }
}

class Driver{
    public static void main(String[] args){
        NewThread t1 = new NewThread("CustomThread");
        System.out.println("t1 state: " + t1.getState());

        t1.start();
        try{
            Thread.sleep(10000);
        }
        catch(InterruptedException e){
            System.out.println(e);
        }

        System.out.println("t1 state: " + t1.getState());
        Thread mainThread = Thread.currentThread();
        System.out.println("mainThread state: " + mainThread.getState());
    }
}
```

```
vijay@DESKTOP-58F6BI5:/mnt/d/OOPJ/Codes/Slides/Multithreading$ java Driver
t1 state: NEW
CustomThread 0
t1 state: RUNNABLE
CustomThread 1
t1 state: RUNNABLE
CustomThread 2
t1 state: RUNNABLE
CustomThread 3
t1 state: RUNNABLE
CustomThread 4
t1 state: RUNNABLE
t1 state: TERMINATED
mainThread state: RUNNABLE
vijay@DESKTOP-58F6BI5:/mnt/d/OOPJ/Codes/Slides/Multithreading$
```

INTERTHREAD COMMUNICATION

- Java includes an elegant interthread communication mechanism via the **wait( )**, **notify( )**, and **notifyAll( )** methods.
- These methods are implemented as *final methods in Object class*, so all classes have them. All three methods can be called only from within a synchronized context.
- ***final void wait( ) throws InterruptedException*** tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls notify( ) or notifyAll( ).
- ***final void notify( )*** wakes up a thread that called wait( ) on the same object.
- ***final void notifyAll( )*** wakes up all the threads that called wait( ) on the same object. One of the threads will be granted access.

```java
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(InterruptedException e){
                System.out.println(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... ");
        try{
            Thread.sleep(10000);
        }
        catch(InterruptedException e){
            System.out.println(e);
        }
        notify();
    }
}
class Driver{
    public static void main(String[] args){
        Customer c1 = new Customer();
        new Thread(){
            public void run(){
                c1.withdraw(15000);
            }
        }.start();
        new Thread(){
            public void run(){
                c1.deposit(10000);
            }
        }.start();
    }
}
```

```
vijay@DESKTOP-58F6BI5:/mnt/d/OOPJ/Codes/Slides/Multithreading$ java Driver
going to withdraw...
Less balance; waiting for deposit...
going to deposit...
deposit completed...
Withdraw completed...
vijay@DESKTOP-58F6BI5:/mnt/d/OOPJ/Codes/Slides/Multithreading$
```

HOW CAN WE SUSPEND, RESUME OR STOP A THREAD?

- Initial versions of Java provided methods like **suspend(), resume()** and **stop()** in the Thread class for suspending, resuming and stopping a thread respectively.
- But these methods sometimes caused serious system failures and because of that they were deprecated from Java 2.0 version. **Suspend()** and **Stop()** methods allowed thread to be suspended or stopped while holding a lock.
- Instead of using these methods, you can use *wait(), notify(),* and *notifyAll()* methods for same purpose.