**Java Coding / Algorithms / Testing**

**Interview Questions**

**► What “Big O” is, how do you calculate / approximate it? Examples of Big O?**

A theoretical measure of the **complexity of an algorithm**, usually the time or memory needed, given the problem size n, which is usually the **number of items**. Informally, saying some equation f(n) = O(g(n)) means it is less than some constant multiple of g(n). The notation is read, "f of n is big oh of g of n".

There is **no mechanical procedure** that can be used to get the Big O. You need to create a math formula to count how many steps of computations get executed given an input of some size. The importance of this measure is to help deciding whether an algorithm is adequate, may just need a better implementation, or will always be too slow.

* Balanced binary trees traversal - **O(log n)**
* Quicksort - **O(n log n)**
* Bubble sort - **O(n²)**

**► When to use recursion?**

When writing any recursive algorithm, you should consider possible recursion depth because it may translate into stack space. As a rule of thumb if it's **O(log n)** recursion is very unlikely to raise any run-time problem. If recursion depth is **O(n)** this could lead to stack troubles, and if it's worse than **O(n)**, you are very likely to crash the stack.

**► What is tail recursion? Why is it so bad?**

Tail recursion is a special kind of recursion where the recursive call is the very **last statement** in the function. It means that you can just pass the result of the recursive call through directly, you don't have to consume any stack space. You can rewrite tail recursive function as an **iteration** or it can be replaced by compiler, if it has such optimization option.

**► How to read value from system input?**

Use Scanner, this could be used in most examples.

import java.util.Scanner;

public static void main(String[] args) {

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

System.out.println("Enter number:");

int num = scanner.**nextInt();**

}

**►► Return true if a given integer is odd, false otherwise?**

// check for not 0 as remainder could be either 0, 1, or -1:

public static boolean isOdd(int number) {  
 return (number%2 != 0) ? true : false;  
}

// an alternative using bits, low bit will always be set to 1 on an odd number.

public static boolean isOddB(int number) {

return ((number & 1) != 0) ? true : false;

}

**►► Determine if a number is Prime number or not?**

A prime number is a whole number greater than 1, whose only two whole-number factors are 1 and itself. The first few prime numbers are 2, 3, 5, 7, 11, 13, 17, 19, 23, and 29.

public class PrimeNumber

{

**public static boolean isPrime(int number) {**

if (number <= 1)

return false;

if (number == 2)

return true;

// it is enough to check from 2 to SQRT(N)

for (int i=2; i\*i <= number; i++ ) {

// if divisor is found - not prime, break

if (number%i == 0)

return false;

}

// found no divisor

return true;

}

}

“Parameterized” **JUnit** test, using test **collection**:

import java.util.Arrays;

import java.util.Collection;

import junit.framework.Assert;

import org.junit.Test;

import org.junit.runner.RunWith;

import org.junit.runners.Parameterized;

**@RunWith(Parameterized.class)**

public class PrimeNumberTest

{

private int input;

private boolean valid;

public PrimeNumberTest(int input, boolean valid)

{

this.input = input;

this.valid = valid;

}

**@Parameterized.Parameters**

**public static Collection toTest()**

{

return Arrays.asList(new Object[][]{

{-12, false},

{ 0, false},

{ 1, false},

{ 2, true},

{ 3, true},

{ 4, false},

{ 5, true},

{ 23, true},

{ 28, false},

{ 29, true},

{ 30, false},

{996, false},

{997, true}

});

}

@Test

public void testPrime() {

// short version:

// **Assert.assertEquals( valid, PrimeNumber.isPrime(input));**

boolean isPrime = PrimeNumber.isPrime(input);

Assert.assertEquals( //-- message, expected, actual

String.format("Unit test failed for [%d], expected: [%s], actual: [%s]",

input, valid, isPrime),

valid, isPrime);

}

}

**►► Check if a number is Armstrong or not?**

[An Armstrong number is a **3 digit number** for which sum of cube of its digits is equal to the number itself. An example of Armstrong number is **153** as 153=1+125+27 which is equal to **1^3+5^3+3^3**.]

// TODO definition, and input check in method

/\*\* @return true if number is Armstrong number or return false \*/

private static boolean isArmStrong(int number) {

int result = 0;

int orig = number;

while(number != 0) {

int remainder = number%10;

result = result + remainder\*remainder\*remainder;

number = number/10;

}

return (orig == result)? true : false;

}

}

**►►► Calculate factorial of a number**

Consider cases of both small and large numbers. The problem is that **12!** is the max correct integer result and **20!** is the max correct long result while integer **13!** and long **21!** give **wrong negative result** because of unchecked overflow. Use **java.math.BigInteger** instead, note it has own constants and operations.

public static **BigInteger** factorialIterBig(int n) {

BigInteger result = **BigInteger.ONE**;

while (n > 1) {

result = **result.multiply( BigInteger.valueOf(n)**);

n--;

}

return result;

}

private static **BigInteger** factorialRbig(int n) {

if (n==1)

return **BigInteger.ONE**;

BigInteger result = factorialRbig(n-1)**.multiply(BigInteger.valueOf(n));**

return result;

}

**►► How to swap two numbers without using third variable?**

int a = 10; int b = 20;

a = a + b; //now a is 30 and b is 20

b = a - b; //now a is 30 but b is 10 (original value of a)

a = a - b; //now a is 20 and b is 10, numbers are swapped

**►► Write a Java program to replace certain characters from String like:**

**String replace(String str, char oldCh, char newCh)**

This coding question can be solved in multiple way e.g. by using charAt() or subString() method, you may be asked to write two versions, one by using recursion and other by using Iteration. They may also ask you to write **JUnit** test for this function which means handling **null**, **empty string**, etc.

// TODO example

**►►► Reverse characters in a String [without using API functions or StringBuffer]**

public static String **reverseRecurs** (String str) {

if (str.length() < 2)

return str;

return **reverseRecurs ( str.substring(1)) + str.charAt(0);**

}

// reverse iteration:

String res1 = "";

for(int i=orig.length()-1; i>=0; i--) {

res1 += orig.charAt(i);

}

**►► Reverse words in a String [without using API functions or StringBuffer]**

// input: "one two three four"

// output: "four three two one"

public static String reverse(String input) {

if (input == null || input.length() < 2) return input;

String[] parts = input.split(" ");

if(parts.length == 1) {

return input;

}

//-- don’t forget about space in between, start from the last or cut off after

String result = parts[ parts.length-1];

**for(int i=parts.length-2; i >= 0; i--) {**

result += " " + parts[i];

}

return result;

}

**►►► FizzBuzz: Print numbers from 1 to 50 as: for multiples of 3 print "Fizz" instead of the number, for the multiples of 5 print "Buzz", for numbers which are multiples of both 3 and 5 print "FizzBuzz".**

public static void main(String args[]) {

for (int **i = 1;** i <= 50; i++) {

if (i % (3\*5) == 0)

System.out.println("FizzBuzz");

else if (i % 5 == 0)

System.out.println("Buzz");

else if (i % 3 == 0)

System.out.println("Fizz");

else

System.out.println(i);

}

}

**►► Write a program to change number into words**

// TODO example

**►► Write a program to print all the permutations of a string**

[eg: if abc is a string then o/p should be as abc,acb,bac,bca,cab,cba. i.e 3 factorial ways.]

// TODO example

**►► Given this program, what is the output of passing 13 to the both functions? Why are the values different?**

public void met1(int x){

System.out.println(x%2);

if(x>0)

met1(x/2);

}

publilc void met2(int x){

if(x>0)

met2(x/2);

System.out.println(x%2);

}

The met1 prints remainder for each input first, and has recursion after that. Result is: 10110

The met2 calls itself recursively and prints remainder after, in **reverse** order. Result is: 01101

**►► Write a program which when given an integer value for month and day determine if the input is a correct value. Example, (1, 33) would return false...**

Note, that February also needs a year to select 28 or 29 days.

static boolean isValidDate(int month, int day)

{

if (month < 1 || day < 1)

return false;

switch (month)

{

// 31 days:

**case 1: case 3: case 5: case 7: case 8: case 10: case 12:**

return (day <= 31);

// 30 days:

**case 4: case 6: case 9: case 11:**

return (day <= 30);

**// 28 or 29 days:**

case 2:

// if (((year % 4 == 0) && !(year % 100 == 0)) || (year % 400 == 0))

// numDays = 29;

return (day <= 29);

default:

return false;

}

}

►► **What will be result of executing the code?**

public class A {

public A() {

System.out.print("A");

}

public static void main(String[] args) {

B b = new B();

}

}

class B extends A {

public B() {

System.out.print("B");

}

}

By default, the first line of constructor is call to super constructor. Result: **AB**

**►► Write a program which allows only at most one instance of a class (singleton) per process?**

If Spring framework is used - recommended, just use **singleton** or **application** **scope** for the bean. **Application** context bean is a singleton per ServletContext (not per Spring ApplicationContext) and is visible as a ServletContext attribute. **@Lazy** annotation could be used to have that singleton lazy initialized.

@Bean(name = QRS\_HTTP\_CLIENT)

**@Scope("singleton")**

**@Lazy**

HttpClient qrsHttpClient() {

…

}

There are several ways to implement Singleton pattern in standard Java.

<http://www.journaldev.com/1377/java-singleton-design-pattern-best-practices-examples>

For example:

* **private constructor** to restrict instantiation of the class from other classes.
* **private static variable** of the same class that is the only instance of the class.
* **public static method** that returns the instance of the class, this is the global access point for outer world to get the instance of the singleton class.

public class EagerInitializedSingleton {

private static final EagerInitializedSingleton instance = new EagerInitializedSingleton();

//private constructor to avoid client applications to use constructor

private EagerInitializedSingleton(){}

public static EagerInitializedSingleton getInstance(){

return instance;

}

}

**►► Given the program, what is printed?**

class X {

int v = 10;

}

import X;

public class Y

{

public static void main(String args[]) {

X x = new X();

x.v = 50;

dowork( x);

System.out.println(x.v);

}

public static void dowork(X obj) {

obj.v = 0;

X xx = new X();

obj = xx;

System.out.println(obj.v);

}

}

Result is:

**10** // obj refers to newly created X

**0** // dowork changed v field of parameter by reference,

// assigning new ref to parameter (obj = xx;) makes no effect to outside

**►►► Write function producing Fibonacci series / sequence [e.g. up to 100].**

The first two numbers in the Fibonacci sequence are 0 and 1, and each subsequent number is the sum of the previous two: **0, 1, 1, 2, 3, 5, 8, 13, ..**. Note, using **recursion** as shown below is **not efficient**, as for each next number it recalculates the sequence twice again.

// Using loop and recursion – not efficient!

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

System.out.printf(“%d ”, fibo(i));

}

public static int fibo(int n) {

if (n==0 || n==1)

return 1;

return fibo(n-2) + fibo(n-1);

}

// Iteration

int f = 1;

int s = 1;

System.out.printf("fibo: %d %d", f, s);

for (int i=0; i<q-2; i++) {

int next = f+s;

f = s;

s = next;

System.out.printf(" %d", next);

}

**►► Design a vending machine which vends Item based upon four denomination of coins and returns coins if there is no Item. [You need to design, implement and write JUnit test]**

// TODO example

**►► Write [a Comparator] method to compare two employees based upon their name, departments and age?**

[Remember that Comparable has compareTo() method and use to sort object based upon their natural order e.g. numeric order for number, and alphabetic order for String, while Comparator can define any arbitrary sorting.]

// TODO example

**► Write program to find maximum number in array.**

// TODO example

**►►► Calculate array Sum using recursion.**

public static void main(String... args) {

int[] a = { 1, 2, 3, 4, 5, 6 };

System.out.println( sumOfArray(a, a.length-1));

}

**public static int sumOfArray(int[] a, int n) {**

if (n == 0)

return a[n];

else

**return a[n] + sumOfArray(a, n-1);**

}

**►► How to remove duplicates from an array of Integers, without using API methods?**

// TODO example

**►► Write a program to find a missing number in array of 100 numbers**

// TODO example

**►► Find the count of number in list? ex: 1,1,1,1,2,2,3,3,4,4,4,4,4**

// TODO example

**►► Code to implement binary search?**

// TODO example

**►► Find middle node of linked list in one pass?**

To find middle element of linked list we need to find **length** first but since we can only traverse linked list one time, we can use two pointers - one which we will increment on each iteration while other which will be incremented every second iteration, so when the first pointer will point to the end of linked list, second will be pointing to the middle element of linked list.

while(current.next() != null) {

length++;

if (length%2 ==0 ) {

middle = middle.next();

}

current = current.next();

}

if (length%2 == 1) {

middle = middle.next();

}

A Java LinkedList is a **doubly linked** list, so in this case the correct solution would be to advance **from both sides** (head and tail) of the doubly linked list simultaneously until you meet in the middle element (happens for **odd** list length) or the next element matches the current element of the other (happens for **even** list lengths).

**►►► How to find if a Linked List contains cycle or not? How to find start of that cycle?**

**Floyd's cycle finding** algorithm, also known as "**tortoise and hare**" cycle detection algorithm – “**fast and slow pointer**” approach.

/\*\* If singly LinkedList contains Cycle then following would be true

\* 1) slow and fast will point to same node i.e. they meet

\* On the other hand if fast will point to null or next node of

\* fast will point to null then LinkedList does not contain cycle.

\*/

public boolean isCyclic(){

Node fast = head;

Node slow = head;

while (fast!= null && fast.next != null) {

fast = fast.next.next;

slow = slow.next;

//if fast and slow pointers are meeting then LinkedList is cyclic

if (fast == slow ){ return true; }

}

return false;

}

**►► Remove the current node from the list without moving back.**

Lets say there is a linked list :A1 -> A2 -> A3 -> A4 -> A5

The pointer is at **A3**. You have to delete the node A3, while you don’t have the pointer to A2.

Answer: copy the value of **A4** to A3, delete the node A4 (A3.next = A3.next.next).

**►► Write program to sort Map based on keys [without using TreeMap]**

new TreeSet(map.keySet());

// TODO example

**►► Given a binary tree, determine if it is a valid Binary Search Tree (BST).**

Assume a BST is defined as follows:

* The left subtree of a node contains only nodes with keys less than the node's key.
* The right subtree of a node contains only nodes with keys greater than the node's key.
* Both the left and right subtrees must also be binary search trees.

/\* returns true if given search tree is BST (efficient version) \*/

boolean isBST() {

return isBSTUtil(root, Integer.MIN\_VALUE,

Integer.MAX\_VALUE);

}

/\* Returns true if the given tree is a BST and its values are >= min and <= max. \*/

boolean isBSTUtil(Node node, int min, int max)

{

/\* an empty tree is BST \*/

if (node == null)

return true;

/\* false if this node violates the min/max constraints \*/

if (node.data < min || node.data > max)

return false;

/\* otherwise check the subtrees recursively tightening the min/max constraints \*/

// Allow only distinct values

return (isBSTUtil(node.left, min, node.data-1) &&

isBSTUtil(node.right, node.data+1, max));

}

**►► Given guaranteed sorted input 1,2,4,6,7,8,9,19,22, insert into a balanced binary search tree (BST).**

See: <http://www.sanfoundry.com/java-program-implement-self-balancing-binary-search-tree/>

**public class SBBSTNode**

{

SBBSTNode left, right;

int data;

int height;

/\* Constructor \*/

public SBBSTNode()

{

left = null;

right = null;

data = 0;

height = 0;

}

/\* Constructor \*/

public SBBSTNode(int n)

{

left = null;

right = null;

data = n;

height = 0;

}

}

**public class SelfBalancingBinarySearchTree**

{

private SBBSTNode root;

/\* Constructor \*/

public SelfBalancingBinarySearchTree()

{

root = null;

}

/\* Function to check if tree is empty \*/

public boolean isEmpty()

{

return root == null;

}

/\* Make the tree logically empty \*/

public void clear()

{

root = null;

}

**/\* Function to insert data \*/**

**public void insert(int data)**

**{**

**root = insert(data, root);**

**}**

/\* Function to get height of node \*/

private int height(SBBSTNode t )

{

return t == null ? -1 : t.height;

}

/\* Function to max of left/right node \*/

private int max(int lhs, int rhs)

{

return lhs > rhs ? lhs : rhs;

}

/\* Function to insert data recursively \*/

**private SBBSTNode insert(int x, SBBSTNode t)**

{

if (t == null)

t = new SBBSTNode(x);

else if (x < t.data)

{

t.left = insert( x, t.left );

if (height( t.left ) - height( t.right ) == 2)

if (x < t.left.data)

t = rotateWithLeftChild( t );

else

t = doubleWithLeftChild( t );

}

else if (x > t.data)

{

t.right = insert( x, t.right );

if (height( t.right ) - height( t.left ) == 2)

if (x > t.right.data)

t = rotateWithRightChild( t );

else

t = doubleWithRightChild( t );

}

else

; // Duplicate; do nothing

t.height = max( height( t.left ), height( t.right ) ) + 1;

return t;

}

/\* Rotate binary tree node with left child \*/

private SBBSTNode rotateWithLeftChild(SBBSTNode k2)

{

SBBSTNode k1 = k2.left;

k2.left = k1.right;

k1.right = k2;

k2.height = max( height( k2.left ), height( k2.right ) ) + 1;

k1.height = max( height( k1.left ), k2.height ) + 1;

return k1;

}

/\* Rotate binary tree node with right child \*/

private SBBSTNode rotateWithRightChild(SBBSTNode k1)

{

SBBSTNode k2 = k1.right;

k1.right = k2.left;

k2.left = k1;

k1.height = max( height( k1.left ), height( k1.right ) ) + 1;

k2.height = max( height( k2.right ), k1.height ) + 1;

return k2;

}

/\*\*

\* Double rotate binary tree node: first left child

\* with its right child; then node k3 with new left child \*/

private SBBSTNode doubleWithLeftChild(SBBSTNode k3)

{

k3.left = rotateWithRightChild( k3.left );

return rotateWithLeftChild( k3 );

}

/\*\*

\* Double rotate binary tree node: first right child

\* with its left child; then node k1 with new right child \*/

private SBBSTNode doubleWithRightChild(SBBSTNode k1)

{

k1.right = rotateWithLeftChild( k1.right );

return rotateWithRightChild( k1 );

}

/\* Functions to count number of nodes \*/

public int countNodes()

{

return countNodes(root);

}

private int countNodes(SBBSTNode r)

{

if (r == null)

return 0;

else

{

int l = 1;

l += countNodes(r.left);

l += countNodes(r.right);

return l;

}

}

/\* Functions to search for an element \*/

public boolean search(int val)

{

return search(root, val);

}

private boolean search(SBBSTNode r, int val)

{

boolean found = false;

while ((r != null) && !found)

{

int rval = r.data;

if (val < rval)

r = r.left;

else if (val > rval)

r = r.right;

else

{

found = true;

break;

}

found = search(r, val);

}

return found;

}

/\* Function for in order traversal \*/

public void inorder()

{

inorder(root);

}

private void inorder(SBBSTNode r)

{

if (r != null)

{

inorder(r.left);

System.out.print(r.data +" ");

inorder(r.right);

}

}

/\* Function for preorder traversal \*/

public void preorder()

{

preorder(root);

}

private void preorder(SBBSTNode r)

{

if (r != null)

{

System.out.print(r.data +" ");

preorder(r.left);

preorder(r.right);

}

}

/\* Function for postorder traversal \*/

public void postorder()

{

postorder(root);

}

private void postorder(SBBSTNode r)

{

if (r != null)

{

postorder(r.left);

postorder(r.right);

System.out.print(r.data +" ");

}

}

}

Post order : 1 4 2 7 9 22 19 8 6

Pre order : 6 2 1 4 8 7 19 9 22

In order : 1 2 4 6 7 8 9 19 22

**►►► Write a program to create Thread and stop Thread**

**Thread.stop()** is deprecated because it is inherently **unsafe**. Most uses of stop should be replaced by code that simply modifies some **variable** to indicate that the target thread should stop running. The target thread should check this variable regularly, and simply **return** from its run method in an orderly fashion. To ensure prompt communication of the stop-request, the variable must be **volatile** (or access to the variable must be **synchronized**).

public static void main(String[] args){

MyThread myThread = new MyThread();

Thread th = new Thread(myThread);

th.start();

//Some logic goes there to decide whether to stop the thread or not.

**myThread.shutdown();**

}

public class MyThread implements Runnable {

**private volatile boolean shutdown;**

public void run() {

while (!shutdown) {

...

}

}

**public void shutdown() {**

**shutdown = true;**

}

}

**►►► Write program to create deadlock. How do you fix it?**

Deadlock happens if four condition is true e.g. mutual exclusion, no waiting, circular wait and no preemption. If you can break any of this condition than you can create Java program which has deadlock. One easy way to avoid deadlock is by imposing an ordering on acquisition and release of locks.

// TODO example

**► Write program which waits for a Thread supplying some incoming messages.**

public class SimpleThreads

{

// Log preceded by the name of the current thread

static void logThread(String info) {

String threadName = Thread.currentThread().getName();

System.out.format("%s: %s%n", threadName, info);

}

private static class MessageLoop implements Runnable {

public void run() {

String importantInfo[] = {

"Mares eat oats", "Does eat oats", "Little lambs eat ivy", "A kid will eat ivy too"

};

try {

for (int i = 0; i < importantInfo.length; i++) {

// Pause for 4 seconds

Thread.sleep(4000);

logThread(importantInfo[i]);

}

} catch (InterruptedException e) {

logThread("I wasn't done!");

}

}

}

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

// Delay, in milliseconds before we interrupt MessageLoop thread (default one hour).

long patience = 1000 \* 60 \* 60;

logThread("Starting MessageLoop thread");

long startTime = System.currentTimeMillis();

Thread messagesThread = new Thread( new MessageLoop(), "Messages Thread");

**messagesThread.start();**

// loop until MessageLoop thread exits

logThread("Waiting for MessageLoop thread to finish");

while (**messagesThread.isAlive())**

{

logThread("Still waiting for messages...");

// Wait max 1 second for MessageLoop thread to finish.

**messagesThread.join( 1000);**

if (((System.currentTimeMillis() - startTime) > patience) && **messagesThread.isAlive()**)

{

logThread("Tired of waiting!");

**messagesThread.interrupt();**

// Shouldn't be long now, wait indefinitely

**messagesThread.join();**

}

}

logThread("Finally!");

}

}

Output:

main: Starting MessageLoop thread

main: Waiting for MessageLoop thread to finish

main: Still waiting for messages...

main: Still waiting for messages...

**Messages Thread: Mares eat oats**

main: Still waiting for messages...

main: Still waiting for messages...

**Messages Thread: Does eat oats**

main: Still waiting for messages...

main: Still waiting for messages...

**Messages Thread: Little lambs eat ivy**

main: Still waiting for messages...

main: Still waiting for messages...

**Messages Thread: A kid will eat ivy too**

main: Finally!

**►► What is the output of the program containing threads?**

public class Counter

{

int count = 0;

public void increment(){

count++;

System.out.println(count);

}

public void decrement(){

count--;

System.out.println(count);

}

}

public class Worker

{

**private static final int MAX\_THREADS = 100;**

public static void main(String args[]) {

Thread[] threads = new Thread[MAX\_THREADS];

Counter counter = new Counter();

for(int i = 0; i < MAX\_THREADS; i++) {

if((i & 1) == 0)

**threads[i] = new Thread(){ public void run(){counter.increment();}};**

else

**threads[i] = new Thread(){ public void run(){counter.decrement();}};**

}

// if all threads were started here and joined later joined

// results would be random because of concurrency, not in any special order

/\* for(int i = 0; i < MAX\_THREADS; i++) {

threads[i].start();

} \*/

try{

for(int i = 0; i < MAX\_THREADS; i++) {

// if thread is started and joined here, main thread will just wait

// until that thread finishes; doing this makes no sense,

// as it is **the same as just working sequentially without threads**

// **threads[i].start();**

**// join() causes the current (main) thread to wait until thread[i] finishes**

**threads[i].join();**

}

}

catch(Exception e){

System.out.println(e.toString());

}

}

}

The way it is written, there will be **no output** because threads were created but **not started**, so threads will not be alive and join will not work.

If we add **threads[i].start()** right before join, the join means main thread will wait indefinitely for the thread to finish, so threads will be launched and executed **sequentially**, one after another. In this case, result will be:

**1 0 1 0 1 0 … 1 0 1 0**

**►► Implement Producer-Consumer design pattern using wait, notify and notifyAll?**

The **producer-consumer problem** (also known as the **bounded-buffer** problem) is a classic Java example of a multi-process **synchronization** problem. The problem describes two processes, the producer and the consumer, who share a common, fixed-size buffer used as a queue. The producer’s job is to generate a piece of data, put it into the buffer and start again. At the same time, the consumer is consuming the data (i.e., removing it from the buffer) one piece at a time.

The challenges are to make sure that the producer won’t try to add data into the buffer if it’s full and that the consumer won’t try to remove data from an empty buffer, produced items will not be left unconsumed and no starting items skipped.

public class ProducerConsumerSolution {

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

// Vector is thread-safe for its own operation e.g. add, remove or update,

// it will not help the producer to wait if queue is full or consumer to wait

// if queue is empty. Those are done by using wait and notify and

// there you need a common object to synchronize it.

Vector sharedQueue = new Vector();

int size = 4;

Thread prodThread = new Thread(new Producer(sharedQueue, size), "Producer");

Thread consThread = new Thread(new Consumer(sharedQueue, size), "Consumer");

prodThread.start();

consThread.start();

}

}

class Producer implements Runnable {

private final Vector sharedQueue;

private final int SIZE;

public Producer(Vector sharedQueue, int size) {

this.sharedQueue = sharedQueue;

this.SIZE = size;

}

@Override

public void run() {

for (int i = 0; i < 7; i++) {

System.out.println("Produced: " + i);

try {

produce(i);

} catch (InterruptedException ex) {

Logger.getLogger(Producer.class.getName()).log(Level.SEVERE, null, ex);

}

}

}

private void produce(int i) throws InterruptedException {

//wait if queue is full

while (sharedQueue.size() == SIZE) {

synchronized (sharedQueue) {

System.out.println("Queue is full " + Thread.currentThread().getName()

+ " is waiting , size: " + sharedQueue.size());

sharedQueue.wait();

}

}

//producing element and notify consumers

synchronized (sharedQueue) {

sharedQueue.add(i);

sharedQueue.notifyAll();

}

}

}

class Consumer implements Runnable {

private final Vector sharedQueue;

private final int SIZE;

public Consumer(Vector sharedQueue, int size) {

this.sharedQueue = sharedQueue;

this.SIZE = size;

}

@Override

public void run() {

while (true) {

try {

System.out.println("Consumed: " + consume());

Thread.sleep(50);

} catch (InterruptedException ex) {

Logger.getLogger(Consumer.class.getName()).log(Level.SEVERE, null, ex);

}

}

}

private int consume() throws InterruptedException {

//wait if queue is empty

while (sharedQueue.isEmpty()) {

synchronized (sharedQueue) {

System.out.println("Queue is empty " + Thread.currentThread().getName()

+ " is waiting , size: " + sharedQueue.size());

sharedQueue.wait();

}

}

//Otherwise consume element and notify waiting producer

synchronized (sharedQueue) {

sharedQueue.notifyAll();

return (Integer) sharedQueue.remove(0);

}

}

}