Multi Threading

By

Apparao G

What is a thread?

Thread Life Cycle

Thread priorities

Two ways of creating Java threads

– Extending Thread class

– Implementing Runnable interface

ThreadGroup

Synchronization

Inter-thread communication

Scheduling a task via Timer and TimerTask

Why threads?

– Need to handle concurrent processes

● Definition

– Single sequential flow of control within a program

– For simplicity, think of threads as processes executed by a program

Example:

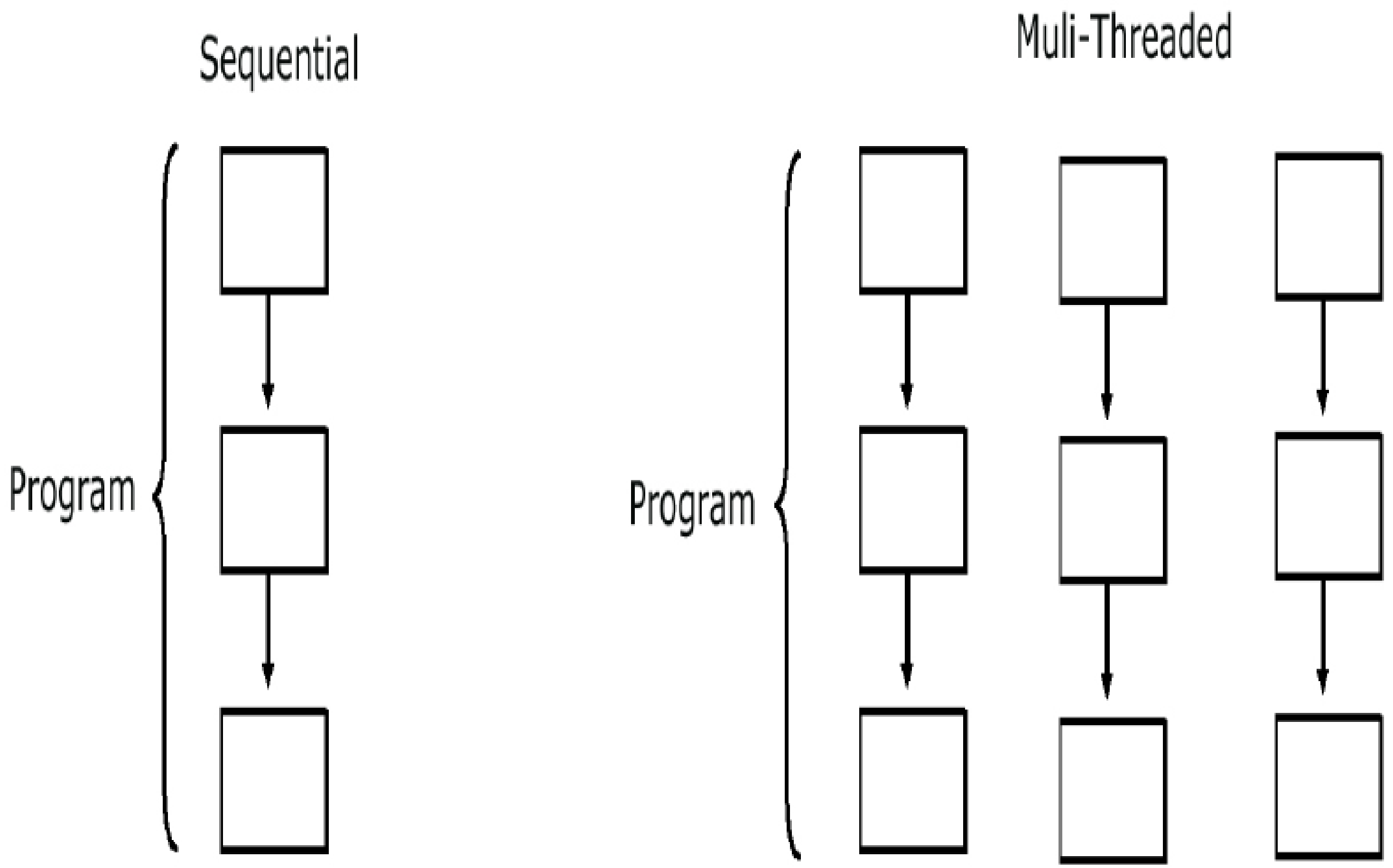
Operating System

HotJava web browser

From the application programmer's point of view, you start with just one thread, called the main thread.

This thread has the ability to create additional threads.

Threads



By definition, multitasking is when multiple processes share common processing resources such as a CPU.

Multi-threading extends the idea of multitasking into applications where you can subdivide specific operations within a single application into individual threads. Each of the threads can run in parallel.

The OS divides processing time not only among different applications, but also among each thread within an application.

Multi-threading enables you to write in a way where multiple activities can proceed concurrently in the same program.

Advantages of Multithreading:

1) We can perform multiple operations at the same time.

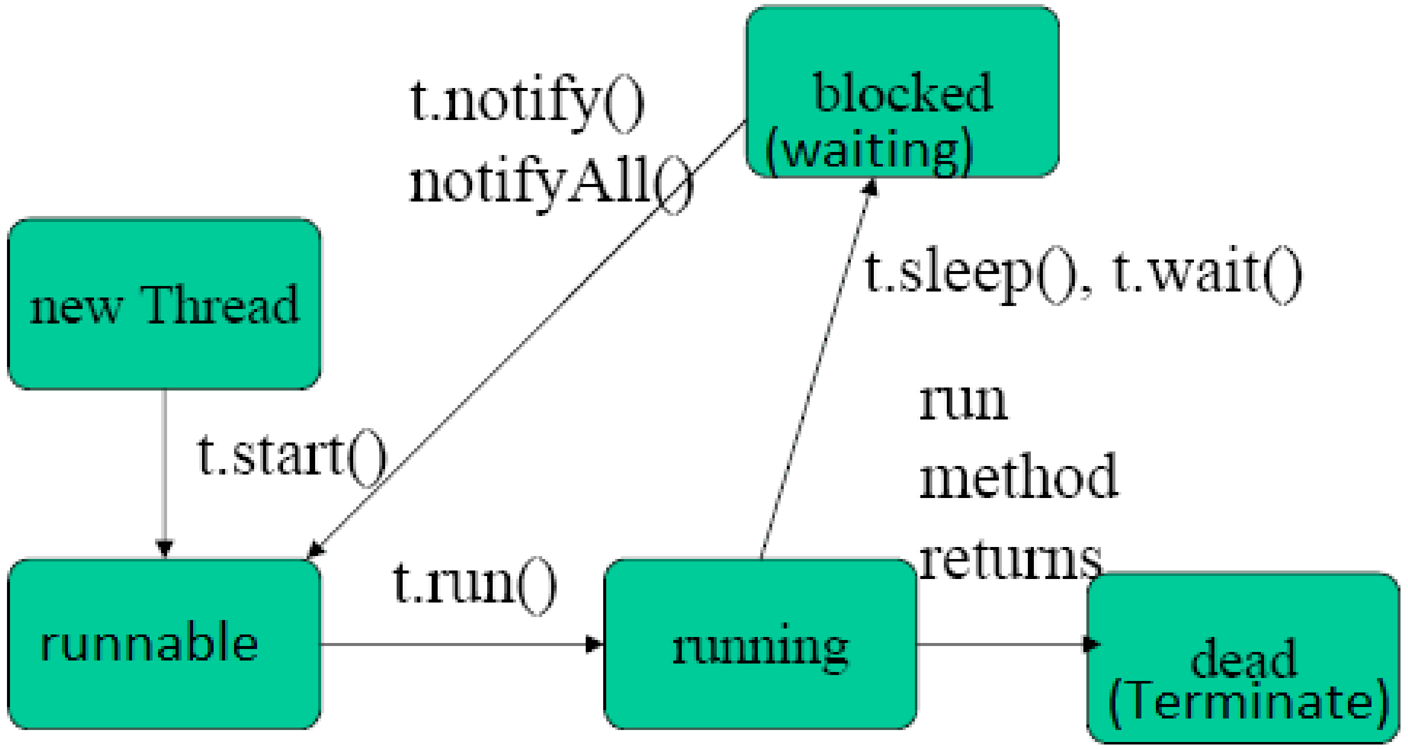
2) We can perform many operations together, so it improves the performance.

3) Threads are independent, so it doesn't affect other threads if an exception occurs in a single thread.

Problems with Threads:

- *Usually very hard to find bugs*
- *Higher cost of code maintenance* since the code inherently becomes harder to reason about
- *Increased utilization of system resources*. Creation of each thread consumes additional memory, CPU cycles for book-keeping and waste of time in context switches.
- *Programs may experience slowdown* as coordination amongst threads comes at a price. Acquiring and releasing locks adds to program execution time. Threads fighting over acquiring locks cause lock contention.

Thread Life Cycle:



New State:

New The thread is in new state if you create an instance of Thread class but before the invocation of start() method.

Runnable:

The thread is in runnable state after invocation of start() method, but the thread scheduler has not selected it to be the running thread.

Running:

The thread is in running state if the thread scheduler has selected it.

Blocked/Waiting:

This is the state when the thread is still alive, but is currently not eligible to run.

Dead/Terminated:

A thread is in terminated or dead state when its run() method completes.

Two Ways of Creating a Thread

1.Extending the *Thread* class

2.Implementing the *Runnable* interface

Extending Thread Class:

1. The subclass extends Thread class

   – The subclass overrides the run() method of Thread class

2. An object instance of the subclass can then be created

3. Calling the start() method of the object instance starts the execution of the thread

   – Java runtime starts the execution of the thread by calling run() method of object instance

Example:

```
public class PrintMessage extends Thread{
    public void run(){
          System.out.println("Printing Hello..");
    }
}
public class PrintMessageTest {
   public static void main(String args[]){
      PrintMessage t1 = new PrintMessage();
      t1.start();
   }
}
```

Constructors of Thread class:

Thread()

Thread(String name)

Thread(Runnable r)

Thread(Runnable r,String name)

Frequently used Thread class methods:

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 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.

public Thread.State getState(): returns the state of the thread.

public boolean isAlive(): tests if the thread is alive.

Runnable Interface:

1. The Runnable interface should be implemented by any class whose instances are intended to be executed as a thread

2. The class must override run() method of no arguments

3. The start() method of the Thread object needs to be explicitly invoked after object instance is created.

Example:

```
public class PrintMessage extends Runnable {
    public void run(){
            System.out.println("Printing Hello message..");
    }
}
public class PrintMessageTest {
   public static void main(String args[]){
      PrintMessage pm = new PrintMessage();
      Thread t1 = new Thread(pm);
      t1.start();
   }
}
```

Extending Thread vs. Implementing Runnable Interface:

Implementing the Runnable interface:

- May take more work since we still
  - Declare a Thread object
  - Call the Thread methods on this object
- Your class can still extend other class

Extending the Thread class:

- Easier to implement
- Your class can no longer extend any other class

Thread Priorities:

Why priorities?

Determine which thread receives CPU control and gets to be executed first.

Definition:

1. Java thread priorities are in the range between MIN_PRIORITY (a constant of 1) and MAX_PRIORITY (a constant of 10). By default, every thread is given priority NORM_PRIORITY (a constant of 5).

2. Higher the thread priority → larger chance of being executed first

Example:

Two threads are ready to run

First thread: priority of 5, already running

Second thread: priority of 10, comes in while first thread is running

Context switch

– Occurs when a thread snatches the control of CPU from another

– When does it occur?

1) Running thread voluntarily relinquishes CPU control

2) Running thread is preempted by a higher priority thread

● More than one highest priority thread that is ready to run

– Deciding which receives CPU control depends on the operating system

– Windows 95/98/NT: Uses time-sliced round-robin

– Solaris: Executing thread should voluntarily relinquish CPU control

In preemptive scheduling, the highest priority task executes until it enters the waiting or dead states or a higher priority task comes into existence.

In time slicing, a task executes for a predefined slice of time and then reenters the pool of ready tasks. The scheduler then determines which task should execute next, based on priority and other factors.

ThreadGroup:
A ThreadGroup represents a set of threads.

A thread group can also include the other thread group.

The thread group creates a tree in which every thread group except the initial thread group has a parent.

A thread is allowed to access information about its own thread group, but it cannot access the information about its thread group's parent thread group or any other thread groups.

Java thread group is implemented by java.lang.ThreadGroup class.

Constructors:

ThreadGroup(String name) : creates a thread group with given name.

ThreadGroup(ThreadGroup parent, String name) : creates a thread group with given parent group and name.

Frequently used Methods:

void checkAccess():This method determines if the currently running thread has permission to modify the thread group.

int activeCount():This method returns an estimate of the number of active threads in the thread group and its subgroups.

int activeGroupCount():This method returns an estimate of the number of active groups in the thread group and its subgroups.

void destroy():This method destroys the thread group and all of its subgroups.

String getName():This method returns the name of the thread group.

ThreadGroup getParent():This method returns the parent of the thread group.

void interrupt():This method interrupts all threads in the thread group.

void list():This method prints information about the thread group to the standard output.

boolean   parentOf(ThreadGroup g):This method tests if the thread group is either the thread group argument or one of its ancestor thread groups.

void suspend():This method is used to suspend all threads in the thread group.

void resume():This method is used to resume all threads in the thread group which was suspended using suspend() method.

void setMaxPriority(int pri):This method sets the maximum priority of the group.

void stop():This method is used to stop all threads in the thread group.

```
public class ThreadGroupTest implements Runnable{
    public void run() {
          System.out.println("Thread Name:"+Thread.currentThread().getName());
    }
   public static void main(String[] args) {
      ThreadGroupDemo runnable = new ThreadGroupDemo();
          ThreadGroup tg1 = new ThreadGroup("Parent ThreadGroup");
          Thread t1 = new Thread(tg1, runnable,"account");
          t1.start();
          Thread t2 = new Thread(tg1, runnable,"order");
          t2.start();
          Thread t3 = new Thread(tg1, runnable,"inventory");
          t3.start();
          System.out.println("Thread Group Name: "+tg1.getName());
         tg1.list();
    }
   }
o/p:
```

Points to remember:

sleep():

The sleep() method of Thread class is used to sleep a thread for the specified amount of time.

syntax:

public static void sleep(long miliseconds)throws InterruptedException

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

Can we start a thread twice?

Answer: No. We cann't start the thread twice. If you start the thread again it will throws IllegalThreadStateException.

What happend if I call run() method directly instead start() method?

Answer: It will not create new thread and executes the run() method on same thread(main thread).

If I call the start() method, it creates new thread and run() method executes on the newly created thread.

Yield() method : What yield() is supposed to do is make the currently running thread head back to runnable to allow other threads of the same priority to get their turn. So the intention is to use yield() to promote graceful turn-taking among equal-priority threads.

A yield() won't ever cause a thread to go to the waiting/sleeping/ blocking state
Signature:
public static void yield() throws InterruptedException;

join( ) Method: The non-static join() method of class Thread lets one thread "join onto the end" of another thread.

If you have a thread B that can't do its work until another thread A has completed its work, then you want thread B to "join" thread A. This means that thread B will not become runnable until A has finished (and entered the dead state).

```
Thread t = new Thread();
 t.start();
 t.join();
```

The preceding code takes the currently running thread (if this were in the main() method, then that would be the main thread) and joins it to the end of the thread referenced by t.

This blocks the current thread from becoming runnable until after the thread referenced by t is no longer alive. In other words, the code t.join() means "Join me (the current thread) to the end of t, so that t must finish before I (the current thread) can run again."

Synchronization:

only one thread can access the resource at a given point in time. This is implemented using a concept called monitors. Each object in Java is associated with a monitor, which a thread can lock or unlock. Only one thread at a time may hold a lock on a monitor.

ATM example with out Synchronization:

```
Public class Account {
 private int balance = 50;
 public int getBalance() {
 return balance;
 }
 public void withdraw(int amount) {
 balance = balance - amount;
 }
}
```

```
public class AccountManager implements Runnable {
 private Account acct = new Account();
  public void run() {
                          for (int x = 0; x < 5; x++) {
                                                  makeWithdrawal(10);
                                                  if (acct.getBalance() < 0) {
                                                                          System.out.println("account is overdrawn!");
                                                  }
                          }
 }
 private void makeWithdrawal(int amt) {
     if (acct.getBalance() >= amt) {
                          System.out.println(Thread.currentThread().getName()+ " is going to withdraw");
                          try {
                                      Thread.sleep(500);
                          } catch(InterruptedException ex) { }
                           acct.withdraw(amt);
                            System.out.println(Thread.currentThread().getName()+ " completes the withdrawal");
       }  else {
                          System.out.println("Not enough in account for "+ Thread.currentThread().getName()+ " to withdraw " + acct.getBalance());
      }
   }
public static void main (String [] args) {
                           AccountManager am = new AccountManager();
                          Thread one = new Thread(am);
                          Thread two = new Thread(am);
                          one.setName(“Bob");
                          two.setName(“Tom");
                          one.start();
                          two.start();
  }
}
```

Run the above example and see the output . Is it possible that, say, Bob checked the balance, fell asleep, Tom checked the balance, Bob woke up and completed his withdrawal, then Tom completes his withdrawal, and in the end they overdraw the account.

Output:

```
Bob is going to withdraw
Tom is going to withdraw
Bob completes the withdrawal
Bob is going to withdraw
Tom completes the withdrawal
Tom is going to withdraw
Bob completes the withdrawal
Bob is going to withdraw
Tom completes the withdrawal
Tom is going to withdraw
Tom completes the withdrawal
account is overdrawn!
Bob completes the withdrawal
account is overdrawn!
Not enough in account for Bob to withdraw -10
account is overdrawn!
Not enough in account for Bob to withdraw -10
account is overdrawn!
Not enough in account for Tom to withdraw -10
account is overdrawn!
Not enough in account for Tom to withdraw -10
account is overdrawn!
```

How to protect the data?

You must do two things:

- Mark the variables private.
- Synchronize the code that modifies the variables.

To fix the above problem,We mark the makeWithdrawal() method synchronized as follows:

```
private synchronized void makeWithdrawal(int amt) {
     if (acct.getBalance() >= amt) {
                System.out.println(Thread.currentThread().getName()+ " is going to withdraw");
                try {
                          Thread.sleep(500);
                } catch(InterruptedException ex) { }
                 acct.withdraw(amt);
                  System.out.println(Thread.currentThread().getName()+ " completes the withdrawal");
       }  else {
                System.out.println("Not enough in account for "+ Thread.currentThread().getName()+ " to withdraw
" + acct.getBalance());
      }
   }
```

Remember the following key points about locking and synchronization:

■ Only methods (or blocks) can be synchronized, not variables or classes.

■ Each object has just one lock.

■ Not all methods in a class need to be synchronized. A class can have both synchronized and non-synchronized methods.

■ If two threads are about to execute a synchronized method in a class, and both threads are using the same instance of the class to invoke the method, only one thread at a time will be able to execute the method. The other thread will need to wait until the first one finishes its method call. In other words, once a thread acquires the lock on an object, no other thread can enter any of the synchronized methods in that class (for that object).

■ If a class has both synchronized and non-synchronized methods, multiple threads can still access the class's non-synchronized methods! If you have methods that don't access the data you're trying to protect, then you don't need to synchronize them. Synchronization can cause a hit in some cases (or even deadlock if used incorrectly), so you should be careful not to overuse it.

■ If a thread goes to sleep, it holds any locks it has—it doesn't release them.
■ A thread can acquire more than one lock. For example, a thread can enter a synchronized method, thus acquiring a lock, and then immediately invoke a synchronized method on a different object, thus acquiring that lock as well. As the stack unwinds, locks are released again. Also, if a thread acquires a lock and then attempts to call a synchronized method on that same object, no problem. The JVM knows that this thread already has the lock for this object, so the thread is free to call other synchronized methods on the same object, using the lock the thread already has.

■ You can synchronize a block of code rather than a method.

synchronized block:

We can use synchronization for specified number of lines (or block of statements) instead of entire method.

In this case we will used synchronized block.

```
synchronized(objectidentifier) {
   // critical section code or shared variables and shared methods.
}
```

Example:

```
class SyncTest {
 public void doStuff() {
   System.out.println("not synchronized");
   synchronized(this) {
   System.out.println("synchronized");
   }
 }
}
```

Note: When you synchronize a method, the object used to invoke the method is the
object whose lock must be acquired.
But when you synchronize a block of code, you specify which object's lock you want to use as the lock.

static synchronization:

static methods can be synchronized. There is only one copy of the static data you're trying to protect, so you only need one lock per class to synchronize static methods—a lock for the whole class.

There is such a lock; every class loaded in Java has a corresponding instance of java.lang.Class representing that class. It's that java.lang.Class instance whose lock is used to protect the static methods of the class (if they're synchronized).

```
public MyClass {
    public static synchronized int getCount() {
       return count;
    }
}
```

This could be replaced with code that uses a synchronized block. If the method is defined in a class called MyClass, the equivalent code is as follows:

```
public static int getCount() {
 synchronized(MyClass.class) {
 return count;
 }
}
```

Deadlock: a situation where two or more threads are blocked forever, waiting for each other. Deadlock occurs when multiple threads need the same locks but obtain them in different order.

A Java multithreaded program may suffer from the deadlock condition because the **synchronized** keyword causes the executing thread to block while waiting for the lock, or monitor, associated with the specified object

Example:

```
public class DeadLockTest {
        public static Object lock1 = new Object();
        public static Object lock2 = new Object();
```

```
private static class PrinterOne extends Thread {
              public void run() {
                                       synchronized (lock1) {
                                                   System.out.println("Thread 1: Holding lock 1...");
                                                   try {
                                                     Thread.sleep(10);
                                                   } catch (InterruptedException e) {}
                                                   System.out.println("Thread 1: Waiting for lock 2...");
                                                   synchronized (Lock2) {
                                                      System.out.println("Thread 1: Holding lock 1 & 2...");
             }            }            }            }
private static class PrinterTwo extends Thread {
                          public void run() {
                                       synchronized (lock2) {
                                                   System.out.println("Thread 2: Holding lock 2...");
                                                   try {
                                                     Thread.sleep(10);
                                                   } catch (InterruptedException e) {}
                                                   System.out.println("Thread 2: Waiting for lock 1...");
                                                   synchronized (Lock1) {
                                                               System.out.println("Thread 2: Holding lock 1 & 2...");
                          }       }             }
             }
```

```
public static void main(String args[]) {
                    PrinterOne t1 = new PrinterOne();
                    PrinterTwo t2 = new PrinterTwo();
                    t1.start();
                    t2.start();
          }
}
```

Output:

```
Thread 1: Holding lock 1...
Thread 2: Holding lock 2...
Thread 2: Waiting for lock 1...
Thread 1: Waiting for lock 2...
```

Note: To resolve the dead lock issue, we need to change the order of the lock.
Run the above example and see the output, it will not create the dead lock.

Inter thread communication:

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.

The Object class has three methods, wait(), notify(), and notifyAll() that help threads communicate about the status of an event that the threads care about.

For example,

if one thread is a mail-delivery thread and one thread is a mail-processor thread,

the mail-processor thread has to keep checking to see if there's any mail to process.

Using the wait and notify mechanism, the mail-processor thread could check for mail, and if it doesn't find any it can say, "Hey, I'm not going to waste my time checking for mail every two seconds. I'm going to go hang out, and when the mail deliverer puts something in the mailbox, have him notify me so I can go back to runnable and do some work."

wait(), notify(), and notifyAll() must be called from within a synchronized context! A thread can't invoke a wait or notify method on an object unless it owns that object's lock.

wait() method:

Causes current thread to release the lock and wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.

Signature:

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.

Signature:

public final void notify()

notifyAll() method:

Wakes up all threads that are waiting on this object's monitor.

Signature:

public final void notifyAll()

```
public class CustomerAccount {
        int amount = 8000;
        synchronized void withdraw(int amount) {
                System.out.println("Going to withdraw amount::"+amount);

                if (this.amount < amount) {
                        System.out.println("Insufficient balance you have only::"+this.amount +"  waiting for deposit...");
                        try {
                                wait();
                        } catch (Exception e) {
                        }
                }
                this.amount -= amount;
                System.out.println(amount +"  withdraw and current account balance is::"+this.amount);
        }
        synchronized void deposit(int amount) {
                System.out.println("going to deposit with amount::"+amount);
                this.amount += amount;
                System.out.println(amount+"  deposited and current account balance is:: "+this.amount);
                notify();
        }
}
```

```
public class InterProcessCommunicationOfThreads {
        public static void main(String args[]) {
                final CustomerAccount c = new CustomerAccount();
                new Thread() {
                        public void run() {
                                c.withdraw(20000);
                        }
                }.start();

                new Thread() {
                        public void run() {
                                c.deposit(15000);
                        }
                }.start();
        }
}
```

Output:

```
Going to withdraw amount::20000
Insufficient balance you have only::8000  waiting for deposit...
going to deposit with amount::15000
15000  deposited and current account balance is:: 23000
20000  withdraw and current account balance is::3000
```

Exception handling using UncaughtExceptionHandler:

Java provides us with a mechanism to catch and treat the unchecked exceptions thrown in a Thread instance to avoid the program crashing. This can be done using UncaughtExceptionHandler.

UncaughtExceptionHandler:

- UncaughtExceptionHandler helps you to run a thread in a way such that it will run until it’s task is done.
- UncaughtExceptionHandler can be used for making logging more robust only as well without restarting the thread because often default logs don’t provide enough information about the context when thread execution failed.

Example:

```
class Task implements Runnable
{
  @Override
  public void run()
  {
    System.out.println(Integer.parseInt("222"));
    System.out.println(Integer.parseInt("333"));
    System.out.println(Integer.parseInt("abc")); //This will cause NumberFormatException
    System.out.println(Integer.parseInt("444"));
  }
}
```

```
public class ThreadTest {
        public static void main(String[] args) {
                Task task = new Task();
                Thread thread = new Thread(task);
                thread.start();
        }
}
output:
222
333
Exception in thread "Thread-0" java.lang.NumberFormatException: For input string: "abc"
        at java.lang.NumberFormatException.forInputString(Unknown Source)
        at java.lang.Integer.parseInt(Unknown Source)
        at java.lang.Integer.parseInt(Unknown Source)
        at com.test.synchronization.Task.run(Task.java:8)
        at java.lang.Thread.run(Unknown Source)
```

With UncaughtExceptionHandler:

Let’s add one UncaughtExceptionHandler implementation to catch any unchecked exception during runtime.

```
public class ExceptionHandler implements UncaughtExceptionHandler
{
  public void uncaughtException(Thread t, Throwable e)
  {
    System.out.printf("An exception has been captured\n");
    System.out.printf("Thread: %s\n", t.getId());
    System.out.printf("Exception: %s: %s\n", e.getClass().getName(), e.getMessage());
    System.out.printf("Stack Trace: \n");
    e.printStackTrace(System.out);
    System.out.printf("Thread status: %s\n", t.getState());
   }
}
```

Now add this exception handler to the thread.

```
class Task implements Runnable
{
   @Override
   public void run()
   {
      Thread.currentThread().setUncaughtExceptionHandler(new ExceptionHandler());
      System.out.println(Integer.parseInt("222"));
      System.out.println(Integer.parseInt("333"));
      System.out.println(Integer.parseInt("abc")); //This will cause NumberFormatException
      System.out.println(Integer.parseInt("444"));
   }
}
```

output:

```
222
333
An exception has been captured
Thread: 11
Exception: java.lang.NumberFormatException: For input string: "abc"
Stack Trace:
java.lang.NumberFormatException: For input string: "abc"
                    at java.lang.NumberFormatException.forInputString(Unknown Source)
                    at java.lang.Integer.parseInt(Unknown Source)
                    at java.lang.Integer.parseInt(Unknown Source)
                    at com.test.synchronization.Task.run(Task.java:9)
                    at java.lang.Thread.run(Unknown Source)
Thread status: RUNNABLE
```