Amity School of Engineering & Technology

B.Tech CSE, Semester 5

Java Threads (Unit 3)

Dr. Dolly Sharma

Objectives

After completing this section, students will be able to
▪ Understand Threads in Java and their use
▪ Create multithreaded programs
▪ Use synchronized methods or blocks to synchronize threads

AMITY UNIVERSITY

Contents

1. What is a Thread ?
2. Creating, Implementing and Extending a Thread
3. The life-cycle of a Thread
4. Interrupt a Thread
5. Thread synchronization

What is a Thread ?

- A sequential (or single-threaded) program is one that, when executed, has only one single flow of control.
  - i.e., at any time instant, there is at most only one instruction (or statement or execution point) that is being executed in the program.

What is a Thread ?

- A multi-threaded program is one that can have multiple flows of control when executed.
  - At some instance, there may exist multiple instructions or execution points) that are being executed in the program

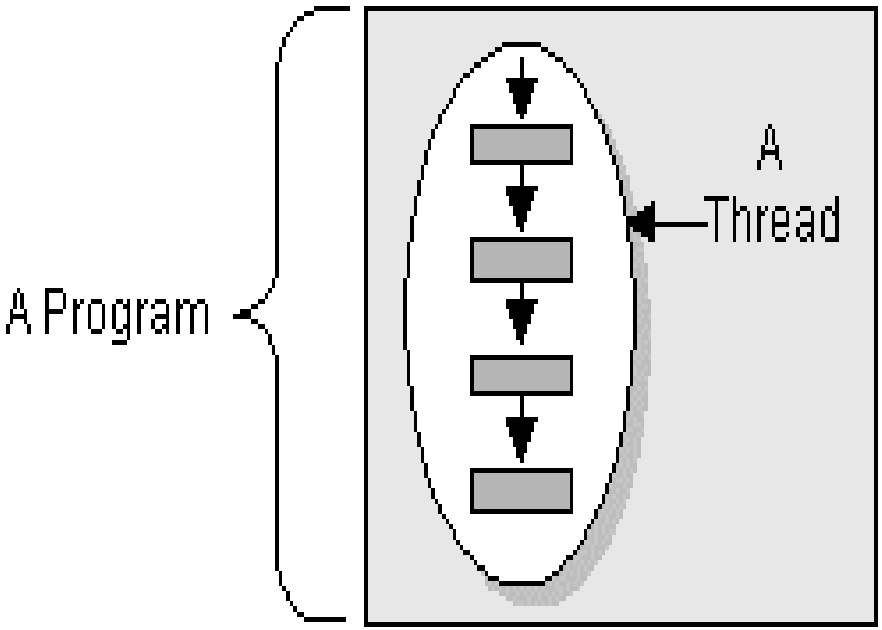
What is a Thread ?

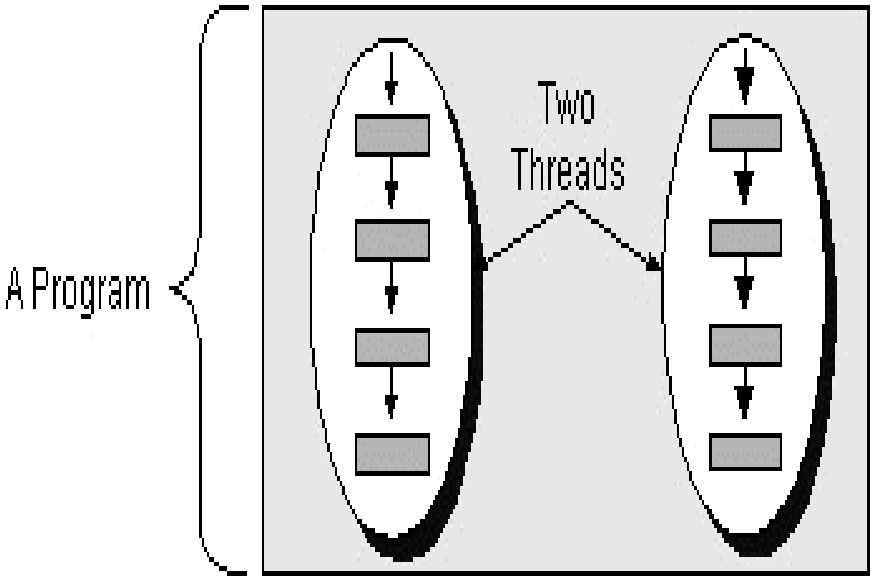
- Ex: in a Web browser we may do the following tasks at the same time:
  - 1. scroll a page,
  - 2. download an applet or image,
  - 3. play sound,
  - 4 print a page

- A thread is a single sequential flow of control within a program.

Thread vs Program?

Single-Threaded vs Multithreaded Programs

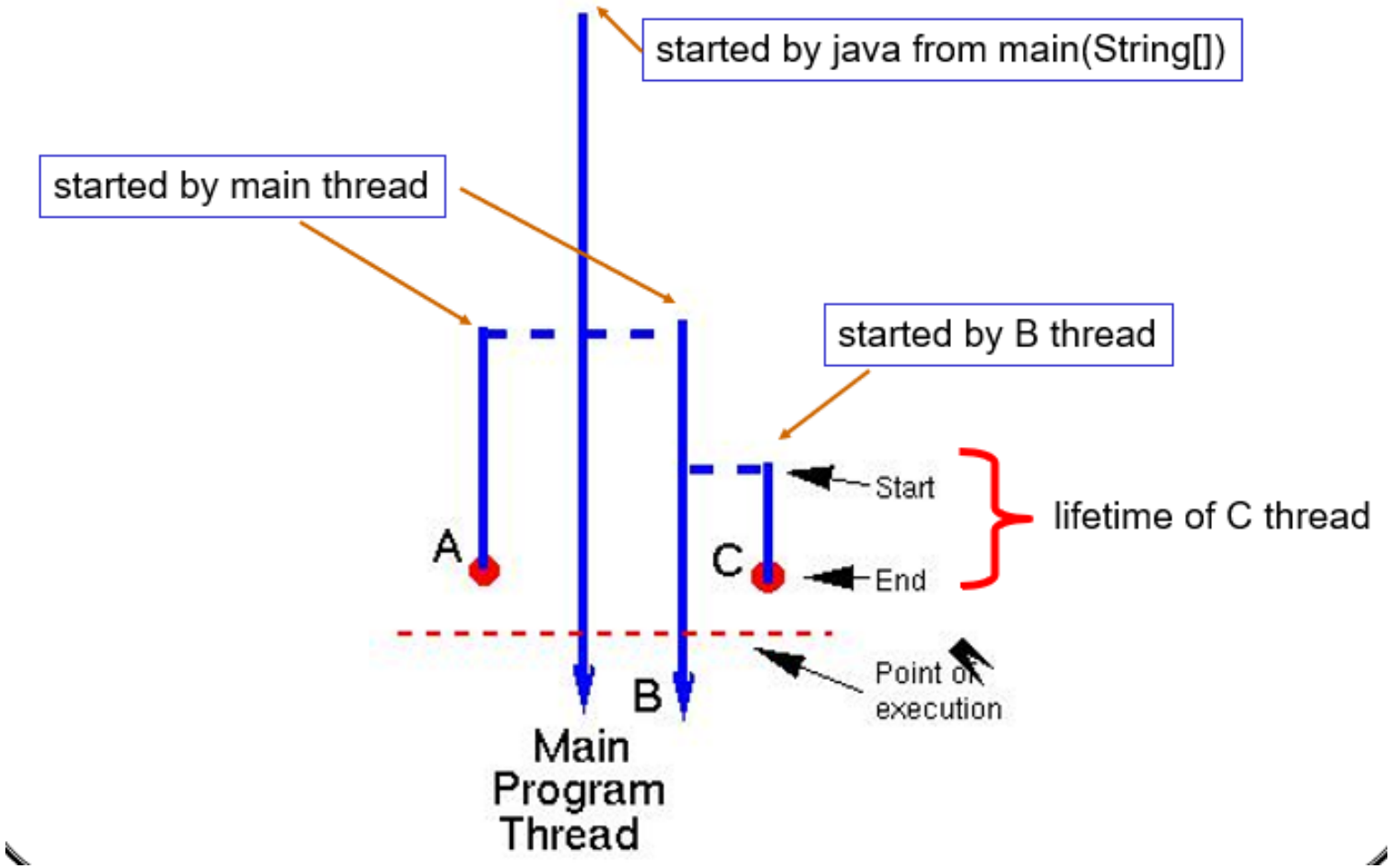





```
{ A(); A1();  A2();   A3();
  B1();  B2();  }
```

```
{  A();
   newThreads {
      { A1(); A2(); A3() };
      {B1(); B2() }
   }
}
```

Thread Ecology in a Java Program



Can we check Threads in Microsoft Word ?

Define and Launch a Java Thread

- Each Java Run time thread is encapsulated in a java.lang.Thread instance.

- Two ways to define a thread:
  1. Extend the Thread class
  2. Implement the Runnable interface :

Define and Launch a Java Thread

Steps for extending the Thread class:

1. Subclass the Thread class;

2. Override the default Thread method run(), which is the entry point of the thread, like the main(String[]) method in a java program.

Define and Launch a Java Thread

Implement the Runnable interface :

```
package java.lang;
public interface Runnable
{
        public void run() ;
}
```

Define a Thread


```
// Example:
public class Print2Console extends Thread {
    public void run() {  // run() is to a thread what
        main() is to a java program
       for (int b = -128; b < 128; b++)
        System.out.println(b);  }
   … // additional methods, fields …
     }
```

Define a Thread

- Impement the Runnable interface if you need a parent class:

```
public class Print2GUI implements Runnable {
    public void run() {
        for (int b = -128; b < 128; b++)
         System.out.println(b);}
     }
```

How to Launch a Thread

1. Create an instance of [a subclass of] Thread, say thread.

   Print2Console thread = new Print2Console()

2. call its start() method, thread.start();. // note: not call run() !!

An Example


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

Output??

thread is running...

An Example


```
class Multi3 implements Runnable{
public void run(){
System.out.println("thread is running...");
}

public static void main(String args[]){
Multi3 m1=new Multi3();
Thread t1 =new Thread(m1);
t1.start();
 }
}
```

Output??

thread is running...

- If you are not extending the Thread class,your class object would not be treated as a thread object.
- So you need to explicitely create Thread class object.
- We are passing the object of your class that implements Runnable so that your class run() method may execute.

Thread Example

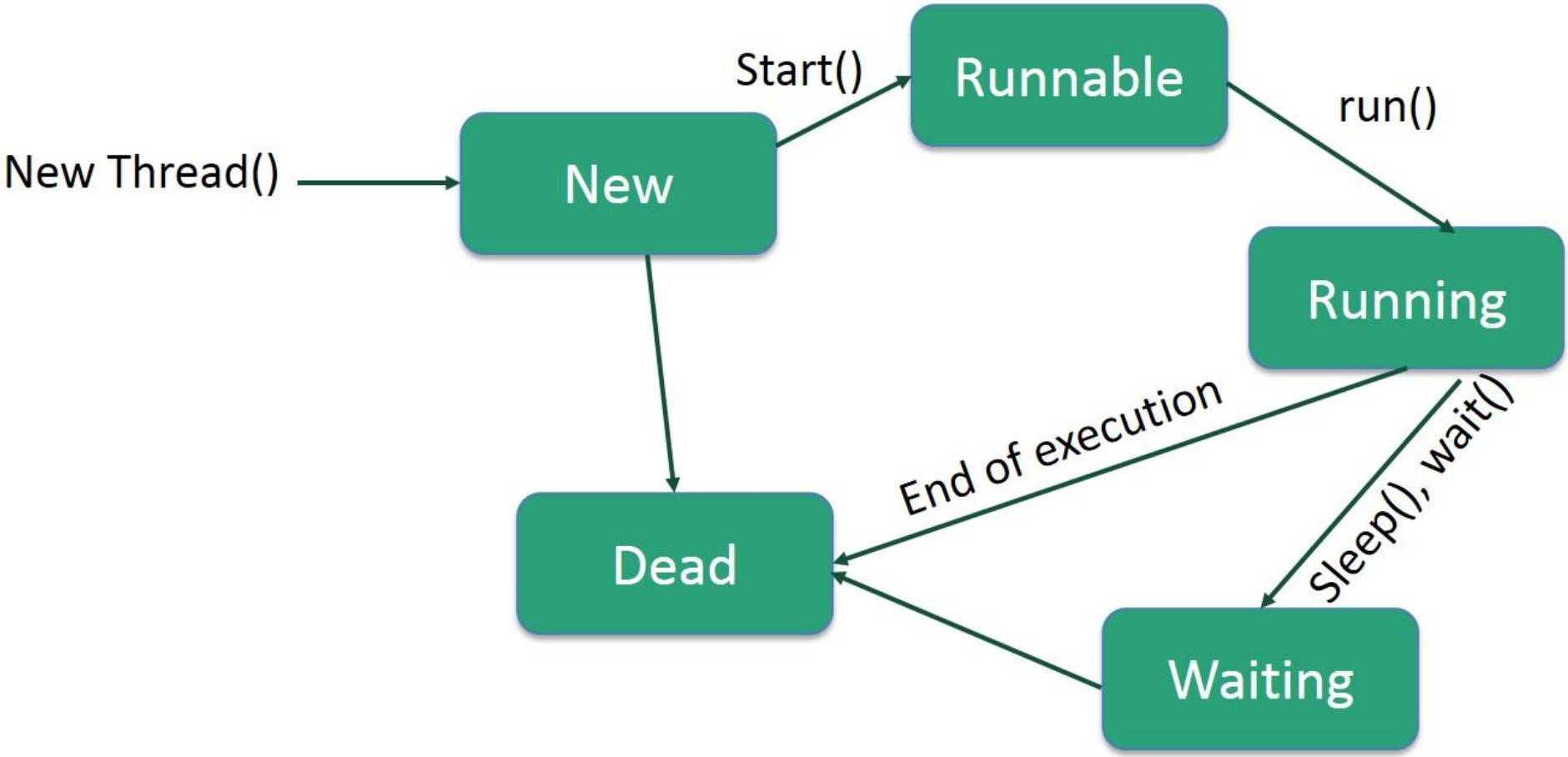

```
public class Main extends Thread {
  public static void main(String[] args) {
    Main thread = new Main();
    thread.start();
    System.out.println("This code is outside of the thread");
  }
  public void run() {
    System.out.println("This code is running in a thread");
  }
}
```

Thread Example


```
public class Main implements Runnable {
  public static void main(String[] args) {
    Main obj = new Main();
    Thread thread = new Thread(obj);
    thread.start();
    System.out.println("This code is outside of the thread");
  }
  public void run() {
    System.out.println("This code is running in a thread");   }  }
```

Life Cycle of a Java Thread

New → (Runnable → blocked/waiting) * → Runnable → dead(terminated)


New Thread()
New
Start()
Runnable
run()
Running
End of execution
Sleep(), wait()
Dead
Waiting

Concurrency Problems

- Because threads run at the same time as other parts of the program, there is no way to know in which order the code will run.
- When the threads and main program are reading and writing the same variables, the values are unpredictable.
- The problems that result from this are called concurrency problems.

Concurrency Problems


```
public class Main extends Thread {
  public static int amount = 0;
  public static void main(String[] args) {
    Main thread = new Main();
    thread.start();
    System.out.println(amount);
    amount++;
    System.out.println(amount);
  }
  public void run() {
    amount++;
  }
}
```

Concurrency Problems

- To avoid concurrency problems, it is best to share as few attributes between threads as possible.
- If attributes need to be shared, one possible solution is to use the isAlive() method of the thread to check whether the thread has finished running before using any attributes that the thread can change.

Concurrency Problem


```
public class Main extends Thread {
  public static int amount = 0;
  public static void main(String[] args) {
    Main thread = new Main();
    thread.start();
    // Wait for the thread to finish
    while(thread.isAlive()) {
      System.out.println("Waiting...");     }
    // Update amount and print its value
    System.out.println("Main: " + amount);
    amount++;
    System.out.println("Main: " + amount);   }
  public void run() {
    amount++;   } }
```

Commonly used Methods of Thread Class

- **public void run():** is 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.

Commonly used Methods of Thread Class

- **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.
- **public Thread.State getState():** returns the state of the thread.
- **public boolean isAlive():** tests if the thread is alive.

Commonly used Methods of Thread Class

- **public void yield():** causes the currently executing thread object to temporarily pause and allow other threads to execute.
- **public void suspend():** is used to suspend the thread(depricated).
- **public void resume():** is used to resume the suspended thread(depricated).
- **public void stop():** is 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.

Commonly used Methods of Thread Class

- **public void wait():** Simply put, calling wait() forces the current thread to wait until some other thread invokes notify() or notifyAll() on the same object.
- **public void wait(long timeout):** Using this method, we can specify a timeout after which a thread will be woken up automatically. A thread can be woken up before reaching the timeout using notify() or notifyAll().
- **public void wait(long timeout, int nanos):** This is yet another signature providing the same functionality. The only difference here is that we can provide higher precision. The total timeout period (in nanoseconds) is calculated as 1_000_000*timeout + nanos.
- **public void notify():** We use the notify() method for waking up threads that are waiting for access to this object's monitor.
- **public void notifyAll():** This method simply wakes all threads that are waiting on this object's monitor.

Sleep Method

- public static void sleep(long miliseconds)throws InterruptedException

- public static void sleep(long miliseconds, int nanos)throws InterruptedException

Sleep Method


```
class TestSleepMethod1 extends Thread{
 public void run(){
  for(int i=1;i<5;i++){
    try{Thread.sleep(500);}catch(InterruptedException e){
System.out.println(e);}
    System.out.println(i);    }   }
 public static void main(String args[]){
  TestSleepMethod1 t1=new TestSleepMethod1();
  TestSleepMethod1 t2=new TestSleepMethod1();
     t1.start();
  t2.start();
} }
```

1 1 2 2 3 3 4 4

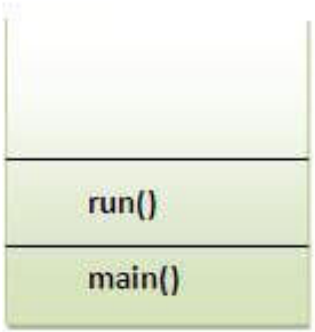
Calling run Method


```
class TestCallRun2 extends Thread{
 public void run(){
  for(int i=1;i<5;i++){
    try{Thread.sleep(500);}catch(InterruptedException e){
System.out.println(e);}
    System.out.println(i);    }   }
 public static void main(String args[]){
  TestCallRun2 t1=new TestCallRun2();
  TestCallRun2 t2=new TestCallRun2();
     t1.run();
  t2.run();
 }  }
```

Output:1
2
3
4
5
1
2
3
4
5
(No context switching)

Calling run Method

- Each thread starts in a separate call stack.
- Invoking the run() method from main thread, the run() method goes onto the current call stack rather than at the beginning of a new call stack.



join Method

- The join() method waits for a thread to die.
- In other words, it causes the currently running threads to stop executing until the thread it joins with, completes its task.

join Method

- Syntax

- public void join()throws InterruptedException
- public void join(long milliseconds)throws InterruptedException

join Method

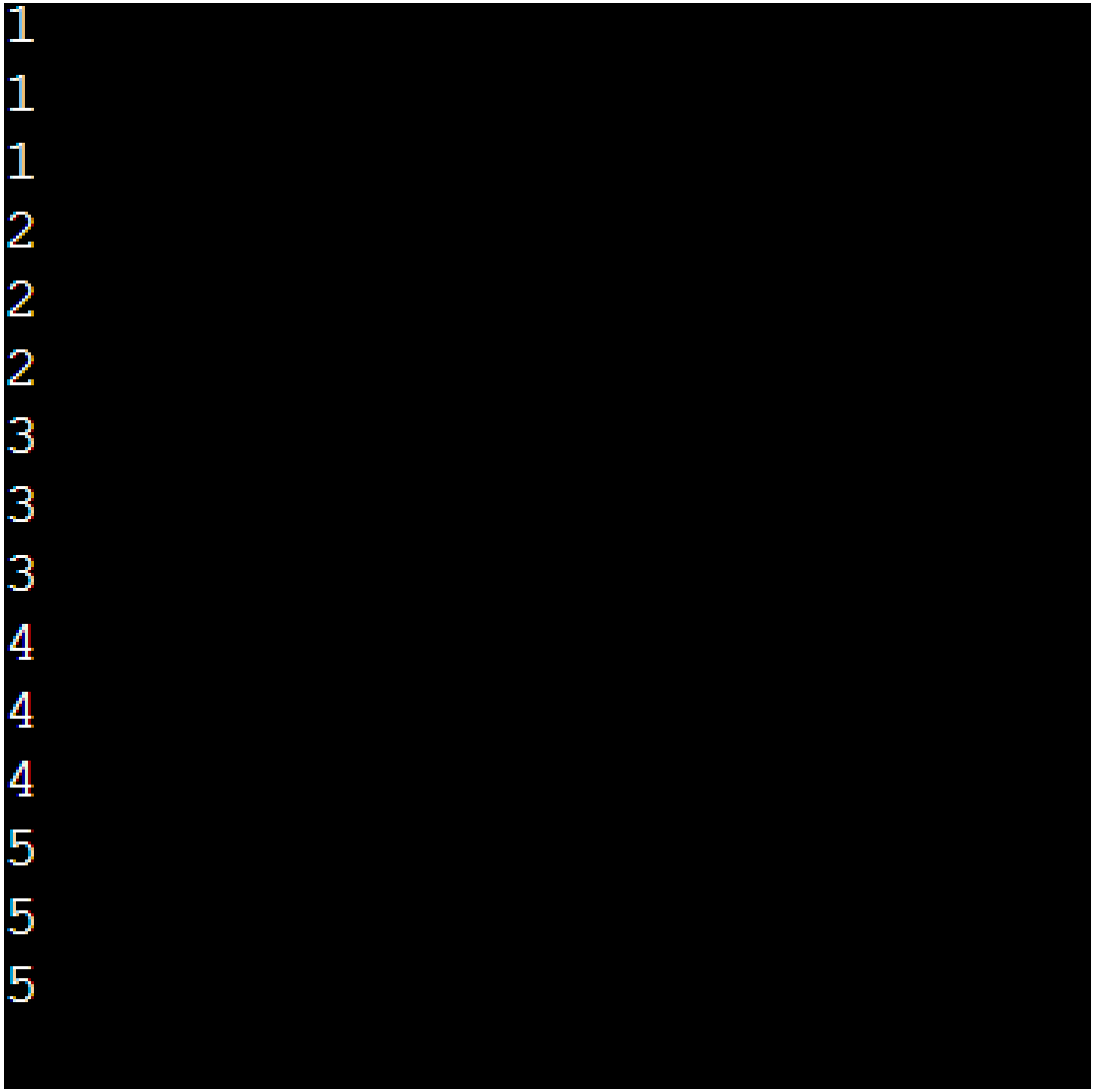

```
class TestJoinMethod1 extends Thread
{
 public void run(){
  for(int i=1;i<=5;i++){
   try{
    Thread.sleep(500);
   }catch(Exception e)
{System.out.println(e);}
  System.out.println(i);
  }   }
```

```
public static void main(String args[]){
 TestJoinMethod1 t1=new TestJoinMethod1();

 TestJoinMethod1 t2=new TestJoinMethod1();

 TestJoinMethod1 t3=new TestJoinMethod1();

 t1.start();
  t2.start();
 t3.start();
 }
}
```


1
1
1
2
2
2
3
3
3
4
4
4
5
5
5

join Method


```
class TestJoinMethod1 extends Thread
{
 public void run(){
  for(int i=1;i<=5;i++){
   try{
    Thread.sleep(500);
   }catch(Exception e)
{System.out.println(e);}
  System.out.println(i);
  }   }
```

Output:1
2
3
4
5
1
1
2
2
3
3
4
4
5
5

```
public static void main(String args[]){
 TestJoinMethod1 t1=new TestJoinMethod1();

 TestJoinMethod1 t2=new TestJoinMethod1();

 TestJoinMethod1 t3=new TestJoinMethod1();

 t1.start();
 try{
  t1.join();
 }catch(Exception e){System.out.println(e);}

 t2.start();
 t3.start();
 }
}
```

join Method


```
class TestJoinMethod1 extends Thread
{
 public void run(){
  for(int i=1;i<=5;i++){
   try{
    Thread.sleep(500);
   }catch(Exception e)
{System.out.println(e);}
  System.out.println(i);
  }   }
```

Output:1
2
3
1
1
4
2
2
5
3
3
4
4
5
5

```
public static void main(String args[]){
 TestJoinMethod1 t1=new TestJoinMethod1();

 TestJoinMethod1 t2=new TestJoinMethod1();

 TestJoinMethod1 t3=new TestJoinMethod1();

 t1.start();
 try{
  t1.join(1500);
 }catch(Exception e){System.out.println(e);}

 t2.start();
 t3.start();
 }
}
```

Thread Priority

- Each thread has a priority.
- Priorities are represented by a number between 1 and 10.
- In most cases, thread schedular 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.

Thread Priority

- 3 constants
- 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.

Thread Priority


```
class TestMultiPriority1 extends Thread{
 public void run(){
   System.out.println("running thread name is:"+Thread.currentThrea
d().getName());
   System.out.println("running thread priority is:"+Thread.currentThre
ad().getPriority());    }

 public static void main(String args[]){
  TestMultiPriority1 m1=new TestMultiPriority1();
  TestMultiPriority1 m2=new TestMultiPriority1();
  m1.setPriority(Thread.MIN_PRIORITY);
  m2.setPriority(Thread.MAX_PRIORITY);
  m1.start();    m2.start();   }  }
```

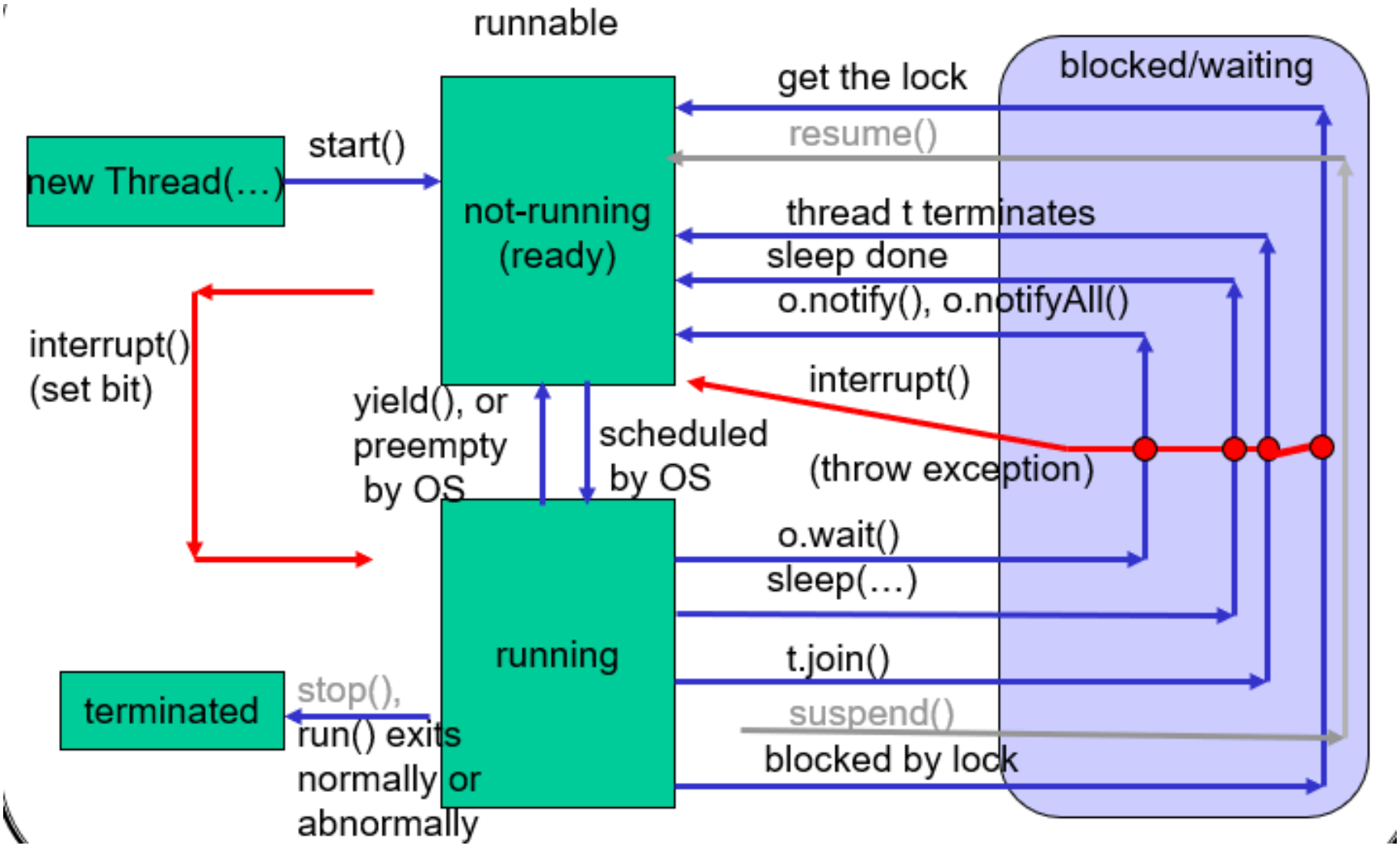
```
running thread name is:Thread-1
running thread priority is:10
running thread name is:Thread-0
running thread priority is:1
```

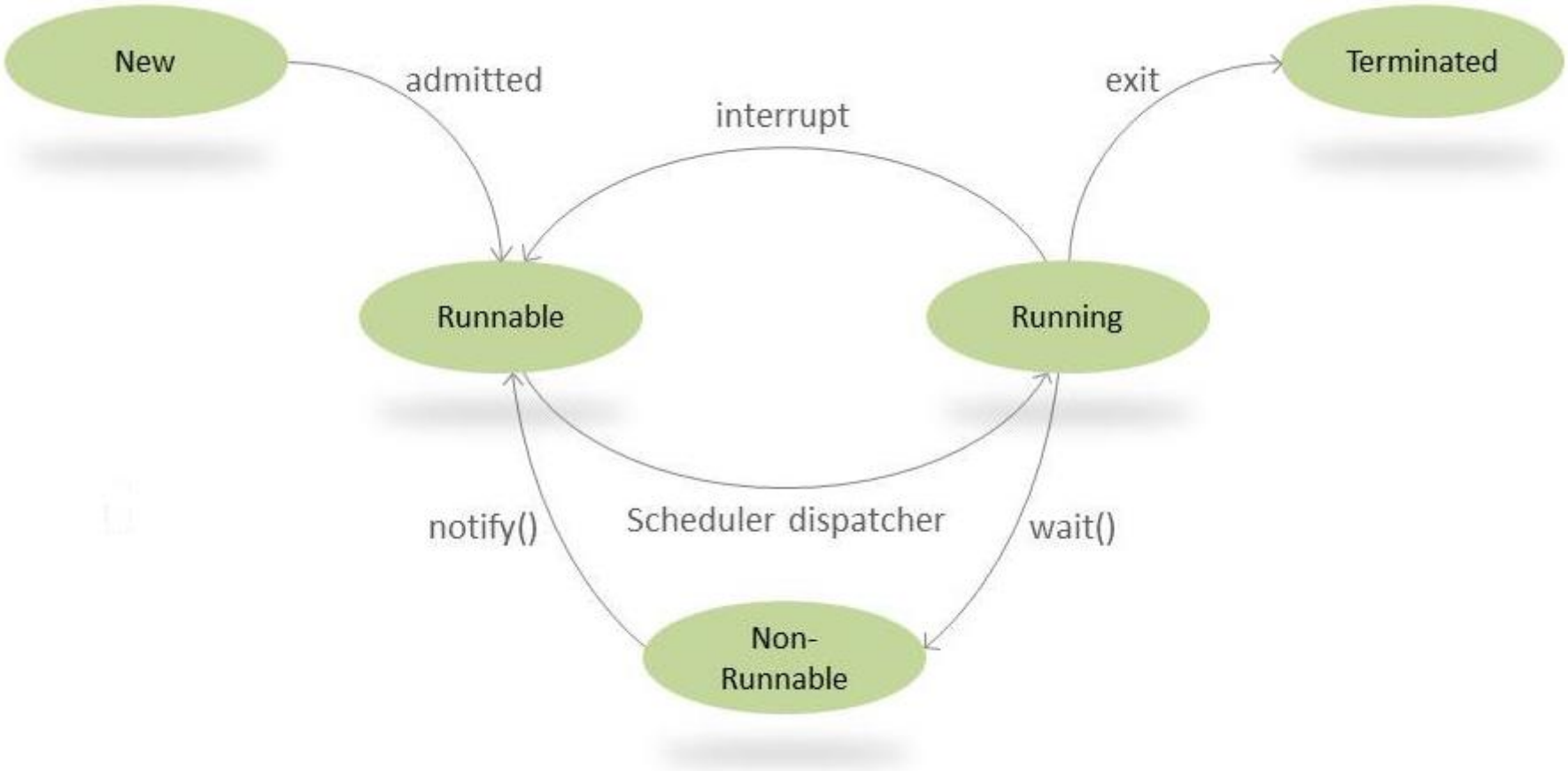
Program for performing two tasks by two threads

```
class Simple1 extends Thread{
 public void run(){
   System.out.println("task one");  }  }
  class Simple2 extends Thread{
 public void run(){
   System.out.println("task two");   }  }
   class TestMultitasking3{
public static void main(String args[]){
 Simple1 t1=new Simple1();
 Simple2 t2=new Simple2();
   t1.start();
   t2.start();   }  }
```

Output:task one
task two

Life Cycle of a Java Thread




New
admitted
interrupt
exit
Terminated
Runnable
Running
notify()
Scheduler dispatcher
wait()
Non-
Runnable

Life Cycle of a Java Thread

- sleep(long ms [,int ns]) // sleep (ms + ns x 10^{-3}) milliseconds and then continue

- [IO] blocked by synchronized method/block
  - synchronized(obj) { … } // synchronized statement
  - synchronized m(…) { … } // synchronized method
  - // return to runnable if IO complete

Life Cycle of a Java Thread

- obj.wait() // retrun to runnable by obj.notify() or obj.notifyAll()

- join(long ms [,int ns]) // Waits at most ms milliseconds plus ns nanoseconds for this thread to die.

Thread Synchronization

- Problem with any multithreaded Java program :
  - Two or more Thread objects access the same pieces of data.
- too little or no synchronization ==> there is inconsistency, loss or corruption of data.
- too much synchronization ==> deadlock or system frozen.
- In between there is unfair processing where several threads can starve another one hogging all resources between themselves

Can Multithreading incur inconsistency?

Multithreading may incur inconsistency : an Example

Two concurrent deposits of 50 into an account with 0 initial balance.:

```
void deposit(int amount) {
   int x = account.getBalance();
   x += amount;
   account.setBalance(x);  }
```

Multithreading may incur inconsistency : an Example

- deposit(50) : // deposit 1

```
x = account.getBalance() //1
x += 50; //2
account.setBalance(x) //3
```

- deposit(50) : // deposit 2

```
x = account.getBalance() //4
x += 50; //5
account.setBalance(x) //6
```

The execution sequence:
1,4,2,5,3,6 will result in unwanted result !!
Final balance is 50 instead of 100!!

Synchronized Methods and Statements

- Multithreading can lead to racing hazards where different orders of interleaving produce different results of computation.
  - Order of interleaving is generally unpredictable and is not determined by the programmer.

Synchronized Methods and Statements

- Java's synchronized method (as well as synchronized statement) can prevent its body from being interleaved by relevant methods.
  - synchronized(obj) { … } // synchronized statement with obj as lock
  - synchronized … m(…) {… } //synchronized method with this as lock
  - When one thread executes (the body of) a synchronized method/statement, all other threads are excluded from executing any synchronized method with the same object as lock.

Inter-thread communication

- 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()

Synchronizing Threads

- Java use the monitor concept to achieve mutual exclusion and synchronization between threads.
- Synchronized methods/statements guarantee mutual exclusion.
  - Mutual exclusion may cause a thread to be unable to complete its task. So monitor allow a thread to wait until state change and then continue its work.

Synchronizing Threads

- wait(), notify() and notifyAll() control the synchronization of threads.
  - Allow one thread to wait for a condition (logical state) and another to set it and then notify waiting threads.
  - condition variables => instance Boolean variables
  - wait => wait();
  - notifying => notify(); notifyAll();

wait() method

- It tells the calling thread to give up the lock and go to sleep until some other thread enters the same monitor and calls notify().

Method	Description
public final void wait()throws InterruptedException	waits until object is notified.
public final void wait(long timeout)throws InterruptedException	waits for the specified amount of time.

notify() method

- Wakes up a single thread that is waiting on this object's monitor.
- If any threads are waiting on this object, one of them is chosen to be awakened.
- The choice is arbitrary and occurs at the discretion of the implementation. Syntax:
- public final void notify()

notifyAll() method

- Wakes up all threads that are waiting on this object's monitor.
- Syntax:
- public final void notifyAll()

Typical Use


```
synchronized void doWhenCondition() {
   while ( !condition )
     wait(); // wait until someone notifies us of
   changes in condition
   … // do what needs to be done when
   condition is true
}
```

Typical Use


```
synchronized void changeCondition {
   // change some values used in condition test
   notify(); // Let waiting threads know
    something changed
}
```

```
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 withdraw...
Less balance; waiting for deposit...
going to deposit...
deposit completed...
withdraw completed

Interrupting a Thread


```
class TestInterruptingThread2 extends Thread{
public void run(){
try{
Thread.sleep(1000);
System.out.println("task");
}catch(InterruptedException e){
System.out.println("Exception handled "+e);  }
System.out.println("thread is running...");  }

public static void main(String args[]){
TestInterruptingThread2 t1=new TestInterruptingThread2();
t1.start();
t1.interrupt(); //to interrupt the sleeping thread }  }
```

```
Exception handled
java.lang.InterruptedException: sleep
interrupted
thread is running...
```

```
7  ********************************************************************
8  class Main extends Thread{
9  public void run(){
10 try{
11 Thread.sleep(1000);
12 System.out.println("task");
13 }catch(InterruptedException e){
14 System.out.println("Exception handled "+e);}
15 System.out.println("thread is running...");}
16
17 public static void main(String args[]){
18 Main t1=new Main();
19 t1.start();
20 //t1.interrupt();
21 }}
22
```

input

```
task
thread is running...


Program finished with exit code 0
```