JAVA INTERVIEW

Q: What's the use of *System.gc()* ?

A: In Java, *gc()* is a method of the *System* class that requests the JVM's garbage collector to perform a garbage collection. Garbage collection is the process by which the JVM frees up memory that is no longer being used by an application.

When an application creates objects, the JVM allocates memory to store those objects. As the application runs, it may no longer need some of the objects that were created earlier. Without garbage collection, this memory would be allocated indefinitely, eventually leading to a memory overflow and application crash.

```java
public class App {

    public static void main(String[] args) {
        // Allocate some objects
        Object obj1 = new Object();
        Object obj2 = new Object();

        // Dereference one of the objects
        obj1 = null;

        // Request a garbage collection
        System.gc();

        // Do some other work...

        // The garbage collector may not have freed the memory yet, so we can wait
        // a little while to allow it to complete
        try {
            Thread.sleep( millis: 1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

}
```

The garbage collector in the JVM automatically frees up memory that is no longer being used by an application. When a garbage collection is triggered, the garbage collector identifies which objects are still in use by the application and which can be safely freed, and then deallocates the memory used by the unreferenced objects.

While the garbage collector in the JVM is designed to run automatically, there may be situations where an application needs to explicitly request a garbage collection. This is where the *gc()* method

comes in - it allows an application to request that the garbage collector run a collection immediately.

It's important to note that while the *gc()* method can be used to request a garbage collection, it is not guaranteed to immediately free up all unused memory. The garbage collector in the JVM is a complex system that runs automatically and optimizes memory usage based on a variety of factors, including memory usage patterns and available system resources. As such, it may take several garbage collections before all unused memory is freed.

Q: What's the off-heap memory?

A: Off-heap memory in Java applications refers to memory that is not managed by the Java Virtual Machine (JVM) heap. It is memory that is allocated outside of the JVM, typically using native code or libraries.

The main advantage of off-heap memory is that it can be used to store large amounts of data that would not fit in the JVM heap, without affecting the JVM's garbage collection performance. Off-heap memory can also be shared across multiple JVM instances, or between different applications running on the same machine, which can help reduce memory usage and improve system performance.

Examples of off-heap memory usage in Java applications include memory-mapped files, direct buffers, and shared memory regions. Off-heap memory can be managed using specialized APIs such as the Java Native Interface (JNI), or third-party libraries such as the DirectMemory or Chronicle Map.

It's worth noting that using off-heap memory can be more complex than using heap memory, as developers are responsible for managing the memory allocation and deallocation themselves. Additionally, accessing off-heap memory requires additional synchronization and serialization mechanisms, which can add overhead to application performance.

The off-heap memory stores the data outside the heap in OS memory part. The data must be stored in specific format that is an array of bytes. So, using the off-heap memory in JVM languages programs introduces the overhead of serializing/deserializing these arrays to corresponding objects every time with additional cost of going outside the JVM and dealing with native memory.

In additional, off-heap memory helps the data to survive JVM crashes. With that it's possible to have a long living hot cache.

But sometimes it's not enough, especially when we need to: cache a lot of data without increasing GC pauses, share cached data between JVMs or add a persistence layer in memory resistant to JVM crashes. In all mentioned cases off-heap memory is one of possible solutions.

But off-heap memory is not the solution in all cases:

- Still short-lived objects (= never promoted to old generation) are better suited for on-heap storage simply because of the simplicity guaranteed by this automatic management.
- Moreover, the JIT can make several optimizations for memory use (e.g., some objects allocation can be skipped thanks to Escape Analysis).
- In addition, off-heap storage involves serialization/deserialization overhead (we can save only arrays of bytes) that doesn't exist in on-heap objects storage.
- Off-heap storage means the manual management of the memory. Sometimes it can lead to memory leaks, seg faults or other uncommon problems in the life of "on-heap Java programmer".

Q: Describe the cases for which we need to use the Off-heap memory

A: Off-heap memory in Java refers to the memory outside of the Java heap that is managed directly by the operating system. Here are some cases where off-heap memory can be useful in Java applications:

1. Large data sets: If your application needs to store and manipulate large data sets, off-heap memory can be used to store the data. Since the off-heap memory is not subject to garbage collection, it can help to avoid the performance overhead of frequent garbage collections.
2. Caching: If your application needs to cache data that is frequently accessed, off-heap memory can be used to store the data. The data can be quickly accessed without the overhead of deserializing it from disk or network.
3. Low-latency applications: In low-latency applications, every millisecond counts. Off-heap memory can be used to store frequently accessed data structures, such as message queues or connection pools, to reduce the latency of the application.
4. Security: Off-heap memory can be used to store sensitive data, such as passwords or encryption keys. Since the off-heap memory is not subject to garbage collection, it can help to reduce the risk of the sensitive data being exposed in a heap dump.
5. Interoperability: If your application needs to interact with native code, off-heap memory can be used to pass data between the Java code and native code. The off-heap memory can be accessed directly by the native code, without the overhead of copying the data into the Java heap.

It's worth noting that using off-heap memory requires careful management and can be more complex than using the Java heap. Additionally, off-heap memory is not subject to the same safety checks as the Java heap, so it's important to ensure that the application properly handles memory allocation and deallocation.

Q: How off-heap memory can be used for caching?

A: To use off-heap memory for caching, you can allocate a block of memory outside the JVM's heap using native memory APIs, such as the Java Native Interface (JNI) or the Unsafe class. You can then use this memory to store your cache data. The data can be quickly accessed without the overhead of deserializing it from disk or network.

Q: What’s the Object pooling and eviction?

A: Object pooling is a design pattern in which a pool of pre-allocated objects is maintained, and these objects are reused as needed, rather than being created and destroyed each time they are needed. This can be particularly useful in situations where object creation and destruction are expensive operations, such as in a high-concurrency Java application.

In Java, object pooling can be implemented using various techniques such as using a custom pool class or using third-party libraries such as Apache Commons Pool. The pool class maintains a pool of pre-allocated objects, and these objects can be checked out by client code when they are needed. Once the client code has finished using the object, it returns it to the pool so that it can be reused by other clients.

Eviction is a related concept in object pooling that refers to the process of removing objects from the pool when they are no longer needed. This can be important in situations where the pool has a limited capacity or where objects in the pool have a limited lifespan. For example, in a database connection pool, connections may be evicted from the pool after a certain period or after a certain number of uses.

The eviction process in object pooling can be implemented using various strategies, such as:

1. *LRU (Least Recently Used)*: Objects that have not been used for the longest time are evicted from the pool.
2. *FIFO (First In First Out)*: Objects that have been in the pool for the longest time are evicted first.
3. *Soft Reference Eviction*: Objects that have been idle for a certain period are evicted from the pool, or when the JVM needs more memory.
4. *Size-based Eviction*: Objects are evicted from the pool when the number of objects in the pool reaches a certain threshold.

In conclusion, object pooling is a useful design pattern in Java applications, especially those that require high concurrency and frequently create and destroy objects. The eviction process is also important to manage the pool's resources and ensure that it does not become overwhelmed. There are various techniques and strategies for implementing object pooling and eviction in Java, and choosing the right approach depends on the specific needs of the application.

Q: What's the relation of Object pooling in the off-heap memory in the JVM?

A: Object pooling can also be implemented using off-heap memory in the JVM. Off-heap memory is memory that is not managed by the Java heap and is instead allocated and managed by the application itself.

Using object pooling with off-heap memory can be beneficial in situations where the amount of memory required by an application is large and the cost of garbage collection on the Java heap becomes significant. By using off-heap memory, the application can reduce the load on the Java heap and improve performance.

In Java, off-heap memory can be allocated using libraries such as Java Native Access (JNA) or Java Native Interface (JNI). The pool class can allocate a fixed amount of off-heap memory and use it to store pre-allocated objects. When an object is requested from the pool, it is returned from the off-heap memory, and when it is returned to the pool, it is stored back into the off-heap memory for future use.

One advantage of using off-heap memory for object pooling is that it can reduce the frequency of garbage collection, as the objects are not managed by the Java heap. However, it is important to note that off-heap memory is not subject to the same memory management mechanisms as the Java heap, and it is the responsibility of the application to manage the memory effectively to avoid memory leaks and other issues.

In conclusion, object pooling can be implemented using off-heap memory in the JVM to reduce the load on the Java heap and improve performance in memory-intensive applications. However, it requires careful management of memory to avoid potential issues.

Q: What’s the Memory-mapped files ?

A: Memory-mapped files in Java refer to a technique used for directly accessing the contents of a file stored on disk as if it were already in memory. In the context of the Java Virtual Machine (JVM), memory-mapped files are used to improve the performance of file I/O operations. The JVM allows for the creation of memory-mapped files using the *java.nio* package, which provides classes for memory-mapped buffers, channels, and files.

When a memory-mapped file is created, a region of virtual memory is mapped to the file. This region is called a memory-mapped buffer and is backed by the file on disk. Any changes made to the buffer are automatically written to the file, eliminating the need for explicit I/O operations.

Memory-mapped files can be used for a variety of purposes, such as reading and writing large data sets, sharing data between processes, and creating persistent data stores. They are particularly

useful for applications that require high-performance I/O operations, such as database systems and search engines.

It's important to note that while memory-mapped files can provide significant performance benefits, they can also pose security risks if not used properly. Access to memory-mapped files should be carefully controlled to prevent unauthorized access to sensitive data.

Q: What's the JVM warmup?

A: Whenever a new JVM process starts, all required classes are loaded into memory by an instance of the *ClassLoader*. This process takes place in three steps:

1. Bootstrap Class Loading: The "Bootstrap Class Loader" loads Java code and essential Java classes such as *java.lang.Object* into memory. These loaded classes reside in *JRE\lib\rt.jar*.
2. Extension Class Loading: The *ExtClassLoader* is responsible for loading all JAR files located at the *java.ext.dirs* path. In non-Maven or non-Gradle based applications, where a developer adds JARs manually, all those classes are loaded during this phase.
3. Application Class Loading: The *AppClassLoader* loads all classes located in the application class path.

This initialization process is based on a lazy loading scheme. Once class-loading is complete, all-important classes (used at the time of process start) are pushed into the JVM cache (native code) – which makes them accessible faster during runtime. Other classes are loaded on a per-request basis.

The first request made to a Java web application is often substantially slower than the average response time during the lifetime of the process. This warm-up period can usually be attributed to lazy class loading and just-in-time compilation.

Keeping this in mind, for low-latency applications, we need to cache all classes beforehand – so that they're available instantly when accessed at runtime.

This process of tuning the JVM is known as warming up.

Q: What tasks need to be completed before the execution of the application code?

A: When a Java application is started, the JVM needs to perform several initialization tasks before it can execute the application code. Here are some of the main tasks:

1. Loading classes: The JVM needs to load all the classes that are used by the application. This includes the classes in the application itself as well as any classes from external libraries or frameworks.
2. Verifying bytecode: The JVM verifies the bytecode to ensure that it is valid and does not violate any of Java's safety rules. This includes checking for things like type safety, array bounds, and control flow.
3. Allocating memory: The JVM allocates memory for the application and its data structures. This includes things like the Java heap, the method area, and the thread stacks.
4. Initializing data structures: The JVM initializes data structures for the application, such as the object header and the constant pool.
5. JIT compilation: The JVM may also perform Just-In-Time (JIT) compilation of the application code. JIT compilation converts the bytecode to machine code at runtime, which can significantly improve the performance of the application.

These initialization tasks are performed only once, when the JVM starts up. Once they are completed, the JVM is ready to execute the application code.

Q: What's the Java hotspot compiler?

A: The Java HotSpot compiler is a Just-In-Time (JIT) compiler that is included in the Java Virtual Machine (JVM) provided by Oracle. It is designed to improve the performance of Java applications by dynamically compiling bytecode to machine code at runtime.

The HotSpot compiler works by analyzing the runtime behavior of the application to identify frequently executed code paths. It then optimizes these code paths by performing a series of transformations on the bytecode, such as inlining methods, removing redundant instructions, and reordering instructions for better cache utilization.

The HotSpot compiler consists of two main components:

1. Client Compiler: The client compiler is designed to optimize the performance of applications that have a shorter startup time and run for a relatively short period of time. It is focused on quickly generating code that runs efficiently on the client's machine.

2. Server Compiler: The server compiler is designed to optimize the performance of applications that run for a long time and have a high volume of requests. It performs more aggressive optimizations and generates highly optimized machine code that takes longer to generate but runs more efficiently in the long run.

The Java HotSpot compiler is widely used and has been continuously improved over the years to provide better performance for Java applications. It is an important component of the Java platform and is one of the key reasons why Java is known for its high performance and scalability.

Q: What are the pros and cons between the Java Primitive's VS Objects?

A: Java has two types of data types: primitives and objects. Here are some of the pros and cons of using each type:

Pros of primitives

1. Memory efficiency: Primitives are generally more memory-efficient than objects, which can be beneficial in resource-constrained environments.
2. Performance: Operations on primitives are generally faster than operations on objects, as they don't require the overhead of object creation and garbage collection.
3. Simplicity: Using primitives can make code simpler and easier to understand, especially for simple operations.

Cons of primitives

1. Limited functionality: Primitives have limited functionality compared to objects. For example, they don't support methods or inheritance.
2. No null value: Primitives can't have a null value, which can be inconvenient in some cases.
3. No dynamic allocation: Primitives can't be dynamically allocated on the heap like objects can, which can be limiting in certain situations.

Pros of objects

1. Rich functionality: Objects have a rich set of functionalities, including methods, inheritance, and dynamic allocation, which can be beneficial for more complex operations.
2. Nullable: Objects can have a null value, which can be useful in certain cases.
3. Polymorphism: Objects support polymorphism, which allows for more flexible and extensible code.

Cons of objects

1. Overhead: Objects can have a significant overhead in terms of memory usage and performance, which can be a concern in resource-constrained environments.
2. Complexity: Objects can add complexity to code, especially for simple operations.
3. Potential for errors: Objects can be prone to errors like null pointer exceptions, which can be difficult to debug.

In summary, the choice between using primitives or objects depends on the specific needs of the application. Primitives can be beneficial for simple operations or in resource-constrained environments, while objects can be more appropriate for complex operations that require rich functionality.

Q: What’s the "stop-the-world" event in Garbage collection respective to the JVM?

A: The "stop-the-world" event in garbage collection refers to a period when the Java Virtual Machine (JVM) pauses the execution of an application's threads to perform garbage collection. During this period, all application threads are stopped, and the JVM scans the heap to identify and remove unused objects.

The stop-the-world event is necessary for the JVM to ensure the consistency of the heap and prevent memory leaks, but it can have a significant impact on application performance, especially for applications with large heaps or real-time requirements.

To minimize the impact of the stop-the-world event on application performance, the JVM provides several garbage collection algorithms and tuning options that developers can use to optimize their application's memory usage and reduce the frequency and duration of garbage collection pauses.

Q: Why the *volatile* keyword is used in the multi-threaded applications?

A: In a multi-threaded Java application, the volatile keyword is used to ensure that the value of a variable is always read and written from main memory, rather than from a thread's local cache. This is important because in a multi-threaded environment, multiple threads may access the same variable concurrently, and without proper synchronization, the behavior of the program may be unpredictable.

When a variable is declared as volatile, any write operation to the variable is guaranteed to be visible to all other threads that subsequently read the variable. Similarly, any read operation from a volatile variable is guaranteed to return the most up-to-date value of the variable.

The reason for this is that the JVM ensures that when a thread accesses a volatile variable, it synchronizes its local cache with main memory to ensure that it has the most up-to-date value of the variable. This prevents the situation where one thread updates the variable, but another thread continues to read an old value from its local cache.

Q: What’s the CPU's registers and caches, and how this is separate from the main memory (RAM)?

A: In a computer system, the CPU's registers and caches are high-speed, small-capacity memory units that are used to store frequently accessed data and instructions. The CPU's registers are the fastest form of memory available to the CPU and are used to hold the data that the CPU is currently working on, as well as the memory addresses of the data it needs to access next. The

CPU's caches are larger than registers, but still much smaller than the main memory (RAM) and are used to hold recently accessed data and instructions that the CPU is likely to need again soon.

The main memory (RAM), on the other hand, is a larger, slower type of memory that is used to store the data and instructions that are currently not being actively used by the CPU. When the CPU needs to access data or instructions that are not in its registers or caches, it must retrieve them from the main memory, which takes longer than accessing the data from the registers or caches.

The reason the registers and caches are separate from the main memory is because they are much faster than the main memory. This speed difference is due to several factors, including the physical proximity of the registers and caches to the CPU, the use of faster memory technologies in the registers and caches, and the use of sophisticated caching algorithms that allow the CPU to predict which data and instructions it is likely to need next and cache them accordingly.

The separation of the registers and caches from the main memory allows the CPU to access frequently used data and instructions much faster than if it had to retrieve them from the main memory every time. This is why the CPU's caches are used extensively in modern computer systems, and why they are an important factor in the overall performance of a system.

Q: What’s the direct buffers in JVM?

A: In the Java Virtual Machine (JVM), direct buffers are a type of buffer used for reading and writing binary data in a more efficient manner. Direct buffers are implemented outside of the Java heap and are managed by the operating system, which allows for more direct interaction with I/O operations and can lead to improved performance.

The Java NIO (New Input/Output) package provides support for using direct buffers. Direct buffers are created using the *ByteBuffer.allocateDirect()* method, which allocates memory outside of the Java heap.

Direct buffers are useful when dealing with large amounts of binary data, such as when reading or writing files or network connections. Because direct buffers are managed outside of the Java heap, they can help reduce garbage collection overhead and improve performance.

However, it's important to note that using direct buffers can also have some drawbacks. Because they are managed outside of the Java heap, they may not be subject to the same memory constraints as other Java objects, which can lead to out-of-memory errors if not managed carefully. Additionally, because direct buffers are not managed by the JVM, they may not be subject to the same security restrictions as other Java objects, which can present potential security risks.

Q: What’s the shared memory regions in JVM?

A: Shared memory regions in the Java Virtual Machine (JVM) allow for efficient communication and data sharing between multiple processes running on the same machine. In a shared memory region, data can be read and written by multiple processes without the need for expensive inter-process communication mechanisms.

The JVM provides several ways to create shared memory regions. One common method is to use the Java Native Interface (JNI) to interact with native code that creates and manages the shared memory region. The native code can use operating system-specific APIs to allocate and manage the shared memory region.

Another way to create shared memory regions in the JVM is to use the *java.nio.MappedByteBuffer* class. This class allows a portion of a file to be mapped into memory, which can then be shared between multiple processes. Changes made to the memory-mapped region by one process are immediately visible to all other processes that have access to the same region.

Shared memory regions in the JVM can be useful in a variety of scenarios, such as when multiple processes need to share large amounts of data, or when high-performance communication is required between processes. However, it's important to note that shared memory regions can also introduce potential security risks, since multiple processes can access the same memory region. Careful consideration should be given to access control and data validation to ensure the security and integrity of shared memory regions.

Q: What’s the Chronicle Queue, how this works and why do we use it?

A: Chronicle Queue is a high-performance, low-latency, persisted publish-subscribe messaging system used for inter-process communication (IPC) and intra-process communication (IPC) in Java-based applications. It provides a durable, high-throughput, and low-latency messaging solution for large-scale distributed systems.

Chronicle Queue is based on the concept of a persisted log that can be written to and read from at high speeds. It uses a memory-mapped file that can be shared across processes and machines, making it an ideal choice for distributed systems.

The queue is designed to be simple to use and highly performant, with low memory overhead and low latency. It supports both blocking and non-blocking operations and allows for a variety of data types to be stored, including primitive types, byte arrays, and objects.

There are several benefits to using Chronicle Queue, including:

1. High throughput: Chronicle Queue is designed for high-performance messaging and can handle millions of messages per second.

2. Low latency: The queue has been optimized for low latency, making it ideal for use in high-frequency trading, real-time analytics, and other time-critical applications.
3. Persistence: The messages stored in the queue are persisted to disk, ensuring that they are not lost in the event of a system failure.
4. Scalability: The queue can be used in distributed systems and can be shared across multiple processes and machines.

Overall, Chronicle Queue is an excellent choice for applications that require fast, reliable messaging with low latency and high throughput.

Q: Can we use Chronicle queue instead of Apache Kafka?

A: Yes, it is possible to use Chronicle Queue as an alternative to Apache Kafka, but it's important to note that Chronicle Queue and Kafka have different design goals and features, so the suitability of one versus the other will depend on your specific use case and requirements.

Here are some differences between Chronicle Queue and Kafka:

1. Message ordering: Kafka provides strict ordering of messages within a partition, while Chronicle Queue does not guarantee strict ordering of messages.
2. Data retention: Kafka is designed to retain data for a longer period, typically days or weeks, while Chronicle Queue is designed for shorter retention periods, typically hours or minutes.
3. Distributed processing: Kafka has built-in support for distributed processing, allowing for parallel processing of messages across multiple nodes, while Chronicle Queue does not have this capability built-in.
4. Ecosystem: Kafka has a large ecosystem of third-party tools and integrations, while Chronicle Queue has a smaller ecosystem.
5. Throughput: Kafka is known for its high throughput and scalability, while Chronicle Queue is optimized for low-latency performance.

In general, if your use case requires strict message ordering, long-term data retention, and distributed processing across multiple nodes, Kafka may be a better choice. If you require low latency and high throughput, with shorter data retention periods and a simpler architecture, Chronicle Queue may be a better fit.

Ultimately, the best choice will depend on the specific requirements of your application and your organization's resources and expertise.

Q: What’s JNI?

A: The Java Native Interface (JNI) is a programming framework that allows Java code running in the Java Virtual Machine (JVM) to call and be called by native applications and libraries written in other programming languages, such as C, C++, and assembly. JNI provides a way to integrate Java code with non-Java code, and to access system-specific functionality that is not available through standard Java APIs.

With JNI, Java code can call functions in a native library, and native code can call Java methods and access Java objects. JNI also provides a mechanism for passing data between Java and native code, such as arrays and strings.

JNI is typically used when Java code needs to interact with operating system-specific functionality, or when performance is a critical concern. By calling native code directly, developers can access system-specific functionality that may not be available through Java's standard library and can often achieve better performance than with pure Java code.

However, JNI can also introduce potential security risks since native code can access system resources and execute potentially malicious code. Careful consideration should be given to access control and data validation to ensure the security and integrity of JNI-based applications.

Q: What’s the JNA?

A: JNA (Java Native Access) is a Java library that allows Java programs to access and use native libraries and functions in a platform-independent way. It provides a simple and easy-to-use API for accessing native libraries from Java code, without the need for writing native code or using Java Native Interface (JNI).

JNA works by using Java's built-in Java Virtual Machine (JVM) to invoke native functions in a platform-independent manner. This means that developers can write code that works across different operating systems without needing to recompile or modify the code.

To use JNA in a Java application, you first need to define a Java interface that corresponds to the native library or function you want to use. Then, you can use JNA's *NativeLibrary* class to load the native library and use JNA's Function class to map the Java interface to the corresponding native function.

Here is an example of how to use JNA to access the *printf* function from the C standard library:

```java
import com.sun.jna.Library;
import com.sun.jna.Native;


public class App {

    public interface CLibrary extends Library {
        void printf(String format, Object... args);
    }

    public static void main(String[] args) {
        CLibrary cLib = (CLibrary) Native.loadLibrary( name: "c", CLibrary.class);
        cLib.printf("Hello, %s\n", "JNA");
    }

}
```

In this example, we define a Java interface *CLibrary* that extends JNA's Library class and defines a single method *printf* that maps to the *printf* function in the C standard library. We then load the c library using JNA's Native.*loadLibrary* method and call the *printf* method on the *CLibrary* instance to print a message to the console.

JNA is a powerful and flexible library that can be used to access a wide range of native libraries and functions from Java code. It can be particularly useful in situations where you need to access system-level functionality that is not available through the standard Java API.

Q: What' the JIT?

A: The Just-In-Time (JIT) compiler is a feature of the Java Virtual Machine (JVM) that dynamically compiles Java bytecode into native machine code at runtime. This allows Java applications to achieve high performance and execute more quickly than they would with interpreted bytecode alone.

When a Java program is executed, its bytecode is interpreted by the JVM's interpreter, which executes each instruction one at a time. However, this can be slow, especially for long-running programs that execute the same code repeatedly.

The JIT compiler addresses this performance issue by compiling frequently executed bytecode into native machine code, which can be executed much more quickly than interpreted bytecode. The JIT compiler identifies hot spots in the code, or frequently executed code paths, and compiles these hot spots on-the-fly, while the program is running. The compiled code is then cached for reuse, so that subsequent invocations of the same code can execute even more quickly.

The JIT compiler is an important feature of the JVM, as it enables Java applications to achieve high performance, while still retaining the benefits of Java's portability and platform independence. The JIT compiler can also optimize code for the specific hardware and operating system on which the application is running, further improving performance.

Overall, the JIT compiler is an important component of Java's runtime environment, and a key factor in Java's success as a high-performance programming language.

Q: What's the native functions and how use them in the Java applications?

A: Native functions are functions or methods that are implemented in a native programming language, such as C or C++, and are compiled into native code that can be executed directly by the processor of the host system. These functions are typically used to access low-level system resources or perform system-level operations that are not available through the standard Java API.

When a Java program needs to access a native function, it must use a mechanism such as Java Native Interface (JNI) or JNA to interface with the native code. These mechanisms allow the Java program to load and call the native function from within the Java environment, without the need for the programmer to write any native code.

For example, a native function might be used to access hardware devices, such as a printer or a scanner, or to perform low-level operations such as memory management or system-level I/O. Native functions are typically used in performance-critical applications or in situations where direct access to system resources is required.

JNA provides a simple and easy-to-use interface for accessing native functions from within a Java program, allowing developers to take advantage of the performance and functionality of native code without needing to write any native code themselves.

Q: What's the long living hot cache in JVM?

A: The long living hot cache in the Java Virtual Machine (JVM) is a type of cache that stores frequently accessed objects and data structures in memory to improve application performance. Unlike short-lived caches, which are often used for temporary storage of frequently accessed data, the long living hot cache is designed to persist across multiple invocations of the JVM.

The long living hot cache is typically used to store objects and data structures that are expensive to create or compute, but frequently accessed by the application. By keeping these objects in memory, the application can avoid the overhead of recreating or recomputing them on each use.

One common example of a long living hot cache in the JVM is the class metadata cache. When a Java class is loaded by the JVM, its metadata is stored in memory to avoid the overhead of repeated loading and parsing. The metadata cache is a long living hot cache because it persists across multiple invocations of the JVM and is frequently accessed by the application during class loading and method dispatch.

The long living hot cache is managed by the JVM's garbage collector, which periodically scans the cache for objects that are no longer referenced by the application and frees up memory accordingly. Developers can also configure the cache size and eviction policy to optimize performance for specific application workloads.

Overall, the long living hot cache in the JVM is an important mechanism for improving application performance by reducing the overhead of recreating or recomputing frequently accessed objects and data structures.

Q: What's the JDK and JRE and difference between them?

A: JDK and JRE are two different software distributions provided by Oracle for running Java applications. Here are the definitions and the differences between them:

- JDK (Java Development Kit): The JDK is a software development kit that contains everything required to develop and compile Java applications. It includes the Java runtime environment (JRE), the Java compiler, the Java Virtual Machine (JVM), and a variety of tools and libraries for Java development. The JDK is intended for developers who want to write Java code, compile it, and run it on their own machines.

- JRE (Java Runtime Environment): The JRE is a runtime environment that contains only the Java Virtual Machine (JVM) and the Java class libraries required to run Java applications. It does not include the Java compiler or any development tools. The JRE is intended for end-users who want to run Java applications on their machines without doing any Java development themselves.

The main difference between the JDK and the JRE is that the JDK is a complete development kit, while the JRE is a runtime environment only. The JDK includes everything required to write, compile, and run Java applications, while the JRE includes only the components required to run Java applications.

In summary, if you are a Java developer and want to write, compile, and run Java code, you will need to download and install the JDK. If you are an end-user who just wants to run Java applications on your machine, you can download and install the JRE.

Q: What do you mean by memory management in Java?

A: Memory is the key resource an application requires to run effectively and like any resource, it is scarce. As such, its allocation and deallocation to and from applications or different parts of an application require a lot of care and consideration.

However, in Java, a developer does not need to explicitly allocate and deallocate memory – the JVM and more specially the Garbage Collector – has the duty of handling memory allocation so that the developer doesn't have to.

This is contrary to what happens in languages like C where a programmer has direct access to memory and literally references memory cells in his code, creating a lot of room for memory leaks.

Q: What is Garbage Collection and what are its advantages?

A: Garbage collection is the process of looking at heap memory, identifying which objects are in use and which are not, and deleting the unused objects.

An in-use object, or a referenced object, means that some part of your program still maintains a pointer to that object. An unused object, or unreferenced object, is no longer referenced by any part of your program. So, the memory used by an unreferenced object can be reclaimed.

The biggest advantage of garbage collection is that it removes the burden of manual memory allocation/deallocation from us so that we can focus on solving the problem at hand.

Q: Are there any disadvantages of Garbage Collection?

A: Yes. Whenever the garbage collector runs, it influences the application's performance. This is because all other threads in the application must be stopped to allow the garbage collector thread to effectively do its work.

Depending on the requirements of the application, this can be a real problem that is unacceptable by the client. However, this problem can be greatly reduced or even eliminated through skillful optimization and garbage collector tuning and using different GC algorithms.

Q: What are stack and heap? What is stored in each of these memory structures, and how are they interrelated?

A: The stack is a part of memory that contains information about nested method calls down to the current position in the program. It also contains all local variables and references to objects on the heap deepened in currently executing methods.

This structure allows the runtime to return from the method knowing the address whence it was called and clear all local variables after exiting the method. Every thread has its own stack. The heap is a large bulk of memory intended for allocation of objects. When you create an object with

the new keyword, it gets allocated on the heap. However, the reference to this object lives on the stack.

Q: What is generational garbage collection and what makes it a popular garbage collection approach?

A: Generational garbage collection can be loosely depended as the strategy used by the garbage collector where the heap is divided into several sections called generations, each of which will hold objects according to their "age" on the heap.

Whenever the garbage collector is running, the first step in the process is called marking. This is where the garbage collector identifies which pieces of memory are in use and which are not. This can be a very time-consuming process if all objects in a system must be scanned.

As more and more objects are allocated, the list of objects grows and grows leading to longer and longer garbage collection time. However, empirical analysis of applications has shown that most objects are short-lived.

With generational garbage collection, objects are grouped according to their "age" in terms of how many garbage collection cycles they have survived. This way, the bulk of the work spread across various minor and major collection cycles. Today, almost all garbage collectors are generational. This strategy is so popular because, over time, it has proven to be the optimal solution.

Q: Describe in detail how generational garbage collection works

A: To properly understand how generational garbage collection works, it is important to first remember how Java heap is structured to facilitate generational garbage collection.

The heap is divided up into smaller spaces or generations. These spaces are Young Generation, Old or Tenured Generation, and Permanent Generation.

The young generation hosts most of the newly created objects. An empirical study of most applications shows that majority of objects are quickly short lived and therefore, soon become eligible for collection. Therefore, new objects start their journey here and are only "promoted" to the old generation space after they have attained a certain "age". The term "age" in generational garbage collection refers to the number of collection cycles the object has survived.

The young generation space is further divided into three spaces: an Eden space and two survivor spaces such as Survivor 1 (s1) and Survivor 2 (s2).

The old generation hosts objects that have lived in memory longer than a certain "age". The objects that survived garbage collection from the young generation are promoted to this space. It is generally larger than the young generation. As it is bigger in size, the garbage collection is more expensive and occurs less frequently than in the young generation.

The permanent generation or more commonly called, *PermGen*, contains metadata required by the JVM to describe the classes and methods used in the application. It also contains the string pool for storing interned strings. It is populated by the JVM at runtime based on classes in use by the application. In addition, platform library classes and methods may be stored here.

First, any new objects are allocated to the Eden space. Both survivor spaces start out empty. When the Eden space fills up, a minor garbage collection is triggered. Referenced objects are moved to the first survivor space. Unreferenced objects are deleted.

During the next minor GC, the same thing happens to the Eden space. Unreferenced objects are deleted and referenced objects are moved to a survivor space. However, in this case, they are moved to the second survivor space (S1).

In addition, objects from the last minor GC in the first survivor space (S0) have their age incremented and are moved to S1. Once all surviving objects have been moved to S1, both S0 and Eden space are cleared. At this point, S1 contains objects with different ages.

At the next minor GC, the same process is repeated. However, this time the survivor spaces switch. Referenced objects are moved to S0 from both Eden and S1. Surviving objects are aged. Eden and S1 are cleared.

After every minor garbage collection cycle, the age of each object is checked. Those that have reached a certain arbitrary age, for example, 8, are promoted from the young generation to the old or tenured generation. For all subsequent minor GC cycles, objects will continue to be promoted to the old generation space.

This pretty much exhausts the process of garbage collection in the young generation. Eventually, a major garbage collection will be performed on the old generation which cleans up and compacts that space.

Q: When does an object become eligible for garbage collection? Describe how the GC collects an eligible object?

A: An object becomes eligible for Garbage collection or GC if it is not reachable from any live threads or by any static references.

The most straightforward case of an object becoming eligible for garbage collection is if all its references are null. Cyclic dependencies without any live external reference are also eligible for GC. So, if object A references object B and object B references Object A and they don't have any other live reference then both Objects A and B will be eligible for Garbage collection.

Another obvious case is when a parent object is set to null. When a kitchen object internally references a fridge object and a sink object, and the kitchen object is set to null, both fridge and sink will become eligible for garbage collection alongside their parent, kitchen.

Q: How do you trigger garbage collection from Java code?

A: You, as Java programmer, cannot force garbage collection in Java; it will only trigger if JVM thinks it needs a garbage collection based on Java heap size.

Before removing an object from memory garbage collection thread invokes finalize() method of that object and gives an opportunity to perform any sort of cleanup required. You can also invoke this method of an object code, however, there is no guarantee that garbage collection will occur when you call this method.

Additionally, there are methods like *System.gc()* and *Runtime.gc()* which is used to send request of Garbage collection to JVM, but it's not guaranteed that garbage collection will happen.

What happens when there is not enough heap space to accommodate storage of new objects?

If there is no memory space for creating a new object in Heap, Java Virtual Machine throws *OutOfMemoryError* or more specially *java.lang.OutOfMemoryError* heap space.

Q: Is it possible to resurrect an object that became eligible for garbage collection?

A: When an object becomes eligible for garbage collection, the GC must run the finalize method on it. The finalize method is guaranteed to run only once, thus the GC tags the object as finalized and gives it a rest until the next cycle.

In the finalize method you can technically "resurrect" an object, for example, by assigning it to a static eld. The object would become alive again and non- eligible for garbage collection, so the GC would not collect it during the next cycle.

The object, however, would be marked as finalized, so when it would become eligible again, the finalize method would not be called. In essence, you can turn this "resurrection" trick only once for the lifetime of the object. Beware that this ugly hack should be used only if you really know what you're doing — however, understanding this trick gives some insight into how the GC works.

Q: Describe strong, weak, soft, and phantom references and their role in garbage collection.

A: In Java, garbage collection is the process of automatically freeing memory occupied by objects that are no longer in use by the application. Java provides several types of references that can be used to control the garbage collection process. Here are the descriptions and roles of the four types of references:

1. Strong references: Strong references are the default type of reference used in Java. An object with a strong reference can't be garbage collected until the reference is no longer reachable from the application code. In other words, as long as there is a strong reference to an object, it will not be garbage collected.
2. Weak references: A weak reference is a reference that does not prevent the object it references from being garbage collected. Weak references are useful for implementing caches or other data structures that should be automatically cleared when the memory is needed. If the object has only weak references remaining, it can be garbage collected.
3. Soft references: A soft reference is a reference that behaves like a weak reference, but is only cleared by the garbage collector when memory is low. Soft references are useful for implementing caches or other data structures that should be cleared only when the memory is needed. If the object has only soft references remaining and the memory is low, it can be garbage collected.
4. Phantom references: A phantom reference is a reference that has no effect on the object's life cycle. It is useful for tracking when an object is finalized. Phantom references are enqueued after the object is finalized, but before it is garbage collected. The garbage collector does not reclaim the memory associated with the object until all phantom references have been dequeued.

In summary, strong references are the default type of reference used in Java, and objects with strong references will not be garbage collected until the reference is no longer reachable. Weak and soft references allow objects to be garbage collected when they are no longer needed, and phantom references are useful for tracking when an object is finalized. These different types of references provide greater control over the garbage collection process and can be used to optimize memory usage in Java applications.

Q: Suppose we have a circular reference (two objects that reference each other). Could such pair of objects become eligible for garbage collection and why?

A: Yes, a pair of objects with a circular reference can become eligible for garbage collection. This is because of how Java's garbage collector handles circular references. It considers objects live not when they have any reference to them, but when they are reachable by navigating the object graph starting from some garbage collection root (a local variable of a live thread or a static eld). If a pair of objects with a circular reference is not reachable from any root, it is considered eligible for garbage collection.

Q: How are strings represented in memory?

A: A String instance in Java is an object with two fields: a char[] value field and an int hash field. The value eld is an array of chars representing the string itself, and the hash field contains the *hashCode* of a string which is initialized with zero, calculated during the first *hashCode()* call and cached ever since. As a curious edge case, if a *hashCode* of a string has a zero value, it must be recalculated each time the *hashCode()* is called.

Important thing is that a String instance is immutable: you can't get or modify the underlying *char[]* array. Another feature of strings is that the static constant strings are loaded and cached in a string pool. If you have multiple identical String objects in your source code, they are all represented by a single instance at runtime.

Q: What is a *StringBuilder* and what are its use cases? What is the difference between appending a string to a StringBuilder and concatenating two strings with + operator? How does StringBuilder different from StringBuilder?

A: *StringBuilder* allows manipulating character sequences by appending, deleting, and inserting characters and strings. This is a mutable data structure, as opposed to the *String* class which is immutable. When concatenating two String instances, a new object is created, and strings are copied. This could bring a huge garbage collector overhead if we need to create or modify a string in a loop. *StringBuilder* allows handling string manipulations much more efficiently.

StringBuffer is different from *StringBuilder* in that it is thread safe. If you need to manipulate a string in a single thread, use StringBuilder instead.

StringBuffer is thread-safe, meaning that they have synchronized methods to control access so that only one thread can access a *StringBuffer* object's synchronized code at a time. Thus, *StringBuffer* objects are generally safe to use in a multi-threaded environment where multiple threads may be trying to access the same *StringBuffer* object at the same time.

StringBuffer access is not synchronized so that it is not thread-safe. By not being synchronized, the performance of *StringBuilder* can be better than *StringBuffer*. Thus, if you are working in a single-threaded environment, using StringBuilder instead of StringBuffer may result in increased performance. This is also true of other situations such as a StringBuilder local variable (i.e., a variable within a method) where only one thread will be accessing a StringBuilder object.

So, prefer *StringBuffer* because,

1. Small performance gain.
2. StringBuilder is a 1:1 drop-in replacement for the StringBuffer class.
3. StringBuilder is not thread synchronized and therefore performs better on most implementations of Java.

Q: How does the static allocation occur in Java?

A: The specific memory application is described below. Remember that this is Sun specific.

1. Classes (loaded by the class loaders) go in a special area on heap : Permanent Generation
2. All the information related to a class like name of the class, Object arrays associated with the class, internal objects used by JVM (like java/lang/Object), and optimization information goes into the Permanent Generation area.

3. All the static member variables are kept on the Permanent Generation area again.
4. Objects go on a different heap : *Young generation*.
5. There is only one copy of each method per class, be the method static or non-static. That copy is put in the Permanent Generation area. For non-static methods, all the parameters and local variables go onto the stack - and whenever there is a concrete invocation of that method, we get a new stack-frame associated with it.
6. Local variables of a static method in Java are stored on the stack, which is a region of memory used by the JVM to store method frames. When a static method is called, a new method frame is created on the stack to hold the method's local variables and any intermediate values used during the method's execution. The size of the method frame is determined by the number and size of the local variables used in the method, and it is allocated from the JVM's stack memory. The method frame is destroyed when the method returns or when an exception is thrown, and the stack memory used by the frame is released for reuse by the JVM. It's worth noting that static methods do not have access to instance variables, as they are associated with an instance of a class, whereas static methods are associated with the class itself.
7. If the objects (in the young generation) need to use a static member (in the permanent generation), they are given a reference to the static member and they are given enough memory space to store the return type of the method, etc.

   If some method accesses a static member variable, what it gets is either a primitive value or an object reference. This may be assigned to an (existing) local variable or parameter, assigned to an (existing) static or non-static member, assigned to an (existing) element of a previously allocated array, or simply used and discarded.

   In no case does *new* storage need to be allocated to hold either a reference or a primitive value. Typically, one word of memory is all that is needed to store an object or array reference, and a primitive value typically occupies one or two words, depending on the hardware architecture. In no case does space need to be allocated by the caller to hold some object / array returned by a method. In Java, objects and arrays are always returned using pass-by-value semantics, but that value that is returned is an object or array reference.

Q: What is the difference between PermGen and Metaspace in the JVM?

A: PermGen and Metaspace are both regions of memory used by the Java Virtual Machine (JVM) to store metadata about classes and their respective class loaders. However, there are some significant differences between them.

PermGen (Permanent Generation) is a region of memory used by older versions of the JVM to store metadata related to classes, such as the class definition itself, its name and methods, and static variables. PermGen is part of the JVM's heap space and has a fixed size that is configured using the *-XX:MaxPermSize* option. One issue with PermGen was that it could cause *OutOfMemoryError* when it became full, which could happen if there were many dynamically

loaded classes or if there was a memory leak in the application. Starting from Java 8, PermGen was replaced by Metaspace.

Metaspace is a region of native memory used by newer versions of the JVM to store metadata about classes, including class definitions, method definitions, and constant pool information. Metaspace is not part of the JVM's heap space and has no fixed size limit; instead, it automatically grows or shrinks as needed, up to the maximum amount of native memory available to the JVM. This allows for more efficient memory usage and eliminates the risk of PermGen-related *OutOfMemoryError*.

In summary, PermGen and Metaspace are both regions of memory used to store metadata about classes and their respective class loaders, but Metaspace is a newer and more flexible replacement for PermGen that offers better memory management and eliminates some of the limitations of PermGen.

Q: Why do we build low latency system single threaded?

A: Low latency Java applications need to be single-threaded because multi-threading can introduce overhead and increase latency due to synchronization, context-switching, and contention for shared resources.

In a multi-threaded Java application, multiple threads may be accessing shared resources such as memory, locks, or I/O operations. When multiple threads access these resources simultaneously, contention can occur, which can lead to delays and increased latency.

Furthermore, context-switching between threads can also introduce overhead, as the operating system needs to save the state of the current thread and restore the state of the next thread. This can add significant overhead, particularly in low-latency applications where response times need to be very fast.

In a single-threaded Java application, on the other hand, there is no contention between threads and no context-switching overhead. This can lead to faster response times and lower latency, particularly in applications where low-latency is critical, such as high-frequency trading or real-time data processing.

However, it is important to note that not all low-latency applications need to be single-threaded. Some applications may benefit from multi-threading if they can be designed to minimize contention and avoid context-switching overhead. In these cases, careful design and implementation of multi-threaded code can still achieve low latency and high performance.