1. Please describe the steps of SpringBoot starts.

**Create the SpringApplication object and then call the run () method.**

**Step 1: at the beginning, it is a stopwatch class, which has a single function. It records the start-up time of springboot. After the completion of springboot, we will start the springboot on the console.**

**Step 2: create springapplicationrunlisteners**

**Step 3: start all listeners in springapplicationrunlisteners**

**Step 4: Environmental preparation**

**Step 5: print banner**

**Step 6: create context**

2. Program:

import java.math.BigDecimal;

import java.util.ArrayList;

import java.util.List;

public class Main {

public static void main(String[] args) {

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

float a = 0.1f;

float b = 0.1f;

float c = 0.1f;

float d = a + b + c;

float e = d \* 3;

float f = d \* 100000;

System.out.println("a + b + c = d = " + d);

System.out.println("e = " + e);

System.out.println("f = " + f);

}

}

The output of this program is

testWithFloats

a + b + c = d = 0.3

e = 0.90000004

f = 30000.002

Why e is not 0.9? Provide 2 ways to fix this problem.

**Floating point numbers cannot be stored precisely. Only numbers consisting of the power of 2 can be stored precisely**

**Solution1:Use BigDecimal to calculate  
public class Main {**

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

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

**BigDecimal a1 = new BigDecimal(0.1);**

**BigDecimal b1 = new BigDecimal(0.1);**

**BigDecimal c1 = new BigDecimal(0.1);**

**BigDecimal d1 = a1.add(b1).add(c1);**

**float e1 = new BigDecimal(3).multiply(d1).floatValue();**

**System.out.println("e1 = " + e1);**

**}**

**}**

**Solution2:transfer a,b,c to integer and use Int[] to implement triple Summation then transfer to origin digits.**

3. What is ReentrantLock and what are the pros and cons compare with synchronized keyword?

**1. Different from the underlying implementation, synchronized is the key word of Java language and the mutual exclusion at the level of native syntax, which needs to be implemented by JVM. Reentrantlock is an API level exclusive lock provided after JDK 1.5. It needs lock() and unlock() methods to cooperate with try / finally statement block. That is to say, synchronized implicitly obtains the release lock, and reentrantlock displays the acquisition and release lock.**

**2. When an exception occurs, synchronized will automatically release the lock occupied by the thread, so it will not cause deadlock; when an exception occurs, if lock does not actively release the lock through unlock(), it is likely to cause deadlock. Therefore, lock needs to be released in the finally block when lock is used.**

**3. Synchronized is synchronous blocking, pessimistic concurrency strategy is used, lock is synchronous non blocking, optimistic concurrency strategy is used.**

**4. Reentrantlock can make the thread waiting for the lock respond to the interrupt, but not synchronized. When synchronized is used, the waiting thread will wait all the time and cannot respond to the interrupt. Through reentrantlock, you can know whether the lock has been successfully acquired, while synchronized cannot. Most importantly, reentrantlock can provide fair locks, while synchronized can only be unfair locks.**

**5. Reentrantlock can improve the efficiency of read operation of multiple threads and realize read-write lock.**

**6. Reentrantlock can bind multiple conditions through condition.**

4. Print a binary tree in an m\*n 2D string array following these rules:

The row number m should be equal to the height of the given binary tree.

The column number n should always be an odd number.

The root node's value (in string format) should be put in the exactly middle of the first row it can be put. The column and the row where the root node belongs will separate the rest space into two parts (left-bottom part and right-bottom part). You should print the left subtree in the left-bottom part and print the right subtree in the right-bottom part. The left-bottom part and the right-bottom part should have the same size. Even if one subtree is none while the other is not, you don't need to print anything for the none subtree but still need to leave the space as large as that for the other subtree. However, if two subtrees are none, then you don't need to leave space for both of them.

Each unused space should contain an empty string "".

Print the subtrees following the same rules.

Example 1:

Input:

1

/

2

Output:

[["", "1", ""],

["2", "", ""]]

Example 2:

Input:

1

/ \

2 3

\

4

Output:

[["", "", "", "1", "", "", ""],

["", "2", "", "", "", "3", ""],

["", "", "4", "", "", "", ""]]

Example 3:

Input:

1

/ \

2 5

/

3

/

4

Output:

[["", "", "", "", "", "", "", "1", "", "", "", "", "", "", ""]

["", "", "", "2", "", "", "", "", "", "", "", "5", "", "", ""]

["", "3", "", "", "", "", "", "", "", "", "", "", "", "", ""]

["4", "", "", "", "", "", "", "", "", "", "", "", "", "", ""]]

Note: The height of binary tree is in the range of [1, 10].

/\*\*

\* Definition for a binary tree node.

\* public class TreeNode {

\* int val;

\* TreeNode left;

\* TreeNode right;

\* TreeNode(int x) { val = x; }

\* }

\*/

**class Solution {**

**public List<List<String>> printTree(TreeNode root) {**

**//1.求出root的高度**

**int maxDepth = getDepth(root);**

**//2.求出输出List的宽度**

**int width = 0, count = maxDepth;**

**while (count-- > 0) {**

**width = width \* 2 + 1;**

**}**

**//对结果集初始化**

**List<List<String>> res = new ArrayList<>(maxDepth);**

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

**List<String> list = new ArrayList<>();**

**for (int j = 0; j < width; j++) {**

**list.add("");**

**}**

**res.add(list);**

**}**

**//3.前序遍历，首先在结果集中填充左子树，然后填充右子树**

**helper(root, 1, 0, width, res);**

**return res;**

**}**

**private void helper(TreeNode root, int depth, int start, int end, List<List<String>> res) {**

**if (root == null || start > end) return;**

**//获取当前节点需要插入List的位置**

**int insert = start + (end - start) / 2;**

**//根据当前层数获得对应的List**

**//插入根节点**

**for (int i = start; i <= end; i++) {**

**if (i == insert) {**

**res.get(depth - 1).set(i, root.val + "");**

**break;**

**}**

**}**

**//递归打印左子树**

**helper(root.left, depth + 1, start, insert - 1, res);**

**helper(root.right, depth + 1, insert + 1, end, res);**

**}**

**private int getDepth(TreeNode root) {**

**if (root == null) return 0;**

**return Math.max(getDepth(root.left), getDepth(root.right)) + 1;**

**}**

**public class TreeNode {**

**int val;**

**TreeNode left;**

**TreeNode right;**

**TreeNode(int x) {**

**val = x;**

**}**

**}**

**}**

5. Write a function that reverses a string. The input string is given as an array of characters char[].

Do not allocate extra space for another array, you must do this by modifying the input array in-place with O(1) extra memory.

You may assume all the characters consist of printable ascii characters.

Example 1:

Input: ["h","e","l","l","o"]

Output: ["o","l","l","e","h"]

Example 2:

Input: ["H","a","n","n","a","h"]

Output: ["h","a","n","n","a","H"]

**class Solution {**

**public void reverseString(char[] s) {**

**int left = 0;**

**int right = s.length - 1;**

**char temp;**

**while (left < right){**

**temp = s[left];**

**s[left++] = s[right];**

**s[right--] = temp;**

**}**

**}**

**}**

6. Write a program to use 3 threads to print 1-99:

Example:

t1: 1

t2: 2

t3: 3

t1: 4

t2: …

**class Solution {**

**private boolean firstFinished;**

**private boolean secondFinished;**

**private boolean thirdFinished = true;**

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

**static int i = 1;**

**public void first(Thread printFirst) throws InterruptedException {**

**synchronized (lock) {**

**while(!thirdFinished){**

**lock.wait();**

**}**

**printFirst.run();**

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

**firstFinished = true;**

**thirdFinished = false;**

**lock.notifyAll();**

**}**

**}**

**public void second(Thread printSecond) throws InterruptedException {**

**synchronized (lock) {**

**while (!firstFinished) {**

**lock.wait();**

**}**

**printSecond.run();**

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

**secondFinished = true;**

**firstFinished = false;**

**lock.notifyAll();**

**}**

**}**

**public void third(Thread printThird) throws InterruptedException {**

**synchronized (lock) {**

**while (!secondFinished) {**

**lock.wait();**

**}**

**printThird.run();**

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

**thirdFinished = true;**

**secondFinished = false;**

**lock.notifyAll();**

**}**

**}**

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

**Thread r1 = new Thread("t1");**

**Thread r2 = new Thread("t2");**

**Thread r3 = new Thread("t3");**

**Solution s = new Solution();**

**while (i < 100){**

**s.first(r1);**

**s.second(r2);**

**s.third(r3);**

**}**

**}**

**}**

7. Describe lifecycle of a Java Bean in JVM. In JDK1.8 and above, how the memory is managed?

**Describe lifecycle of a Java Bean in JVM**

**1. Instantiate a bean - we often call it new;**

**2. Configure the instantiated bean according to the spring context - that is, IOC injection;**

**3. If the bean has implemented the beannameaware interface, the setbeanname (string) method it implements will be called, and the ID value of the bean in the spring configuration file will be passed here**

**4. If the bean has implemented the beanfactoryaware interface, the setbeanfactory (beanfactory) it implements will be called to pass the spring factory itself (you can get other beans in this way, just configure a common bean in the spring configuration file);**

**5. If the bean has implemented the applicationcontextaware interface, the setapplicationcontext (ApplicationContext) method will be called to pass in the spring context (the same method can also implement the content of step 4, but it is better than 4, because ApplicationContext is a sub interface of beanfactory, and there are more implementation methods);**

**6. If the bean is associated with the beanpostprocessor interface, the postprocessbeforeinitialization (object obj, string s) method will be called. The beanpostprocessor is often used as the change of the bean content. Because this method is called at the end of the bean initialization, it can also be applied to memory or cache technology;**

**7. If a bean has the init method attribute configured in the spring configuration file, it will automatically call its configured initialization method.**

**8. If the bean is associated with the beanpostprocessor interface, the postprocessafterinitialization (object obj, string s) method will be called;**

**Note: after the above work is completed, the bean can be applied. Then the bean is a singleton. Generally, we call the bean with the same ID in the same instance with the same content address. Of course, non singleton can also be configured in the spring configuration file. We won't elaborate here.**

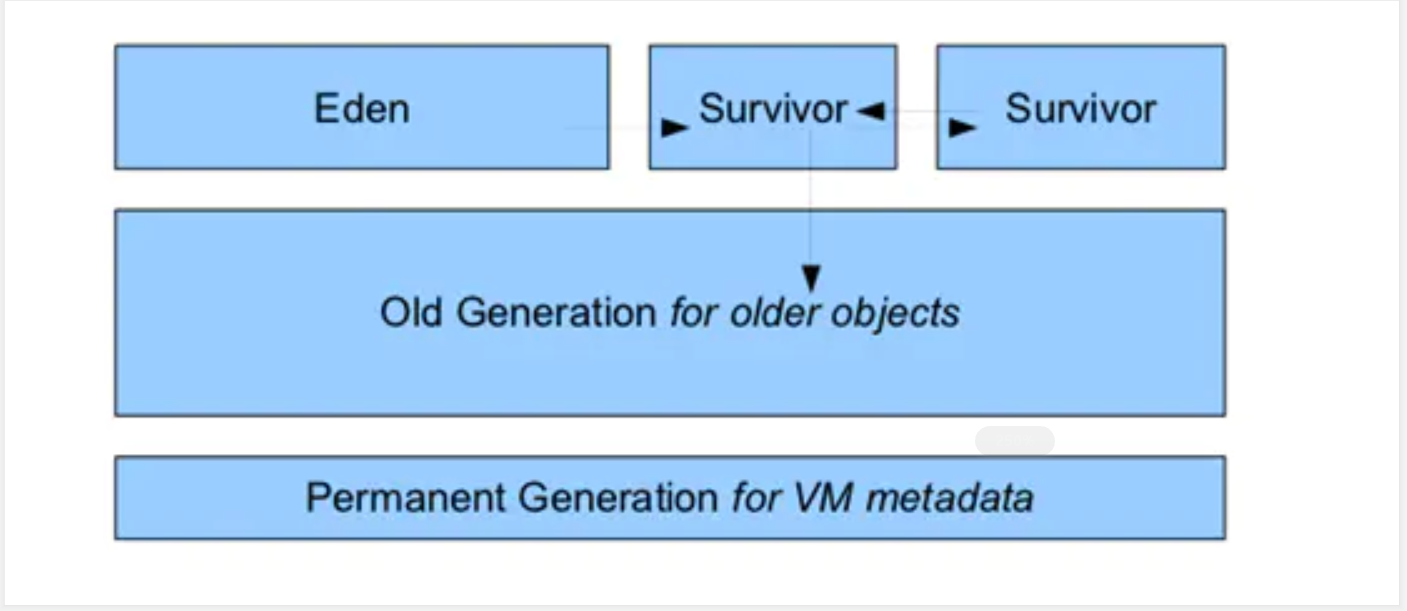
**9. When the bean is no longer needed, it will go through the cleaning stage. If the bean implements the disposablebean interface, it will call the destroy () method of its implementation;**

**10. Finally, if the destroy method attribute is configured in the spring configuration of this bean, the configured destroy method will be called automatically.**

----------------------------------------------------------------------------------------------------------------------------

**How the memory is managed**

**In JDK7 and its earlier versions of JDK, heap memory is usually divided into three parts: Nursery memory (young generation), old generation and permanent generation for VM matedata. The figure below is java7**

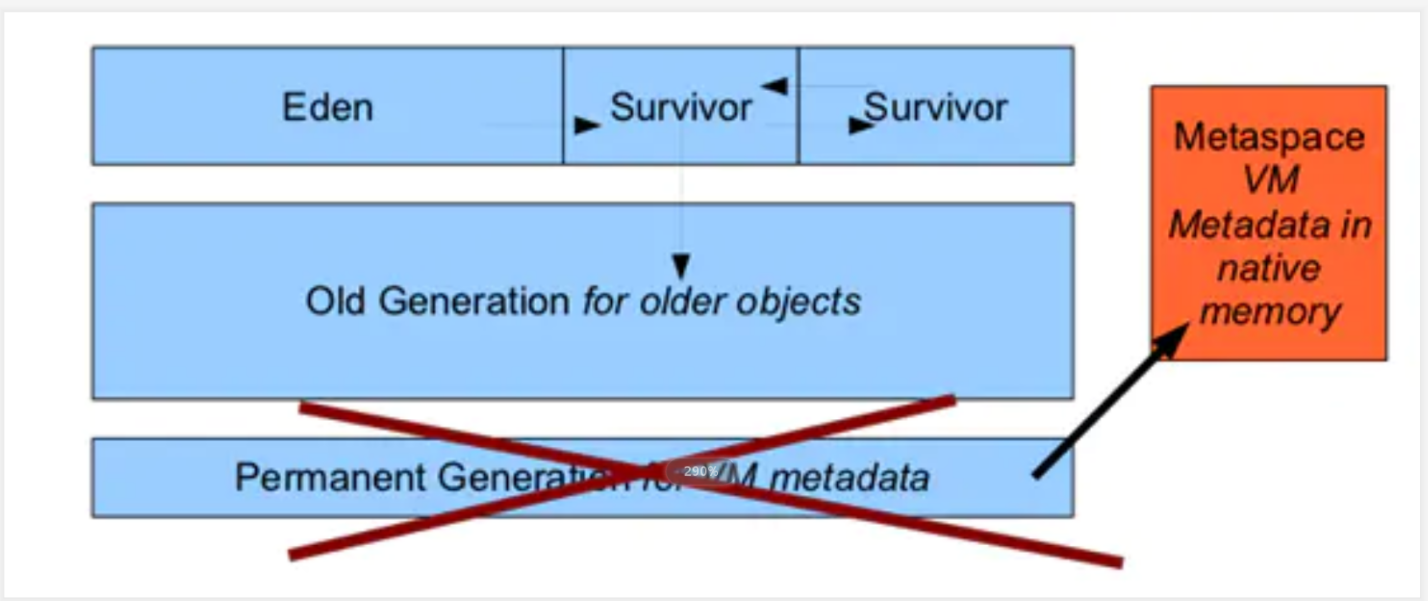
****

**The top layer is nursery memory. After an object is created, it is first put into Eden memory in nursery. If the lifetime exceeds two survivors, it will be transferred to old generation**

**Permanent memory stores metadata information such as methods and variables of objects. If there is not enough permanent memory, we will get the following error:**

**java.lang.OutOfMemoryError: PermGen**

**The situation in jdk8 has changed obviously, that is, you will not get this error in general. The reason is that the permanent memory in the metadata is moved from the heap memory to the local memory in jdk8. The JVM heap memory structure in jdk8 is as follows:**

****

**In this way, the permanent memory will no longer occupy the heap memory. It can avoid the common permanent memory errors (Java. Lang. outofmemoryerror: permgen) in JDK7 and previous versions through automatic growth. Maybe this is one of the reasons for upgrading your JDK to jdk8. Of course, jdk8 also provides a new parameter to set the memory size of matespace. Through this parameter, we can set the memory size of matespace. In this way, we can avoid excessive waste of local memory and achieve effective utilization according to the actual situation of our project.**

8. Reverse an integer:

For Example:

input: 123

output: 321

note the range of integer.

**class Solution {**

**public int reverse(int x) {**

**int rev = 0;**

**while (x != 0) {**

**int pop = x % 10;**

**x /= 10;**

**if (rev > Integer.MAX\_VALUE/10 || (rev == Integer.MAX\_VALUE / 10 && pop > 7)) return 0;**

**if (rev < Integer.MIN\_VALUE/10 || (rev == Integer.MIN\_VALUE / 10 && pop < -8)) return 0;**

**rev = rev \* 10 + pop;**

**}**

**return rev;**

**}**

**}**