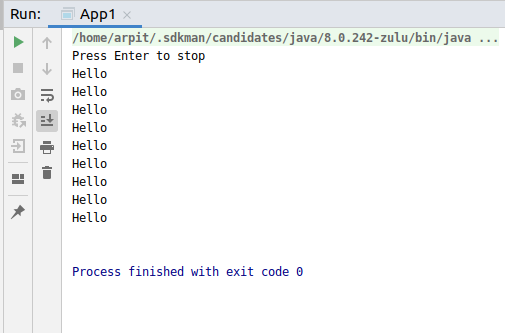
Basics of Multi Threading

Exercise

1. Write a program do to demonstrate the use of volatile keyword.

Ans. In this program I have used a volatile variable so that the variable value can be changed to false to end the loop. Because in some implementations of java a if we don’t declare the variable volatile then java will only cache its value at compile time and not check it afterwards with its real value at runtime.



Code:

**package** MultiThread1;

**import** java.util.Scanner;

**class** Process **extends** Thread{

**private volatile boolean runing** = **true**;

**public void** run() {

**while** (**runing**){

System.***out***.println(**"Hello"**);

**try** {

Thread.*sleep*(100);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

**public void** shutdown(){

**runing**=**false**;

}

}

**public class** App1 {

**public static void** main(String[] args) {

Process process1 = **new** Process();

process1.start();

System.***out***.println(**"Press Enter to stop"**);

Scanner scanner =**new** Scanner(System.***in***);

scanner.nextLine();

process1.shutdown();

}

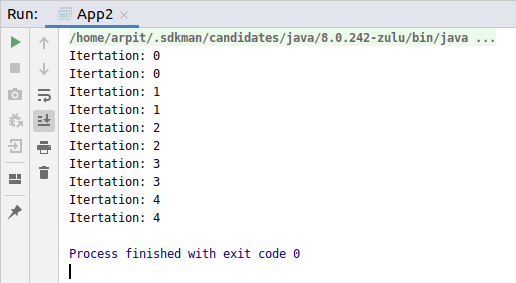
}

1. Write a program to create a thread using Thread class and Runnable interface each.

Ans. In this program I have created two threads.

t1 thread is created using a class Runner which implements the interface Runnable.

t2 thread is created using a class Thread1 which extends the class Thread.



Code:

**package** MultiThread1;

**class** Runner **implements** Runnable{

@Override

**public void** run() {

**for** (**int** i=0;i<5;i++){

System.***out***.println(**"Itertation: "**+ i);

**try** {

Thread.*sleep*(100);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

}

**class** Thread1 **extends** Thread{

@Override

**public void** run() {

**for** (**int** i=0;i<5;i++){

System.***out***.println(**"Itertation: "**+ i);

**try** {

Thread.*sleep*(100);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

}

**public class** App2 {

**public static void** main(String[] args) {

Thread t1 = **new** Thread(**new** Runner());

Thread1 t2 = **new** Thread1();

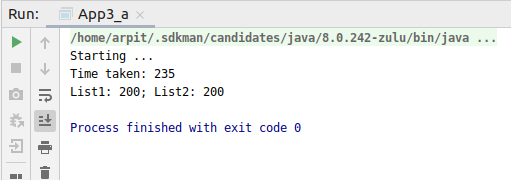
t1.start();

t2.start();

}

}

1. Write a program using synchronization block and synchronization method
2. This is using Synchronized block.



Code:

**package** MultiThread1;

**import** java.util.ArrayList;

**import** java.util.List;

**import** java.util.Random;

**public class** App3\_a {

**private** Random **random** = **new** Random();

**private** Object **lock1** = **new** Object();

**private** Object **lock2** = **new** Object();

**private** List<Integer> **list1** = **new** ArrayList<Integer>();

**private** List<Integer> **list2** = **new** ArrayList<Integer>();

**public void** stageOne(){

**synchronized** (**lock1**){

**try** {

Thread.*sleep*(1);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

**list1**.add(**random**.nextInt(100));

}

}

**public void** stageTwo(){

**synchronized** (**lock2**){

**try** {

Thread.*sleep*(1);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

**list2**.add(**random**.nextInt(100));

}

}

**public void** process(){

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

stageOne();

stageTwo();

}

}

**public void** myThread() {

System.***out***.println(**"Starting ..."**);

**long** start = System.*currentTimeMillis*();

Thread t1 = **new** Thread(**new** Runnable() {

**public void** run() {

process();

}

});

Thread t2 = **new** Thread(**new** Runnable() {

**public void** run() {

process();

}

});

t1.start();

t2.start();

**try** {

t1.join();

t2.join();

} **catch** (InterruptedException e) {

e.printStackTrace();

}

**long** end = System.*currentTimeMillis*();

System.***out***.println(**"Time taken: "** + (end - start));

System.***out***.println(**"List1: "** + **list1**.size() + **"; List2: "**

+ **list2**.size());

}

**public static void** main(String[] args) {

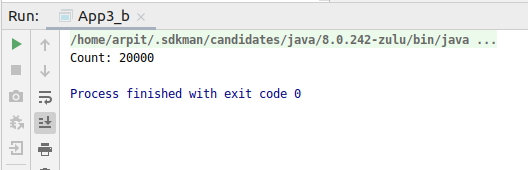
App3\_a obj = **new** App3\_a();

obj.myThread();

}

}

1. This is using Synchronized method



Code:-

**package** MultiThread1;

**public class** App3\_b **extends** Thread{

**private int count** = 0;

**public synchronized void** increment(){

**count**++;

}

**public static void** main(String[] args) {

App3\_b SyncObj =**new** App3\_b();

SyncObj.doWork();

}

**private void** doWork() {

Thread t1 =**new** Thread(**new** Runnable() {

@Override

**public void** run() {

**for** (**int** i=0;i<10000;i++){

increment();

}

}

});

Thread t2 = **new** Thread(**new** Runnable() {

@Override

**public void** run() {

**for** (**int** i=0;i<10000;i++){

increment();

}

}

});

t1.start();

t2.start();

**try** {

t1.join();

t2.join();

} **catch** (InterruptedException e) {

e.printStackTrace();

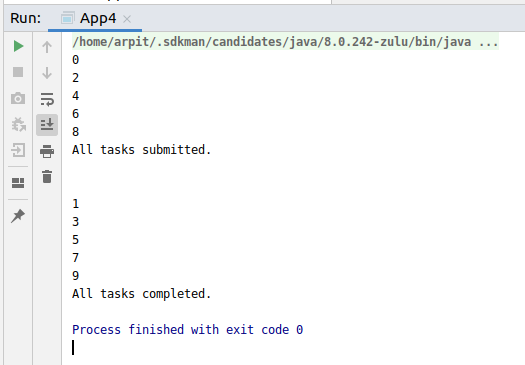
}

System.***out***.println(**"Count: "**+**count**);

}

}

1. Write a program to create a Thread pool of 2 threads where one Thread will print even numbers and other will print odd numbers.



Code:

**package** MultiThread1;

**import** java.util.concurrent.ExecutorService;

**import** java.util.concurrent.Executors;

**import** java.util.concurrent.TimeUnit;

**class** Proc **implements** Runnable{

**public void** run() {

**for** (**int** i=0; i<10;i=i+2){

System.***out***.println(i);

}

**try** {

Thread.*sleep*(500);

} **catch** (InterruptedException e) {

}

System.***out***.println(**"\n"**);

**for** (**int** i=1;i<10;i=i+2){

System.***out***.println(i);

}

}

}

**public class** App4 {

**public static void** main(String[] args) {

ExecutorService executor = Executors.*newFixedThreadPool*(2);

executor.submit(**new** Proc());

executor.shutdown();

System.***out***.println(**"All tasks submitted."**);

**try** {

executor.awaitTermination(1, TimeUnit.***DAYS***);

} **catch** (InterruptedException e) {

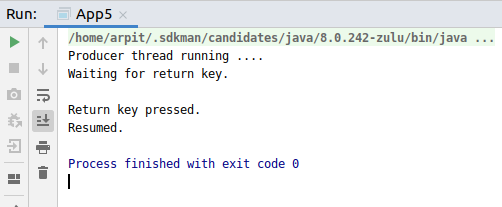
}

System.***out***.println(**"All tasks completed."**);

}

}

1. Write a program to demonstrate wait and notify methods.



Code:

**package** MultiThread1;

**import** java.util.Scanner;

**class** Myclass {

**public void** produce() **throws** InterruptedException {

**synchronized** (**this**) {

System.***out***.println(**"Producer thread running ...."**);

wait();

System.***out***.println(**"Resumed."**);

}

}

**public void** consumer() **throws** InterruptedException{

Scanner scanner = **new** Scanner(System.***in***);

Thread.*sleep*(2000);

**synchronized** (**this**) {

System.***out***.println(**"Waiting for return key."**);

scanner.nextLine();

System.***out***.println(**"Return key pressed."**);

notify();

Thread.*sleep*(5000);

}

}

}

**public class** App5 {

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

**final** Myclass obj = **new** Myclass();

Thread t1 = **new** Thread(**new** Runnable() {

@Override

**public void** run() {

**try** {

obj.produce();

}**catch** (InterruptedException e){

}

}

});

Thread t2 = **new** Thread(**new** Runnable() {

@Override

**public void** run() {

**try** {

obj.consumer();

}**catch** (InterruptedException e){

}

}

});

t1.start();

t2.start();

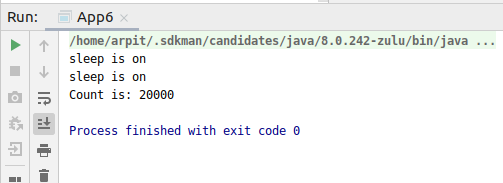
t1.join();

t2.join();

}

}

1. Write a program to demonstrate sleep and join methods.



Code:

**package** MultiThread1;

**public class** App6 **implements** Runnable {

**private int count** = 0;

**public void** run() {

Thread thread1 = **new** Thread(**new** Runnable() {

**public void** run() {

**for**(**int** i = 0; i < 10000; i++) {

**count**++;

}

**try** {

Thread.*sleep*(100);

System.***out***.println(**"sleep is on"**);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

});

thread1.start();

Thread thread2 = **new** Thread(**new** Runnable() {

**public void** run() {

**for**(**int** i = 0; i < 10000; i++) {

**count**++;

}

**try** {

Thread.*sleep*(100);

System.***out***.println(**"sleep is on"**);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

});

thread2.start();

**try** {

thread1.join();

thread2.join();

} **catch** (InterruptedException e) {

e.printStackTrace();

}

System.***out***.println(**"Count is: "** + **count**);

}

**public static void** main(String[] args)

{

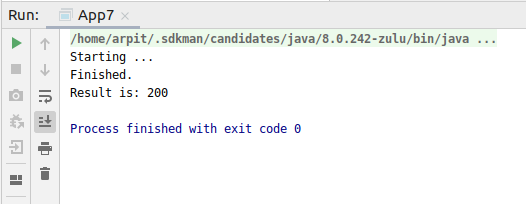
App6 obj = **new** App6();

obj.run();

}

}

1. Run a task with the help of callable and store it's result in the Future.



Code:

**package** MultiThread1;

**import** java.io.IOException;

**import** java.util.Random;

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

**public class** App7 {

**public static void** main(String[] args) {

ExecutorService executor = Executors.*newCachedThreadPool*();

Future<Integer> future = executor.submit(**new** Callable<Integer>() {

@Override

**public** Integer call() **throws** Exception {

Random random = **new** Random();

**int** duration = random.nextInt(400);

**if** (duration > 2000) {

**throw new** IOException(**"Sleeping for too long."**);

}

System.***out***.println(**"Starting ..."**);

**try** {

Thread.*sleep*(duration);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

System.***out***.println(**"Finished."**);

**return** duration;

}

});

executor.shutdown();

**try** {

System.***out***.println(**"Result is: "** + future.get());

} **catch** (InterruptedException e) {

e.printStackTrace();

} **catch** (ExecutionException e) {

IOException ex = (IOException) e.getCause(

);

System.***out***.println(ex.getMessage());

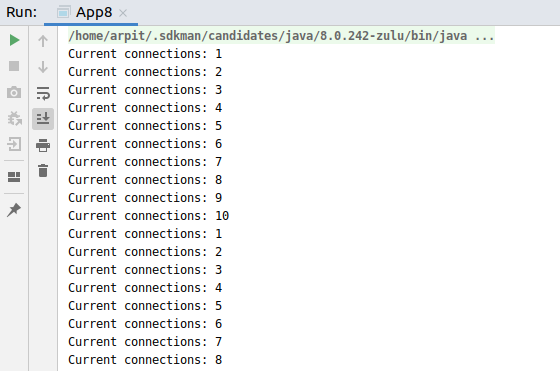
}

}

}

1. Write a program to demonstrate the use of semaphore

Ans. In this example as the Semaphore object is initialised for 10. Therefore the connections variable cannot exceed the value 10. This is because only 10 threads can run simultaneously as they need to acquire the semaphore to run.



Code:

**package** MultiThread1;

**import** java.util.concurrent.ExecutorService;

**import** java.util.concurrent.Executors;

**import** java.util.concurrent.TimeUnit;

**import** java.util.concurrent.Semaphore;

**class** Connections {

**private static** Connections *instance* = **new** Connections();

**private** Semaphore **sem** = **new** Semaphore(10, **true**);

**private int connections** = 0;

**private** Connections() {

}

**public static** Connections getInstance() {

**return** *instance*;

}

**public void** connect() {

**try** {

**sem**.acquire();

} **catch** (InterruptedException e1) {

e1.printStackTrace();

}

**try** {

doConnect();

} **finally** {

**sem**.release();

}

}

**public void** doConnect() {

**synchronized** (**this**) {

**connections**++;

System.***out***.println(**"Current connections: "** + **connections**);

}

**try** {

Thread.*sleep*(2000);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

**synchronized** (**this**) {

**connections**--;

}

}

}

**public class** App8 {

**public static void** main(String[] args) {

ExecutorService executor = Executors.*newCachedThreadPool*();

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

executor.submit(**new** Runnable() {

**public void** run() {

Connections.*getInstance*().connect();

}

});

}

executor.shutdown();

**try** {

executor.awaitTermination(1, TimeUnit.***DAYS***);

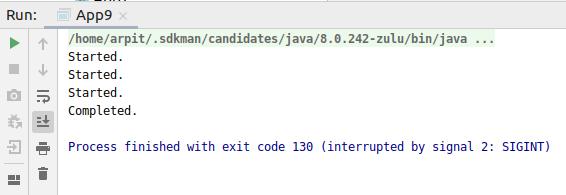
} **catch** (InterruptedException e) {

}

}

}

1. Write a program to demonstrate the use of CountDownLatch



Code:

**package** MultiThread1;

**import** java.util.concurrent.CountDownLatch;

**import** java.util.concurrent.ExecutorService;

**import** java.util.concurrent.Executors;

**class** Proc1 **implements** Runnable {

**private** CountDownLatch **latch**;

**public** Proc1(CountDownLatch latch) {

**this**.**latch**=latch;

}

**public void** run() {

System.***out***.println(**"Started."**);

**try** {

Thread.*sleep*(3000);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

**latch**.countDown();

}

}

**public class** App9 {

**public static void** main(String[] args) {

CountDownLatch latch = **new** CountDownLatch(3);

ExecutorService executor = Executors.*newFixedThreadPool*(3);

**for**(**int** i=0; i < 3; i++) {

executor.submit(**new** Proc1(latch));

}

**try** {

latch.await();

} **catch** (InterruptedException e) {

e.printStackTrace();

}

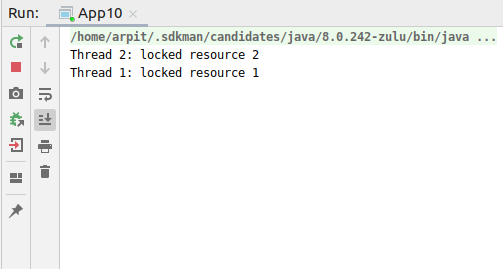
System.***out***.println(**"Completed."**);

}

}

1. Write a program which creates deadlock between 2 threads

Ans. As you can see the program is not terminating. This is because thread 2 has locked the resource 2 and thread 1 has locked the resource 2 and now both the threads are waiting for the other to release them.



Code:

**package** MultiThread1;

**public class** App10 {

**public static void** main(String[] args) {

**final** String resource1 = **"resourse 1"**;

**final** String resource2 = **"resourse 2"**;

*// t1 tries to lock resource1 then resource2*

Thread t1 = **new** Thread() {

**public void** run() {

**synchronized** (resource1) {

System.***out***.println(**"Thread 1: locked resource 1"**);

**try** { Thread.*sleep*(100);} **catch** (Exception e) {}

**synchronized** (resource2) {

System.***out***.println(**"Thread 1: locked resource 2"**);

}

}

}

};

*// t2 tries to lock resource2 then resource1*

Thread t2 = **new** Thread() {

**public void** run() {

**synchronized** (resource2) {

System.***out***.println(**"Thread 2: locked resource 2"**);

**try** { Thread.*sleep*(100);} **catch** (Exception e) {}

**synchronized** (resource1) {

System.***out***.println(**"Thread 2: locked resource 1"**);

}

}

}

};

t1.start();

t2.start();

}

}