**NO.1**

**Basic components of computer architecture:**

Input Unit, Output Unit, Storage Unit, Central Processing Unit (CPU), Arithmetic and Logic Unit (ALU), Control Unit.

**Memory Heap:**

Memory Heap is the run time area from which the memory for all java class instances and arrays is allocated. The heap is created when the JVM starts up and may increase or decrease in size while the application runs.

**When you compile and execute your program (eg: Hello.java), what happens? Provide details of execution environment.**

.java files are compiled into .class files and pushed into JVM through class loader. The loaded class will be stored in method area. The virtual machine executes the code in the method area. In the running process, whenever a Java method is called, the Java virtual machine generates a stack frame in the Java method stack of the current thread to store the local variables and the operands of bytecode.

**JVM memory architecture components:**

Heap area, Method area, Java VM Stack, Native Method Stack, PC Registers

**JVM Heap:**

The Java virtual machine has a heap, which is the runtime data area, from which all class instance and array memory is allocated. The heap is created when the Java virtual machine is started

**JVM Non-Heap:**

The non-heap memory is where JVM stores class-level information such as fields and methods of a class, method code, runtime constant pool and internalized Strings.

**Cache:**

Stores compiled code (i.e. native code) generated by JIT compiler, JVM internal structures, loaded profiler agent code and data, etc. -When Code Cache exceeds a threshold, it gets flushed (and objects are not relocated by the GC).

**Native Cache:**

Cache is a hardware or software component that stores data so that future requests for that data can be served faster.

**Java Native Interface (JNI):**

The Java Native Interface (JNI) is a foreign function interface programming framework that enables Java code running in a Java virtual machine (JVM) to call and be called by native applications (programs specific to a hardware and operating system platform) and libraries written in other languages.

**NO.2**

**Garbage collector (GC)** is an automatic memory management system that reclaims heap memory for objects.

**Young generation** is served for containing newly-allocated objects. Young generation includes three parts: Eden memory and two survivor memory spaces (S0, S1)

Most of the newly-created objects goes Eden space.

When Eden space is filled with objects, Minor GC (Young Collection) is performed and all survivor objects are moved to one of the survivor spaces.

Minor GC also checks survivor objects and move them to other survivor space. So at any time, one of the survivor space is always empty.

Objects that are survived after many cycles of GC, are moved to the Old generation memory space. Usually it’s done by setting a threshold for the age of the young generation objects before they become eligible to promote to Old generation.

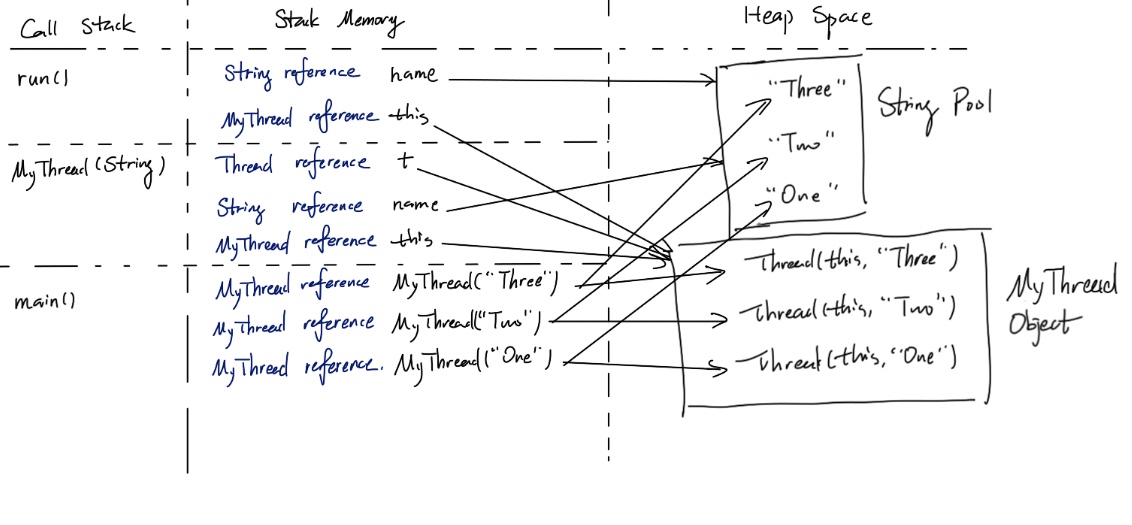
**Old generation** is reserved for containing long lived objects that could survive after many rounds of Minor GC

When Old generation space is full, Major GC (Old Collection) is performed (usually takes longer time)

**Eden Space:** The pool from which memory is initially allocated for most objects

**Survivor Space:** The pool containing objects that have survived the garbage collection of the Eden space.

**NO.4**



**NO.5**

A) When Thread 1 calls addChild() method, it has a lock on List children, and when it runs to child.setParentOnly(this), it ask for a lock of TreeNode parent. At the same time, Thread 2 calls setParent(TreeNode parent), it has a lock on TreeNode parent, and when it runs to parent.addChildOnly(this), it ask for a lock of List children. At that time, thread 1 is asking for a list that is already been locked by thread 2, and thread 2 is asking for a treenode that is already been locked by thread 1. Then a deadlock occurs.

C) No. All of the methods in the class are described as synchronized which means they cannot be accessed by multiple threads at the same time. So that race condition is not possible in this code.

**NO.6**

**1. ClassLoader Subsystem**

Java's dynamic class loading functionality is handled by the ClassLoader subsystem. It loads, links. and initializes the class file when it refers to a class for the first time at runtime, not compile time.

**1.1 Loading**

Classes will be loaded by this component. BootStrap ClassLoader, Extension ClassLoader, and Application ClassLoader are the three ClassLoaders that will help in achieving it.

**BootStrap ClassLoader** – Responsible for loading classes from the bootstrap classpath, nothing but rt.jar. Highest priority will be given to this loader.

Extension ClassLoader – Responsible for loading classes which are inside the ext folder (jre\lib).

**Application ClassLoader** –Responsible for loading Application Level Classpath, path mentioned Environment Variable, etc.

The above ClassLoaders will follow Delegation Hierarchy Algorithm while loading the class files.

**1.2 Linking**

**Verify** – Bytecode verifier will verify whether the generated bytecode is proper or not if verification fails we will get the verification error.

**Prepare** – For all static variables memory will be allocated and assigned with default values.

**Resolve** – All symbolic memory references are replaced with the original references from Method Area.

**1.3 Initialization**

This is the final phase of ClassLoading; here, all static variables will be assigned with the original values, and the static block will be executed.

**2. Runtime Data Area**

The Runtime Data Area is divided into five major components:

**Method Area** – All the class-level data will be stored here, including static variables. There is only one method area per JVM, and it is a shared resource.

**Heap Area** – All the Objects and their corresponding instance variables and arrays will be stored here. There is also one Heap Area per JVM. Since the Method and Heap areas share memory for multiple threads, the data stored is not thread-safe.

**Stack Area** – For every thread, a separate runtime stack will be created. For every method call, one entry will be made in the stack memory which is called Stack Frame. All local variables will be created in the stack memory. The stack area is thread-safe since it is not a shared resource. The Stack Frame is divided into three subentities:

Local Variable Array – Related to the method how many local variables are involved and the corresponding values will be stored here.

**Operand stack** – If any intermediate operation is required to perform, operand stack acts as runtime workspace to perform the operation.

**Frame data** – All symbols corresponding to the method is stored here. In the case of any exception, the catch block information will be maintained in the frame data.

**PC Registers** – Each thread will have separate PC Registers, to hold the address of current executing instruction once the instruction is executed the PC register will be updated with the next instruction.

**Native Method stacks** – Native Method Stack holds native method information. For every thread, a separate native method stack will be created.

**3. Execution Engine**

The bytecode, which is assigned to the Runtime Data Area, will be executed by the Execution Engine. The Execution Engine reads the bytecode and executes it piece by piece.

**Interpreter** – The interpreter interprets the bytecode faster but executes slowly. The disadvantage of the interpreter is that when one method is called multiple times, every time a new interpretation is required.

**JIT Compiler** – The JIT Compiler neutralizes the disadvantage of the interpreter. The Execution Engine will be using the help of the interpreter in converting byte code, but when it finds repeated code it uses the JIT compiler, which compiles the entire bytecode and changes it to native code. This native code will be used directly for repeated method calls, which improve the performance of the system.

**Intermediate Code Generator** – Produces intermediate code

**Code Optimizer** – Responsible for optimizing the intermediate code generated above

Target Code Generator – Responsible for Generating Machine Code or Native Code

**Profiler** – A special component, responsible for finding hotspots, i.e. whether the method is called multiple times or not.

**Garbage Collector:** Collects and removes unreferenced objects. Garbage Collection can be triggered by calling System.gc(), but the execution is not guaranteed. Garbage collection of the JVM collects the objects that are created.

**Java Native Interface (JNI):** JNI will be interacting with the Native Method Libraries and provides the Native Libraries required for the Execution Engine.

**Native Method Libraries:** This is a collection of the Native Libraries, which is required for the Execution Engine.