1. Observer

public interface Observer {

void update(String message);

}

public class StockObserver implements Observer {

private String name;

private String message;

private Subject subject;

public StockObserver(Subject subject, String name) {

this.name =name;

this.subject =subject;

this.subject.register(this);

}

@Override

public void update(String msg) {

this.message =msg;

System.out.println(name +':' +msg);

}

}

public interface Subject {

void register(Observer o);

void unregister(Observer o);

void stockUpdate(String S);

}

public class StockGrabber implements Subject {

private String message;

private ArrayList<Observer> observers;

public StockGrabber() {

observers =new ArrayList<Observer>();

}

@Override

public void register(Observer o) {

observers.add(o);

}

@Override

public void unregister(Observer o) {

observers.remove(o);

}

@Override

public void stockUpdate(String S) {

this.message =S;

this.NotifyObservers();

}

private void NotifyObservers() {

for(Observer o: observers) {

o.update(this.message);

}

}

}

1. Decorator

public class DecoratorDemo {  
 public static void main(String[] args){  
 MyPizza basicPizza = new Ham(new TomatoSauce(new Cheese(new PlainPizza())));  
 System.*out*.println("Ingredients: " + basicPizza.ingredients());  
 System.*out*.println("Price: " + basicPizza.getCost());  
  
 }  
}  
  
interface MyPizza {  
 public String ingredients();  
 public double getCost();  
}  
  
class PlainPizza implements MyPizza {  
 public String ingredients() {  
 return "dough";  
 }  
  
 public double getCost() {  
 return 4.00;  
 }  
}  
  
abstract class ToppingDecorator implements MyPizza {  
 protected MyPizza tempPizza;  
  
 public ToppingDecorator(MyPizza newPizza){  
 tempPizza = newPizza;  
 }  
  
 public String ingredients() {  
 return tempPizza.ingredients();  
 }  
  
 public double getCost() {  
 return tempPizza.getCost();  
 }  
}  
  
class Cheese extends ToppingDecorator {  
  
 public Cheese(MyPizza newPizza) {  
 super(newPizza);  
 }  
  
 public String ingredients(){  
 return tempPizza.ingredients() + ", cheese";  
 }  
  
 public double getCost(){  
 return tempPizza.getCost() + 2.00;  
 }  
}  
  
class TomatoSauce extends ToppingDecorator {  
  
 public TomatoSauce(MyPizza newPizza) {  
 super(newPizza);  
 }

public String ingredients(){  
 return tempPizza.ingredients() + ", tomato sauce";  
 }  
  
 public double getCost(){  
 return tempPizza.getCost() + .35;  
 }  
}  
  
class Ham extends ToppingDecorator {  
 public Ham(MyPizza newPizza) {  
 super(newPizza);  
 }  
  
 public String ingredients() {  
 return tempPizza.ingredients() +", ham";  
 }  
  
 public double getCost() {  
 return tempPizza.getCost() +1.00;  
 }  
}

Parameterized test

import org.junit.Test;  
import org.junit.runner.RunWith;  
import org.junit.runners.Parameterized;  
  
import java.util.Arrays;  
import java.util.Collection;  
  
import static org.junit.Assert.\*;  
  
@RunWith(Parameterized.class)  
public class QuickSortTest {  
 public int[] actual;  
 public int[] expected;  
  
 public QuickSortTest(int[] expected, int[] actual) {  
 this.expected =expected;  
 this.actual =actual;  
 }  
  
 @Parameterized.Parameters  
 public static Collection<Object[]> parameters() {  
 return Arrays.*asList*(new Object[][] {  
 {new int[] {1,2,4,5,6,7,8}, new int[]{4,1,2,5,7,8,6}},  
 {new int[] {1,2,3,6,7}, new int[]{3,2,1,6,7}}  
 });  
 }  
  
 @Test  
 public void test() {  
 QuickSort quicksort =new QuickSort();  
 quicksort.sort(actual);  
 *assertTrue*(Arrays.*equals*(expected, actual));  
 }  
  
}

Mocking test

public class FindMaxUsingSorting {  
 public static int findmax (int[] inputArr, Sorter sorter) {  
 int[] result = sorter.sort(inputArr);  
 return result[result.length-1];  
 }  
}

public interface Sorter {  
 public int[] sort(int[] inputArr);  
}

import org.jmock.Expectations;  
import org.junit.Test;  
import org.jmock.Mockery;  
import org.jmock.integration.junit4.JUnit4Mockery;  
  
  
public class TestSortingWithMock {  
  
 @Test  
 public void testSort() {  
  
 Mockery context =new JUnit4Mockery();  
  
 final Sorter sorter =context.mock(Sorter.class);  
 FindMaxUsingSorting find =new FindMaxUsingSorting();  
  
 final int[] inputArray =new int[]{1, 4, 3, 5};  
  
 context.checking(new Expectations(){{  
 oneOf(sorter).sort(inputArray);  
 will(*returnValue*(new int[]{1, 3, 4, 5}));  
 }});  
  
  
 find.*findmax*(inputArray, sorter);  
  
 context.assertIsSatisfied();  
 }  
}

Web testing

package CC8;  
  
import org.openqa.selenium.By;  
import org.openqa.selenium.JavascriptExecutor;  
import org.openqa.selenium.Keys;  
import org.openqa.selenium.StaleElementReferenceException;  
import org.openqa.selenium.WebDriver;  
import org.openqa.selenium.WebElement;  
import org.openqa.selenium.chrome.ChromeDriver;  
import org.openqa.selenium.firefox.FirefoxDriver;  
import org.openqa.selenium.support.ui.ExpectedConditions;  
import org.openqa.selenium.support.ui.WebDriverWait;  
  
public class OrderFoodWithBotExtended {  
  
 static String *myUserName* = "xingxuan\_li@mymail.sutd.edu.sg";  
 static String *myPassword* = "409060778lxx";  
 static String *incorrectPassword* = "blahblah";  
 static String *myPostCode* ="485997";  
  
 public static void main(String[] args) throws InterruptedException {  
  
 System.*setProperty*("webdriver.gecko.driver","/Users/study/Desktop/ESC\_HW/geckodriver");  
  
 WebDriver driver = new FirefoxDriver();  
 driver.get("https://deliveroo.com.sg/login");  
  
 // get all the links  
 java.util.List<WebElement> links = driver.findElements(By.*tagName*("a"));  
 System.*out*.println(links.size());  
  
 for (int i = 0; i < links.size(); i=i+1) {  
 System.*out*.println(i + " " + links.get(i).getAttribute("href"));  
 }  
  
  
 // get the user name field of the account page  
 WebElement username = driver.findElement(By.*name*("login\_email"));  
  
 // send my user name to fill up the box  
 username.sendKeys(*myUserName*);  
  
 // now locate the password field in the current page  
 WebElement password = driver.findElement(By.*name*("login\_password"));  
  
 // send correct password  
 password.sendKeys(*myPassword*);  
 password.submit();  
  
 Thread.*sleep*(5000);  
  
 WebElement postcode =driver.findElement(By.*name*("postcode"));  
 postcode.sendKeys(*myPostCode*);  
 postcode.submit();  
  
  
 }  
}

Trylock

import java.util.concurrent.TimeUnit;  
import java.util.concurrent.locks.ReentrantLock;  
  
public class Trylockexample {  
 public static void main (String[] args) throws Exception {  
 final ReentrantLock reentrantLock = new ReentrantLock();  
 (new RoadHoggingThread(reentrantLock)).start();  
 while (true) {  
 Thread.*sleep*(1000);  
 boolean flag = reentrantLock.tryLock(1000, TimeUnit.MILLISECONDS);  
 if (flag) {  
 try {  
 System.*out*.println(Thread.*currentThread*().getName() +": Lock acquired.");  
 System.*out*.println("Performing task...");  
 } finally {  
 System.*out*.println(Thread.*currentThread*().getName() +": Lock released.");  
 reentrantLock.unlock();  
 }  
  
 break;  
 }  
 else {  
 System.*out*.println(Thread.*currentThread*().getName() +": Lock not available. Retry again");  
 }  
 }  
 }  
}  
  
class RoadHoggingThread extends Thread {  
 private ReentrantLock reentrantLock;  
 public RoadHoggingThread (ReentrantLock reentrantLock) {  
 this.reentrantLock = reentrantLock;  
 }  
  
 public void run () {  
 reentrantLock.lock();  
 System.*out*.println(Thread.*currentThread*().getName() +": Lock acquired.");  
 try {  
 Thread.*sleep*(5000);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 } finally {  
 reentrantLock.unlock();  
 }  
 }  
}

Global ordering

class MyCup {  
 private int water = 0;  
 private static final Object *tieLock* = new Object();  
  
 public MyCup (int i) {  
 water = i;  
 }  
  
 public synchronized void top (int quantify) {  
 water = water + quantify;  
 }  
  
 public synchronized void pour (int quantify) {  
 water = water - quantify;  
 }  
  
 public void pour (MyCup other, int quantify) {  
 int myHash = System.*identityHashCode*(this);  
 int otherHash = System.*identityHashCode*(other);  
  
 if (myHash < otherHash) {  
 pour(quantify);  
 other.top(quantify);   
 }  
 else if (myHash > otherHash) {  
 other.top(quantify);  
 pour(quantify);  
 }   
 else {  
 synchronized(*tieLock*) {  
 pour(quantify);  
 other.top(quantify);  
 }  
 }  
 }  
}

Future task

package Week10;  
  
 import java.util.concurrent.Callable;  
 import java.util.concurrent.ExecutionException;  
 import java.util.concurrent.FutureTask;  
  
public class FutureTaskExample {  
 public static void main(String[] args) {  
 FutureTask<String> future = new FutureTask<>(new CallableTask());  
 //Cancelling code before run  
 /\*boolean b = future.cancel(true);  
 System.out.println("Cancelled="+b);\*/  
 future.run();  
 //Cancelling code after run  
 /\*boolean b = future.cancel(true);  
 System.out.println("Cancelled="+b);\*/  
 System.*out*.println("Result=");  
 try {  
 String result = future.get();  
 System.*out*.println(result);  
 } catch (InterruptedException | ExecutionException e) {  
 System.*out*.println("EXCEPTION!!!");  
 e.printStackTrace();  
 }  
 }  
}  
  
class CallableTask implements Callable<String>{  
 public String call() throws Exception {  
 Thread.*sleep*(10000);  
 System.*out*.println("Executing call() !!!");  
 /\*if(1==1)  
 throw new java.lang.Exception("Thrown from call()");\*/  
 return "success";  
 }  
}

package Week10;  
  
 import java.util.concurrent.Callable;  
 import java.util.concurrent.ExecutionException;  
 import java.util.concurrent.FutureTask;  
  
public class FutureTaskExample2 {  
 public static void main(String[] args) {  
 FutureTask<String> future = new FutureTask<>(new CallableTask());  
 //Cancelling code before run  
 /\*boolean b = future.cancel(true);  
 System.out.println("Cancelled="+b);\*/  
 DemoFutureTask<String> myThread = new DemoFutureTask<String>(future);  
 myThread.start();  
 //Cancelling code after run  
 /\*boolean b = future.cancel(true);  
 System.out.println("Cancelled="+b);\*/  
 System.*out*.println("Result=");  
 try {  
 String result = future.get();  
 System.*out*.println(result);  
 } catch (InterruptedException | ExecutionException e) {  
 System.*out*.println("EXCEPTION!!!");  
 e.printStackTrace();  
 }  
 }  
}  
  
class DemoFutureTask<E> extends Thread {  
 private final FutureTask<E> task;  
 public DemoFutureTask (FutureTask<E> task) {  
 this.task = task;  
 }  
  
 public void run () {  
 task.run();  
 }  
}

Semaphore

package Week11;  
  
 import java.util.concurrent.Semaphore;  
  
public class SemaphoreExample {  
 private final Semaphore roomOrganizer=new Semaphore(3, true); //true: first come, first serve  
  
 public static void main(String[] args) {  
 System.*out*.println("lalal"); SemaphoreExample test=new SemaphoreExample();  
 test.mystart();  
 }  
  
 public void mystart() {  
 for(int i=0; i<10; i++) {  
 People people=new People();  
 people.start();  
 }  
 }  
  
 class People extends Thread {  
 public void run() {  
 try {  
 roomOrganizer.acquire();  
 } catch (InterruptedException e) {  
 System.*out*.println("received InterruptedException");  
 return;  
 }  
  
 System.*out*.println("Thread "+this.getId()+" starts to use the room");  
  
 try {  
 *sleep*(1000);  
 } catch (InterruptedException e) {  
 }  
  
 System.*out*.println("Thread "+this.getId()+" leaves the room\n");  
 roomOrganizer.release();  
 }  
 }  
}

Cyclic barrier

package inclassprograms;  
  
 import java.util.concurrent.CyclicBarrier;  
 import java.util.logging.Level;  
 import java.util.logging.Logger;  
  
public class BarrierExample {  
  
 private static class Task implements Runnable {  
 private CyclicBarrier barrier;  
  
 public Task(CyclicBarrier barrier) {  
 this.barrier = barrier;  
 }  
  
 public void run() {  
 try {  
 System.*out*.println(Thread.*currentThread*().getName() + " is waiting on barrier");  
 barrier.await();  
 System.*out*.println(Thread.*currentThread*().getName() + " has crossed the barrier");  
 } catch (Exception ex) {  
 Logger.getLogger(BarrierExample.class.getName()).log(Level.SEVERE, null, ex);  
 }  
 }  
 }  
  
 public static void main(String args[]) {  
 //A CyclicBarrier supports an optional Runnable command that is run once per barrier point,   
 //after the last thread in the party arrives, but before any threads are released. This   
 //barrier action is useful for updating shared-state before any of the parties continue.  
 final CyclicBarrier cb = new CyclicBarrier(3, new Runnable(){  
 public void run(){  
 //This task will be executed once all thread reaches barrier  
 System.*out*.println("All parties are arrived at barrier, lets play");  
 }  
 });  
  
 //starting each of thread  
 Thread t1 = new Thread(new Task(cb), "Thread 1");  
 Thread t2 = new Thread(new Task(cb), "Thread 2");  
 Thread t3 = new Thread(new Task(cb), "Thread 3");  
  
 t1.start();  
 t2.start();  
 t3.start();  
 }  
}

Count down latch

package Week11;  
  
 import java.util.concurrent.CountDownLatch;  
 import java.util.logging.Level;  
 import java.util.logging.Logger;  
  
public class CountDownLatchExample {  
  
 public static void main(String args[]) {  
 final CountDownLatch latch = new CountDownLatch(3);  
 Thread cacheService = new Thread(new Service("CacheService", 1000, latch));  
 Thread alertService = new Thread(new Service("AlertService", 1000, latch));  
 Thread validationService = new Thread(new Service("ValidationService", 1000, latch));  
  
 cacheService.start(); //separate thread will initialize CacheService  
 alertService.start(); //another thread for AlertService initialization  
 validationService.start();  
  
 try{  
 latch.await(); //main thread is waiting on CountDownLatch to finish  
 Thread.*sleep*(1000);  
 System.*out*.println("All services are up, Application is starting now");  
 }catch(InterruptedException ie){  
 ie.printStackTrace();  
 }  
 }  
}  
  
class Service implements Runnable{  
 private final String name;  
 private final int timeToStart;  
 private final CountDownLatch latch;  
  
 public Service(String name, int timeToStart, CountDownLatch latch){  
 this.name = name;  
 this.timeToStart = timeToStart;  
 this.latch = latch;  
 }  
  
 public void run() {  
 try {  
 Thread.*sleep*(timeToStart);  
 } catch (InterruptedException ex) {  
 Logger.getLogger(Service.class.getName()).log(Level.SEVERE, null, ex);  
 }  
 System.*out*.println( name + " is Up");  
 latch.countDown(); //reduce count of CountDownLatch by 1  
 System.*out*.println( name + " has passed countdown");  
 }  
}

Phaser

package Week11;  
  
 import java.util.concurrent.Phaser;  
  
public class PhaserExample  
{  
 public static void main(String[] args) throws InterruptedException  
 {  
 Phaser phaser = new Phaser();  
 phaser.register();//register self... phaser waiting for 1 party (thread)  
 int phasecount = phaser.getPhase();  
 System.*out*.println("Phasecount is "+phasecount);  
 new PhaserExample().testPhaser(phaser,2000);//phaser waiting for 2 parties  
 new PhaserExample().testPhaser(phaser,4000);//phaser waiting for 3 parties  
 new PhaserExample().testPhaser(phaser,6000);//phaser waiting for 4 parties  
 //now that all threads are initiated, we will de-register main thread   
 //so that the barrier condition of 3 thread arrival is meet.  
 phaser.arriveAndDeregister();  
 System.*out*.println(phaser.getRegisteredParties() + " have registered.");  
 Thread.*sleep*(10000);  
 phasecount = phaser.getPhase();  
 System.*out*.println("Phasecount is "+phasecount);  
  
 }  
  
 private void testPhaser(final Phaser phaser,final int sleepTime)  
 {  
 phaser.register();  
 System.*out*.println(phaser.getRegisteredParties() + " have registered.");  
 new Thread(){  
 @Override  
 public void run()  
 {  
 try  
 {  
 System.*out*.println(Thread.*currentThread*().getName()+" arrived");  
 phaser.arriveAndAwaitAdvance();//threads register arrival to the phaser.  
 //phaser.arrive();  
 //System.out.println(Thread.currentThread().getName()+" pass arrived");  
 Thread.*sleep*(sleepTime);  
 }  
  
 catch (InterruptedException e)  
 {  
 e.printStackTrace();  
 }  
 System.*out*.println(Thread.*currentThread*().getName()+" after passing barrier");  
 }  
 }.start();  
 }  
}

Striped map

package CC6;  
  
public class StripedMapWithSize {  
 //synchronization policy: buckets[n] guarded by locks[n%N\_LOCKS]  
 private static final int *N\_LOCKS* = 16;  
 private final Node[] buckets;  
 private final Object[] locks;  
  
 public StripedMapWithSize (int numBuckets) {  
 buckets = new Node[numBuckets];  
 locks = new Object[*N\_LOCKS*];  
  
 for (int i = 0; i < *N\_LOCKS*; i++) {  
 locks[i] = new Object();  
 }  
 }  
  
 public Object put(Object key, Object value) {  
 int hash = hash(key);  
 synchronized (locks[hash % *N\_LOCKS*]) {  
 for (Node m = buckets[hash]; m != null; m = m.next)  
 if (m.key.equals(key)) {  
 m.value = value;  
 return m.value;  
 }  
 buckets[hash] = new Node(key,value,buckets[hash]);  
 }  
 return null;  
 }  
  
 public Object get (Object key) {  
 //*todo: get the item with the given key in the map* int hash =hash(key);  
 synchronized (locks[hash %*N\_LOCKS*]) {  
 for (Node m =buckets[hash]; m !=null; m =m.next) {  
 if (m.key.equals(key)) {  
 return m.value;  
 }  
 }  
  
 System.*out*.println("The item does not exist in the bucket.");  
 }  
  
 return null;  
 }  
  
 private final int hash (Object key) {  
 return Math.*abs*(key.hashCode() % buckets.length);  
 }  
  
 public void clear () {  
 //*todo: remove all objects in the map* for (int i =0; i <*N\_LOCKS*; i ++) {  
 synchronized (locks[i]) {  
 for (Node m=buckets[i]; m !=null; m =m.next) {  
 m.value =null;  
 }  
 }  
 }  
 }

public int size () {  
 //*todo: count the number of elements in the map* int size =0;  
 for (int i =0; i <*N\_LOCKS*; i ++) {  
 synchronized (locks[i]) {  
 for (Node m =buckets[i]; m !=null; m =m.next) {  
 size ++;  
 }  
 }  
 }  
 return size;  
 }  
  
 class Node {  
 Node next;  
 Object key;  
 Object value;  
 Node(Object key, Object value, Node next) {  
 this.next = next;  
 this.key = key;  
 this.value = value;  
 }  
 }  
}

Parallel

public class SPMDIntegration {  
 public static double *result* =0;  
 public static void main(String[] args) throws Exception {  
 int NTHREADS = 5;  
 ExecutorService exec = Executors.*newFixedThreadPool*(NTHREADS - 1);  
 // *todo: complete the program by writing your code below.* Runnable task1 =new Runnable() {  
 @Override  
 public void run() {  
 *integrate*(0, 0.25);  
 }  
 };  
  
 Runnable task2 =new Runnable() {  
 @Override  
 public void run() {  
 *integrate*(0.25, 0.5);  
 }  
 };  
  
 Runnable task3 =new Runnable() {  
 @Override  
 public void run() {  
 *integrate*(0.5, 0.75);  
 }  
 };  
  
 Runnable task4 =new Runnable() {  
 @Override  
 public void run() {  
 *integrate*(0.75, 1);  
 }  
 };  
  
 exec.execute(task1);  
 exec.execute(task2);  
 exec.execute(task3);  
 exec.execute(task4);  
  
 exec.shutdown();  
 exec.awaitTermination(2, TimeUnit.*SECONDS*);  
 System.*out*.println(*result*);  
  
 }  
  
 public static double f(double x) {  
 return 4.0 / (1 + x \* x);  
 }  
  
 public static double integrate(double a, double b) {  
 int N = 10000; // preciseness parameter  
 double h = (b - a) / (N - 1); // step size  
 double sum = 1.0 / 2.0 \* (*f*(a) + *f*(b)); // 1/2 terms  
  
 for (int i = 1; i < N - 1; i++) {  
 double x = a + h \* i;  
 sum += *f*(x);  
 }  
 *result* +=sum \*h;  
  
 return sum \* h;  
 }  
}

CAS

package Week12;  
  
public class SimulatedCAS {  
 private int value;  
  
 public synchronized int get() {  
 return value;  
 }  
  
 public synchronized int compareAndSwap(int expectedValue,  
 int newValue) {  
 int oldValue = value;  
 if (oldValue == expectedValue)  
 value = newValue;  
 return oldValue;  
 }  
  
 public synchronized boolean compareAndSet(int expectedValue,  
 int newValue) {  
 return (expectedValue  
 == compareAndSwap(expectedValue, newValue));  
 }  
}