3-technical-questions

Contents

Important topics **1**
- Structures . . . 2
- Algorithms . . . 2
- Concepts . . . 2

Solving process **2**

Compilation & Interpretation **3**
- Key & Core Differences . . . 3
- Compiled & Interpreted . . . 4
- Java virtual machine . . . 4
  - Introduction and description . . . 4
  - Key features and processes . . . 5
  - Memory management & control . . . 6
  - Garbage collection algorithms . . . 7
  - Garbage collection strategies . . . 8

Trick questions **8**

- Important topics
  - Structures
  - Algorithms
  - Concepts
- Solving process
- Compilation & Interpretation
  - Key & Core Differences
  - Compiled & Interpreted
  - Java virtual machine
    - Introduction and description
    - Key features and processes
    - Memory management & control
    - Garbage collection algorithms
    - Garbage collection strategies
- Trick questions

Important topics

For each of the topics below make sure you understand how to use and implement them, on paper and then on a computer.

Structures

1. Linked lists
2. Trees, Tires and Graphs
3. Stacks & Queues
4. Heaps
5. Vectors / Arrays
6. Hash tables

Algorithms

1. Breadth-First Search
2. Depth-First Search
3. Binary Search
4. Merge Sort
5. Quick Sort
6. Other Sorting algorithms

Concepts

1. Bit manipulation
2. Memory (Stack & Heap)
3. Recursion
4. Dynamic programming
5. Big O time & space

Solving process

Make sure to understand the problem , ask for paper and pen and record the problem on it, write it down, do not try to memorize it otherwise you will miss important details. Ask for example, draw and write down it too.

When drawing an example make it as specific to the problem as possible, use the data you have been given, or make up one yourself, but try to make the example about the problem, do not draw generic trees, stacks etc. Make it large so you can see patterns, edge cases etc.

Try to trade off space or time for the other e.g. - store data (hash-table, vector, list, stack, tree etc.), sort data (mutate the input example, by sorting it to fit in the problem's solution).

Sometimes it might be useful to write down an implementation that solves the issue incorrectly, or not completely.

1. Listen - pay very close attention to any information, in the problem description, ask for more details, as much as you can think of to narrow down the problem at hand, try to make no assumptions about the problem.
2. Example - ask for example, more than one if you have to, be aggressive with the questions, to allow for no foul play
3. Brute force - try to solve the problem in the most naive obvious way, that comes to mind
4. Optimize - try to apply different approaches to the problem solution afterwards, such as dynamiting programming or memorization etc - avoid bottlenecks, unnecessary work or duplicated work
5. Walk through - analyze the problem, try to describe it, make sure it makes sense, if not, go back to the example and start over, from step 2
6. Implement - make sure whatever you implement is well structured, now that it

7. Test - go through your code and test it, if you have to do any changes to adjust the logic go back to step 3

Compilation & Interpretation

- Compiled Languages: In strictly compiled languages, such as C and C++, source code is translated directly into machine code by a compiler before execution. The resulting machine code is specific to the target hardware architecture.
    1. Compilation Process: The compilation process involves translating the entire source code into machine code in one pass. This generates an executable file containing native machine code that can be directly executed by the CPU.
    2. Execution: Strictly compiled languages produce standalone executables that are independent of the original source code. The compiled code is executed directly by the CPU without the need for an intermediate runtime environment.
- Interpreted Languages: In interpreted languages, such as Python and JavaScript, source code is executed line by line by an interpreter at runtime. The interpreter reads and executes each statement of the source code sequentially.
    1. Interpretation Process: The interpreter reads the source code, parses it into intermediate representations (such as bytecode or AST), and executes it immediately. There is no separate compilation step to generate machine code.
    2. Execution: Interpreted languages typically require an interpreter to be installed on the target system to execute the source code. The interpreter translates and executes the source code on-the-fly, without producing standalone executables.

Key & Core Differences

1. Compilation vs. Interpretation:
    - Compiler: Translates entire source code into machine code before execution.
    - Interpreter: Executes source code line by line at runtime without prior translation.
2. Output:
    - Compiler: Produces standalone executable files containing native machine code.
    - Interpreter: Executes source code directly without generating standalone executables.
3. Execution Speed:
    - Compiler: Generally produces faster-executing code as it is optimized for the target hardware architecture.
    - Interpreter: May have slower execution speed compared to compiled languages due to the overhead of interpretation at runtime.
4. Portability:
    - Compiler: Generates machine code specific to the target hardware architecture, potentially limiting portability.
    - Interpreter: Source code is platform-independent, and the interpreter translates and executes code on-the-fly, allowing for greater portability.

In summary, strictly compiled languages translate source code into machine code before execution, resulting in standalone executables, while interpreted languages execute source code directly at runtime using an interpreter.

Compiled & Interpreted

Several interpreted languages utilize Just-In-Time (JIT) compilation techniques to improve performance. Some examples of interpreted languages with JIT compilation include:

- JavaScript (V8 Engine): JavaScript is commonly used in web development, and modern JavaScript engines such as Google's V8 engine (used in Chrome and Node.js) employ JIT compilation to dynamically compile frequently executed JavaScript code into native machine code for improved performance.
- Python (PyPy): While Python is traditionally interpreted, projects like PyPy implement a JIT compiler for Python code. PyPy's JIT compiler (Just-In-Time Compiler for Python) dynamically compiles Python bytecode into native machine code at runtime, resulting in significant performance improvements over traditional interpreters.
- Ruby (YARV): The YARV (Yet Another Ruby VM) interpreter for Ruby includes a JIT compiler called MJIT (Method JIT). MJIT dynamically compiles Ruby methods into native machine code, providing performance enhancements for Ruby applications.
- PHP (HHVM): The HipHop Virtual Machine (HHVM), developed by Facebook, includes a JIT compiler for PHP code. HHVM's JIT compiler dynamically compiles PHP bytecode into native machine code, offering performance improvements over traditional PHP interpreters.
- Lua (LuaJIT): LuaJIT is a Just-In-Time (JIT) compiler for the Lua programming language. It dynamically compiles Lua bytecode into highly optimized native machine code at runtime, resulting in significant performance gains for Lua applications.
- Java (JVM): like HotSpot, incorporate Just-In-Time (JIT) compilation. The JVM dynamically compiles frequently executed bytecode sequences into optimized native machine code during runtime. This compilation process aims to improve the performance of Java applications by translating bytecode into efficient machine code

Java virtual machine

Introduction and description

Several implementations of the Java Virtual Machine (JVM) have been developed by different organizations and communities. Here are some notable implementations:

1. Oracle HotSpot VM: Developed and maintained by Oracle Corporation. HotSpot is the default JVM implementation included in Oracle's JDK (Java Development Kit) distributions. It is known for its high performance, advanced optimization techniques, and support for features like Just-In-Time (JIT) compilation and multiple garbage collection algorithms.
2. OpenJ9: Developed and maintained by the Eclipse Foundation (formerly IBM). OpenJ9 is an open-source JVM implementation that focuses on high performance, low memory footprint, and fast startup times. It is included in Eclipse Adoptium (formerly AdoptOpenJDK) distributions and used in various cloud and enterprise Java environments.
3. GraalVM: Developed and maintained by Oracle Labs. GraalVM is a high-performance polyglot virtual machine that supports multiple programming languages, including Java, JavaScript, Python, Ruby, and others. It includes the GraalVM Native Image technology, which enables ahead-of-time compilation of Java applications into native executables for improved startup time and reduced memory overhead.
4. Azul Zulu JVM: Developed and maintained by Azul Systems. Zulu is a certified, fully compatible, and open-source JVM implementation based on the OpenJDK project. It is available for various platforms, including Linux, Windows, macOS, and Docker, and is commonly used in enterprise environments.

5. Amazon Corretto JVM: Developed and maintained by Amazon Web Services (AWS). Corretto is a no-cost, multiplatform, production-ready distribution of OpenJDK. It is designed to provide long-term support, security updates, and performance enhancements for Java applications running on AWS cloud services and on-premises.

6. SAP JVM: Developed and maintained by SAP SE. SAP JVM is a high-performance, highly optimized JVM implementation designed for enterprise-grade Java applications. It includes features like advanced garbage collection algorithms, memory management optimizations, and support for SAP application platforms.

Key features and processes

1. The Java Virtual Machine (JVM) executes Java bytecode, which is an intermediate representation of Java code compiled from source code (.java files). Here's an overview of the pipeline for executing Java code on the JVM:

2. Java Source Code Compilation: Java source code files (.java) are compiled into Java bytecode files (.class) using the Java compiler (javac). The Java compiler translates Java source code into platform-independent bytecode instructions, which are stored in .class files.

3. Class Loading: The JVM loads classes (.class files) into memory as they are needed during program execution. Class loading involves three steps: loading, linking, and initialization. Loading: The class loader reads the bytecode from the .class file and creates a representation of the class in memory. Linking: The class loader performs verification, preparation (e.g., allocating memory for static fields), and resolution (e.g., resolving symbolic references to other classes). Initialization: The JVM initializes the class by executing static initializers and initializing static fields.

4. Bytecode Verification: Before executing bytecode, the JVM performs bytecode verification to ensure that the bytecode adheres to certain safety constraints and does not violate the Java language rules. Bytecode verification checks for various properties, such as type safety, stack integrity, and proper control flow.

5. Just-In-Time (JIT) Compilation: The JVM employs a Just-In-Time (JIT) compiler to translate bytecode into native machine code at runtime. When a method is called frequently or identified as a hot spot during profiling, the JIT compiler compiles the bytecode of the method into optimized native machine code. The native machine code is then cached and executed directly by the CPU, bypassing interpretation of the bytecode. JIT compilation aims to improve performance by dynamically optimizing frequently executed code paths.

6. Execution: The JVM executes the compiled native machine code, which corresponds to the bytecode instructions of the Java program. Execution involves interpreting bytecode or executing the compiled native machine code, depending on whether JIT compilation has been performed for specific code paths.

7. Garbage Collection: The JVM manages memory allocation and deallocation using automatic garbage collection. Garbage collection identifies and reclaims memory that is no longer in use, preventing memory leaks and ensuring efficient memory usage.

Here is a very simple example of the different steps the java virtual machine might go through to generate byte or machine code from an input java source

1. Java source code

```
public class Calculator {
    public static int sum(int a, int b) {
        return a + b;
    }
```

```
    }
```

2. Bytecode generation

```
    javac Calculator.java
    javap -c Calculator
```

3. Java byte code

```
    public class Calculator {
        public static int sum(int, int);
            Code:
            0: iload_0
            1: iload_1
            2: iadd
            3: ireturn
    }
```

4. Native machine code

```
    sum:
        mov eax, DWORD PTR [rsp+0x8]    ; Load the value of the first
           argument (a)
        add eax, DWORD PTR [rsp+0xc]    ; Add the value of the second
           argument (b)
        ret                             ; Return the result
```

After JIT compilation, the bytecode instructions are translated into native machine code. The native machine code for the sum method is represented using x86 assembly language here. Instructions such as mov and add perform similar operations to the bytecode instructions but are specific to the native architecture. The ret instruction is used to return the result from the method.

Memory management & control

Below are presented some common concepts about memory management in a garbage collected language such as java, most of the principles below apply for most other languages using similar approaches

1. **What is Garbage Collection in Java?** Garbage collection in Java is an automatic memory management process where the JVM automatically reclaims memory occupied by objects that are no longer in use or reachable by the program. It helps developers by eliminating the need to manually free memory and reduces the risk of memory leaks.

2. **How does Java's Garbage Collector work?** Java's garbage collector works by periodically identifying and reclaiming memory occupied by objects that are no longer reachable or needed by the program. It uses various algorithms like Mark-Sweep, Copying, and Generational to efficiently manage memory. The process involves marking objects that are still in use, reclaiming memory occupied by unreachable objects, and compacting memory to reduce fragmentation.

3. **What are the different types of garbage collectors in Java?** Java offers different garbage collection algorithms to suit different application needs. These include Serial, Parallel, CMS (Concurrent Mark-Sweep), G1 (Garbage-First), and ZGC (Z Garbage Collector). Each algorithm has its own characteristics such as throughput, latency, and suitability for different application scenarios.

4. **What is the difference between System.gc() and Runtime.getRuntime().gc()?** System.gc() and Runtime.getRuntime().gc() are both used to suggest garbage collection, but they differ in their

invocation. System.gc() is a static method that suggests garbage collection to the JVM, while Runtime.getRuntime().gc() is an instance method that provides a reference to the runtime object before suggesting garbage collection.

5. **How do you prevent memory leaks in Java?** Memory leaks in Java can be prevented by following best practices such as releasing resources in a timely manner, closing streams properly using try-with-resources, being cautious with static variables, avoiding excessive object creation, and using memory profiling tools to identify potential leaks.

6. **Explain the concept of memory leak in Java?** A memory leak occurs in Java when objects are no longer needed by the application but are still referenced, preventing them from being garbage collected. This leads to a gradual increase in memory usage over time, potentially causing OutOfMemoryErrors and performance degradation.

7. **What is the purpose of the finalize() method in Java?** The finalize() method in Java is called by the garbage collector before reclaiming an object's memory. Its purpose is to perform any necessary cleanup or resource releasing operations. However, it's generally not recommended to rely on finalize() for resource cleanup due to its unpredictable nature and performance implications.

8. **How can you monitor and tune garbage collection in Java applications?** Garbage collection in Java applications can be monitored and tuned using various tools such as VisualVM, JConsole, and Mission Control. These tools provide insights into garbage collection activity, heap usage, and performance metrics. Tuning can involve adjusting JVM parameters like heap size, garbage collector type, and GC tuning flags based on application requirements and performance goals.

9. **What is the impact of garbage collection on application performance?** Garbage collection can have a significant impact on application performance, affecting factors such as CPU overhead, pause times, and throughput. Proper garbage collection tuning is essential to balance these factors and optimize application responsiveness and scalability.

10. **What are the following class types and their purpose?** `WeakReference`, `SoftReference`, and `PhantomReference` are types of reference objects in Java used for more controlled object lifecycle and memory management. - `WeakReference` objects allow objects to be garbage collected when no longer referenced strongly. - `PhantomReference` objects provide more control over cleanup actions without interfering with garbage collection. These reference types are particularly useful in scenarios like caching and managing native resources. - `SoftReference` objects are garbage collected only when memory is low.

Garbage collection algorithms

Mark and Sweep

- Mark Phase: The first phase, known as the "mark" phase, involves traversing all reachable objects starting from the roots (e.g., global variables, local variables, and active threads). During the traversal, each reachable object is marked or flagged as "in-use" or "reachable". This phase identifies all objects that are still actively being used by the program and marks them for retention.

- Sweep Phase: The second phase, known as the "sweep" phase, involves scanning the entire heap memory. During this phase, memory regions not marked as "in-use" are considered garbage and are eligible for reclamation. The garbage collector then reclaims the memory occupied by these unreachable objects, effectively freeing it up for future allocations.

- Advantages: Mark and Sweep is straightforward and relatively easy to implement. It doesn't require moving objects around in memory, which can simplify memory management in certain scenarios. It can handle cycles in object references (objects referencing each other) without additional complexity.

- Disadvantages: Mark and Sweep can cause fragmentation in memory since reclaimed memory may not be contiguous. It typically involves stop-the-world pauses during the mark and sweep phases, which can lead to application pauses and reduced responsiveness in interactive applications. It doesn't perform memory compaction, which means it may not efficiently utilize available memory space, especially in long-running applications.

Garbage collection strategies

1. **Serial Garbage Collector:** The Serial Garbage Collector (also known as the Serial Collector) is the simplest garbage collector. It is best suited for single-threaded applications or small-scale applications with small heaps. This collector is also known for its simplicity and low overhead but can cause noticeable pauses during garbage collection, as it stops all application threads during the collection process. It uses a mark-and-sweep algorithm.
2. **Parallel Garbage Collector:** The Parallel Garbage Collector (also known as the Throughput Collector) is designed for multi-threaded applications and is the default garbage collector for the Java HotSpot VM. It uses multiple threads to perform garbage collection, which helps in reducing the pause times experienced by applications. This collector is suitable for applications where the goal is to maximize throughput and performance rather than minimizing pause times. It uses a mark-and-sweep algorithm.
3. **Garbage-First (G1) Garbage Collector:** The G1 Garbage Collector is designed to provide both low-latency pauses and high throughput. It divides the heap into regions and performs garbage collection on these regions independently, allowing it to collect only the regions with the most garbage first. G1 aims to meet garbage collection pause time goals with predictable and consistent pause times. It is suitable for large heap applications and applications that require low-latency garbage collection.
4. **Z Garbage Collector (ZGC):** ZGC is a scalable garbage collector designed to provide low-latency performance for large heaps (multi-terabyte heaps). It is intended for applications requiring very low pause times, typically less than 10ms, even on very large heaps. ZGC uses concurrent algorithms for all major phases of garbage collection, including marking, reference processing, relocation, and evacuation. It is suitable for applications with stringent latency requirements, such as financial trading systems, gaming platforms, and real-time analytics.

Trick questions

1. NullPointerException with Autoboxing:

```
Integer i = null;
int j = i; // What happens here?
```

   This code will throw a NullPointerException because autoboxing attempts to unbox a null reference to an int.

2. ConcurrentModificationException with Iterators:

```
List<String> list = new ArrayList<>();
list.add("a");
list.add("b");
for (String s : list) {
    list.remove(s); // What happens here?
}
```

   This code will throw a ConcurrentModificationException because you're trying to modify the list while iterating over it without using an iterator's remove() method.

3. ArithmeticException with Division:

```
int result = 5 / 0; // What happens here?
```

This code will throw an ArithmeticException because dividing by zero is undefined in Java.

4. IndexOutOfBoundsException with Arrays:

```
int[] array = new int[5];
int value = array[5]; // What happens here?
```

This code will throw an IndexOutOfBoundsException because array indices in Java are zero-based, and trying to access index 5 in an array of length 5 is out of bounds.

5. ClassCastException with Generics:

```
List<String> strings = new ArrayList<>();
List<Object> objects = (List<Object>) strings; // What happens here?
```

This code will compile without error, but it will throw a ClassCastException at runtime because you cannot cast a generic type to another generic type with different type arguments due to type erasure.

6. NullPointerException with String Comparison:

```
String str = null;
if (str.equals("hello")) { // What happens here?
    System.out.println("Hello");
}
```

This code will throw a NullPointerException because you're trying to call the equals() method on a null reference.

7. NumberFormatException with String to Integer Conversion:

```
String str = "abc";
int num = Integer.parseInt(str); // What happens here?
```

This code will throw a NumberFormatException because "abc" cannot be parsed into an integer.

8. UnsupportedOperationException with Unmodifiable Collections:

```
List<String> list = Collections.emptyList();
list.add("item"); // What happens here?
```

This code will throw an UnsupportedOperationException because Collections.emptyList() returns an immutable (unmodifiable) list.

9. StackOverflowError with Recursion:

```
public void recursiveMethod() {
    recursiveMethod(); // What happens here?
}
```

This code will cause a StackOverflowError because it creates an infinite recursion without an exit condition.

10. IllegalArgumentException with Enum.valueOf():

```
public enum Color { RED, GREEN, BLUE }
Color color = Enum.valueOf(Color.class, "YELLOW"); // What happens here?
```

This code will throw an IllegalArgumentException because there is no enum constant named "YELLOW" in the Color enum.

11. EmptyStackException from stack.pop()

```
Stack<Integer> stack = new Stack<>();
int element = stack.pop(); // What happens here?
```

Trying to pop an element from an empty stack (pop() on an empty stack) will throw an EmptyStackException.

12. Integer Overflow:

```
    int value = Integer.MAX_VALUE;
    value++; // What happens here?
```

Incrementing Integer.MAX_VALUE will wrap around to Integer.MIN_VALUE, causing overflow.

13. String Pool Behavior:

```
    String s1 = "hello";
    String s2 = "hello";
    System.out.println(s1 == s2); // What is the output?
```

Strings created using string literals are pooled, so s1 and s2 will refer to the same object, resulting in true when comparing them with ==.

14. Floating-Point Precision:

```
    double result = 0.1 + 0.2; // What is the value of result?
```

Due to floating-point precision, the result of 0.1 + 0.2 is not exactly 0.3 but a close approximation due to binary representation limitations.

15. Equals Method with String and StringBuilder:

```
    String str1 = "hello";
    StringBuilder str2 = new StringBuilder("hello");
    System.out.println(str1.equals(str2)); // What is the output?
```

The equals() method in Java compares the content of objects. Since str1 is a String and str2 is a StringBuilder, they will not be equal even if they contain the same characters.

16. Casting and Arithmetic Operations:

```
    byte a = 127;
    byte b = 127;
    byte c = a + b; // What happens here?
```

The result of adding two bytes (a and b) will be promoted to an int, so assigning it to a byte (c) without explicit casting will cause a compilation error.

17. Object Comparison:

```
    Integer x = new Integer(100);
    Integer y = new Integer(100);
    System.out.println(x == y); // What is the output?
```

Comparing x and y with == checks for reference equality. Since x and y are different objects (even though they contain the same value), the result will be false.