Q) Producer shud be adding elements and consumer shud be removing from the list parallely. But the max no. of the elements in list shud not exceed 5.

Q) object is mutable on which i am acquiring the lock? state of the object is changed, it will not be able to release the lock and the other threads

class Task implements Runnable{

Object lock1= new Object();

Employee obj; //heap : shared by all threads

Task(Employee obj){

this.obj= obj;

}

void add(){

int x=0; //stack : thread

x++;

Sysout("Entering"); //stack :each thread has it's own stack

synchronized(lock1){ //should not acquire lock on mutable, null, string literals, constants..

System.out.println(e);//corrupt data/ wrong value// dirty write list.add(): 2 ;3 5

lock.wait()

}

}

void remove(){

synchronized(Object.class){

//should not acquire lock on mutable, null, string literals, constants..

System.out.println(e);//shared object : block preferred over synchronized method (20 lines of code)

}

}

class ABC{

void synchronized add(){

}

void synchronized remove(){

}

}

ABC obj1, obj2

locking a room: key u r using to lock it. Someone changes the lock

String s1= "Hello";

String s2= "Hello";

list | removing

add remove

synchronized(s1)

vector | hashtable: performance

Lock interface

1) We should not acquire lock on mutable objects, null values, string literals, constants

2) In synchronized, synchronized block is preferred over synchronized method

3) Synchronized block, we can acquire either a object lock or a class lock.

synchronized(obj)

synchronized(Object.class)

4)if we are using synchronized methods, how do we acquire a class lock?

public synchronized void add(){ //lock on current object

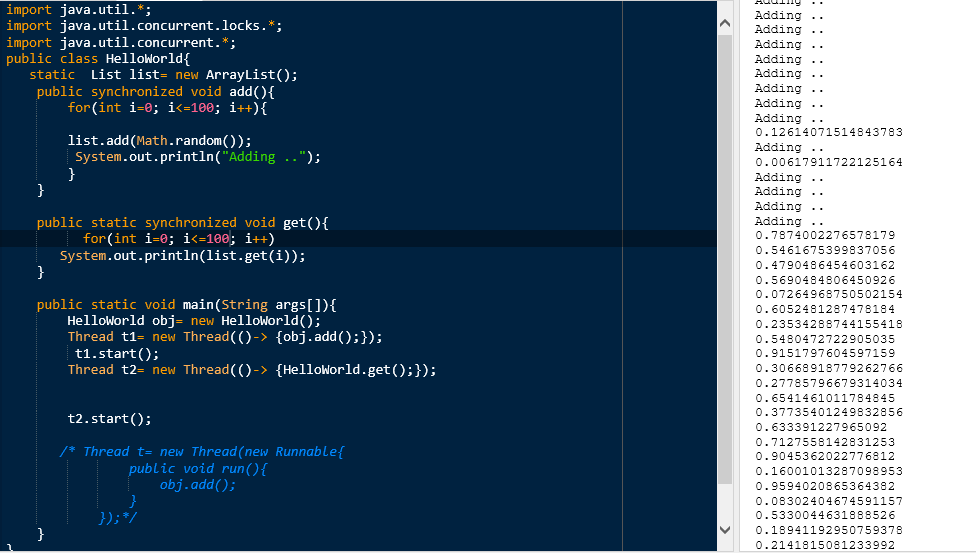
}

public static synchronized void add(){ //lock on current class

}

These would be running in parallel, bcoz one is object lock, other is class lock

Lock, Threadgroup, threadpool



If we acquire a class lock, that means no other thread would be

able to acquire lock on any object of that class.

Problems with synchronization:

1. Deadlock
2. Performance slowed down: Threads are waiting endlessly to get the lock
3. Scheduler which decides which thread to execute first :

OS: logic to schedule threads: priority based / first come first served/ round robin/ slowest task first.

Priority: thread 1: min priority (starvation)

Threads: 5 (normal priority)

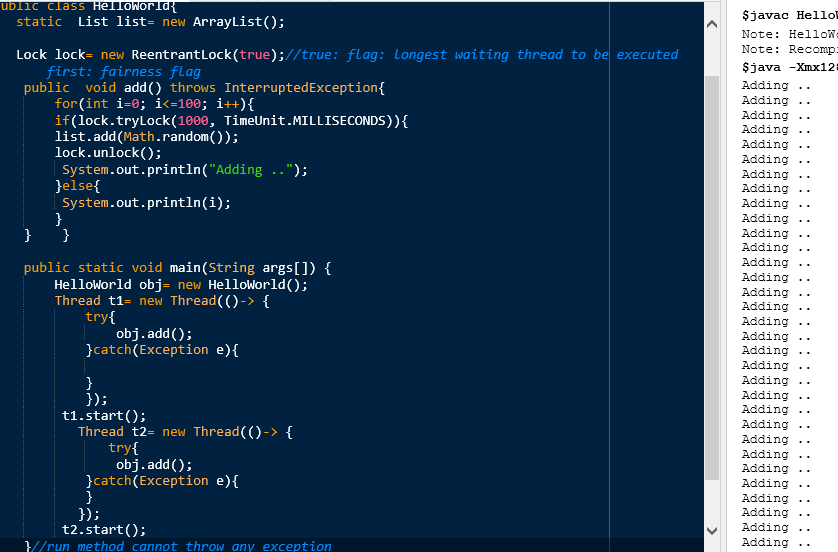
1. Synchronized over a block/ method. Cannot use it across methods

Lock interface

1. Lock()
2. Unlock()

Synchronized block:

1. tryLock()



Using lock across methods:

import java.util.\*;

import java.util.concurrent.locks.\*;

import java.util.concurrent.\*;

public class HelloWorld{

static List list= new ArrayList();

Lock lock= new ReentrantLock(true);//true: flag: longest waiting thread to be executed first: fairness flag

public void add() throws InterruptedException{

for(int i=0; i<=100; i++){

if(lock.tryLock(1000, TimeUnit.MILLISECONDS)){

list.add(Math.random());

System.out.println("Not Adding .."+ get(0));

}else{

System.out.println(i);

} } }

public Object get(int i){

Object o= list.get(i);

lock.unlock();

return o;

}

public static void main(String args[]) {

HelloWorld obj= new HelloWorld();

Thread t1= new Thread(()-> {

try{

obj.add();

}catch(Exception e){

}

});

t1.start();

Thread t2= new Thread(()-> {

try{

obj.add();

}catch(Exception e){

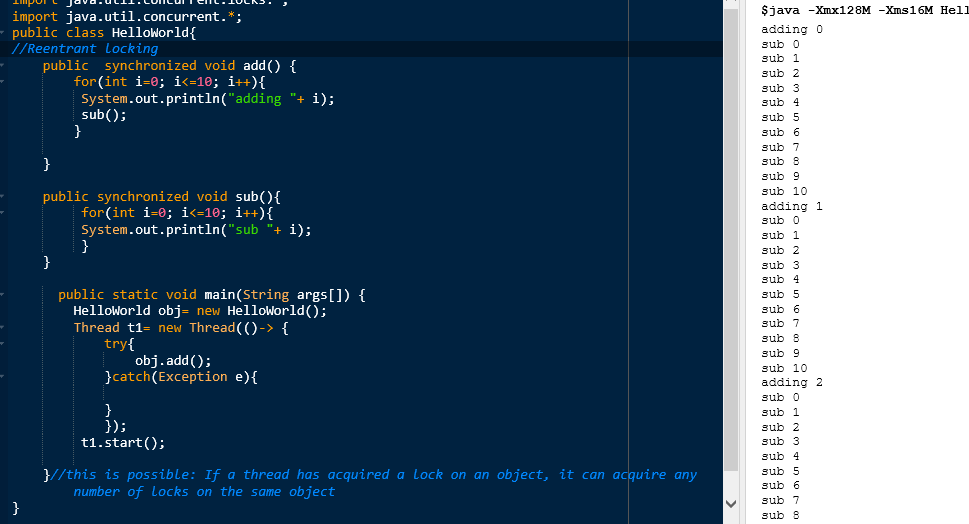
}

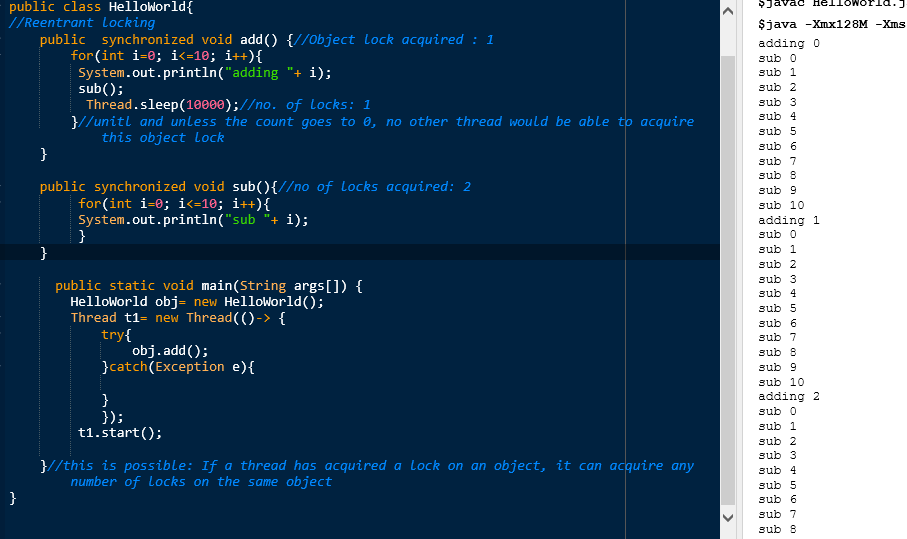
});

t2.start();

}//run method cannot throw any exception

}





If i call lock.unlock() and the current thread is not holding a lock, it will throw IllegalMonitorStateException

1. getHoldCount()
2. isFair()
3. ....

Synchronized, lock, re-entrant lock class, volatile, join, sleep, wait, notify, notify all

Executor framework

Q) MyList class: locks/ synchronized to make list as synchronized

MyList :

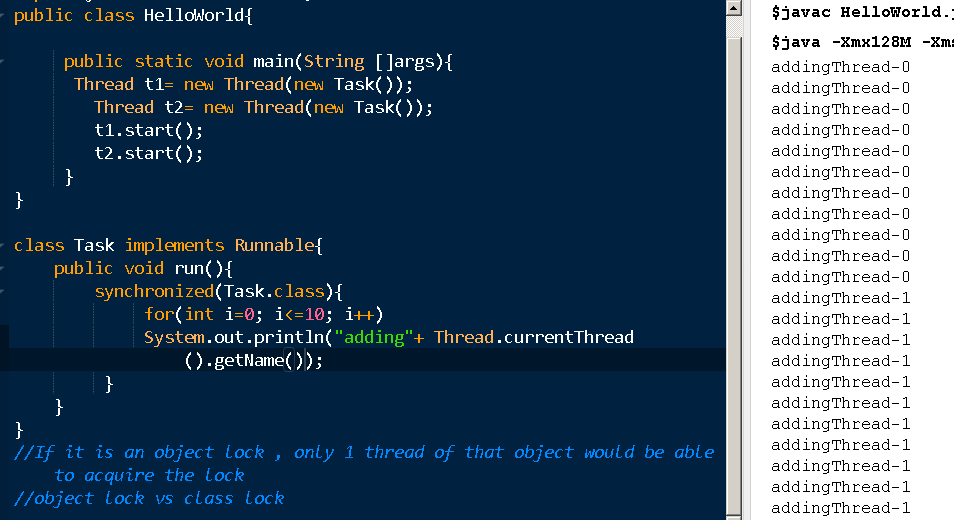
List list

Add(){

List.add();  
}

//remove

Get(): any no. Of threads can call it.



Connection Pool (Datasource): Database: 1000 users: connection pool: no. Of open connections: 10 open connections. Wait time for ur request: 1000 millisec 11th

/ Webserver: handle ur web requests (60 requests) : Pool of open threads

Thread pools: job is queued and when a thread is available, it will serve your request

Thread[] threads= new Thread[10]; //max 10 threads are available to serve your request

Tasks : 20 tasks :

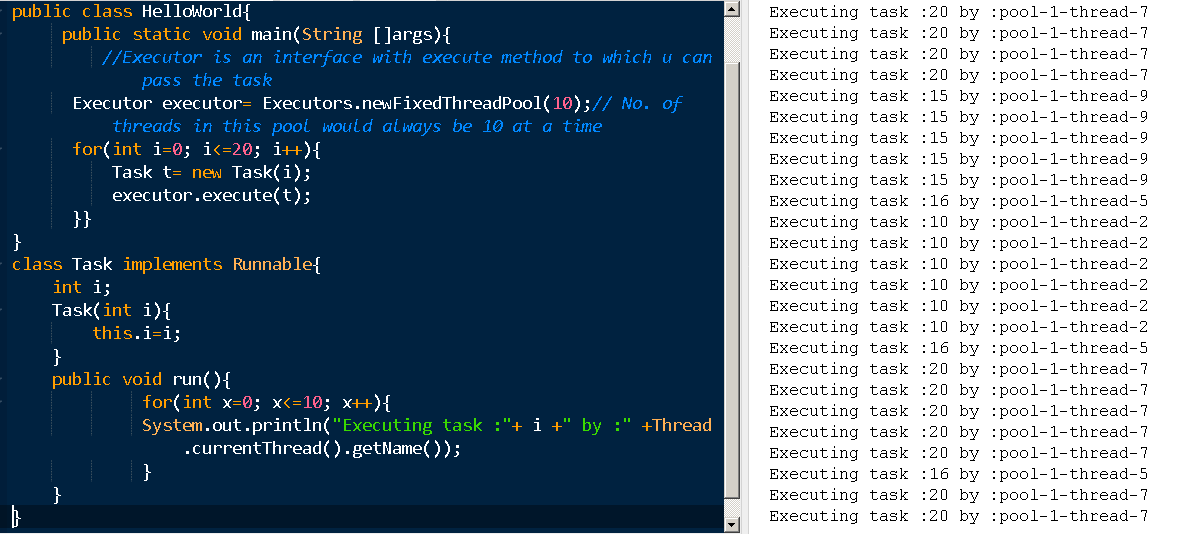
Thread: run() -> completes it, thread is dead. You cannot restart the dead thread. So u hv to create a new thread

Executor framework: to create thread pools. U don’t have to worry about managing the thread lifecycle and assigning tasks to them.

You can submit all tasks to the pool and automatically threads would be managed by this framework

Q) Create a thread array of 10 in size. They should be doing 20 tasks. If 1 thread is done, u can create new thread to handle other task

//Thread pool

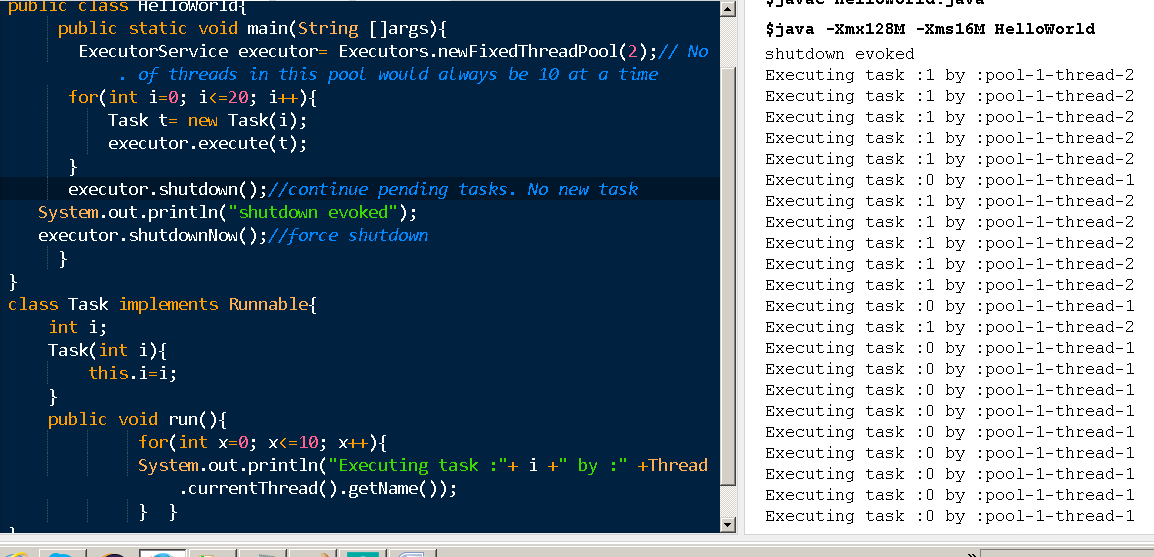


Executors class: various kinds of thread pool

1. Fixed Thread pool: No. Of thread are constant
2. Cached Thread pool

Executor: execute()

ExecutorService interface: extends Executor

1. Execute()
2. Submit()
3. Shutdown(): that no more tasks can be accepted after shutdown is called. If u try to submit a new task after shutdown, it throws RejectedExecutionException. But if tasks are running, they can continue execution
4. shutdownNow(): If any tasks are running, force termination
5. 

Eg:

import java.util.concurrent.\*;

//Thread pool

public class HelloWorld{

public static void main(String []args) throws Exception{

ExecutorService executor= Executors.newFixedThreadPool(2);// No. of threads in this pool would always be 10 at a time

Future[] future= new Future[20];

for(int i=0; i<20; i++){

Task t= new Task();

future[i]=executor.submit(t);

}

for(int i=0; i<20; i++){

System.out.println(future[i].get());//blocking

}

}

}

class Task implements Callable{

public Object call(){

int sum=0;

for(int i=0; i<=10; i++){

sum=sum+i;

}

System.out.println("Executed by :"+ Thread.currentThread().getName());

return new Integer(sum);

}

}

Future

1. Get(): return result: object. If the task throws an exception: callable is throwing exception.
2. Get(): if called on runnable task, returns null

import java.util.concurrent.\*;

import java.util.\*;

//Thread pool

public class HelloWorld{

public static void main(String []args) {

ExecutorService executor= Executors.newFixedThreadPool(2);// No. of threads in this pool would always be 10 at a time

Future[] future= new Future[20];

for(int i=0; i<20; i++){

Task2 t= new Task2( i);

future[i]=executor.submit(t);

}

for(int i=0; i<20; i++){

try{

System.out.println(future[i].get(1000, TimeUnit.MILLISECONDS));//blocking

}catch(Exception e){}

}

}

}

class Task implements Callable{

int i;

Task(int i){

this.i=i;

}

public Object call() throws Exception{

System.out.println("Executed by :"+ Thread.currentThread().getName());

if(i==0){

throw new Exception();

}

return i;

}

}

class Task2 implements Runnable{

int i;

Task2(int i){

this.i=i;

}

public void run() {

System.out.println("Executed by :"+ Thread.currentThread().getName());

}

}

Executors.newCachedThreadPool(): No upper limit on number of threads to be exceuted. We use it for short lived tasks.

Executors.newSingleThreadPool();: single thread serving request

import java.util.concurrent.\*;

import java.util.\*;

//Thread pool

public class HelloWorld{

public static void main(String []args) {

ExecutorService executor= Executors.newSingleThreadExecutor();

Future[] future= new Future[200];

for(int i=0; i<200; i++){

Task2 t= new Task2( i);

future[i]=executor.submit(t);

}

for(int i=0; i<200; i++){

try{

System.out.println(future[i].get(1000, TimeUnit.MILLISECONDS));//blocking

}catch(Exception e){}

}

}

}

class Task implements Callable{

int i;

Task(int i){

this.i=i;

}

public Object call() throws Exception{

System.out.println("Executed by :"+ Thread.currentThread().getName());

if(i==0){

throw new Exception();

}

return i;

}

}

class Task2 implements Runnable{

int i;

Task2(int i){

this.i=i;

}

public void run() {

System.out.println("Executed by :"+ Thread.currentThread().getName());

}

}

newScheduledThreadExecutor: run the task after a fixed delay or to run tasks after a specific period

T1: 3sec: period: 3sec

T2: 1 sec..2 sec

T3: 0.5 sec

Executor, ExecutorService, Executors,

invokeAny(), invokeAll()