**Questions:**

1. The name of java file should be same as of class name. If there are multiple classes and more than one classes are public, then what will be the name of java file? What will be the scenario?

2. Java or C++ approach? Which one is better while we are on one same machine?

- Java: time and gap problem...solve-by capturing performance limiter by JIT

3. JIT(Just In Time) Compiler?

4. JRE(java Runtime Environment)?

**Answers:**

**Ans No 1.**

We can have any number of classes in one Java program but it can contain at most zero or one public class. If no public class is available in a file then we can give any name to the Java file. If a public class is available, in that case, the name of the Java file should be the same as the class name.

There are no restrictions on the number of classes that can be present in one Java program. But each Java program should have only one class declared with public access specifier. There cannot be two public classes in a single Java program.

While defining multiple classes in a single Java file you need to make sure that only one class among them is public. If you have more than one public classes a single file a compile-time error will be generated.

2. Use JIT compiler ….

**Ans 3.**

In computing, **just-in-time compilation** is compilation during execution of a program rather than before execution. This may consist of source code translation but is more commonly bytecode translation to machine code, which is then executed directly.

Java programs consists of classes, which contain platform-neutral bytecodes that can be interpreted by a JVM on many different computer architectures. At run time, the JVM loads the class files, determines the semantics of each individual bytecode, and performs the appropriate computation. The additional processor and memory usage during interpretation means that a Java application performs more slowly than a native application. The JIT compiler helps improve the performance of Java programs by compiling bytecodes into native machine code at run time.

The **JIT compiler is enabled by default**. When a method has been compiled, the JVM calls the compiled code of that method directly instead of interpreting it. Theoretically, if compilation did not require processor time and memory usage, compiling every method could allow the speed of the Java program to match that of a native application.

**JIT compilation does require processor time and memory usage.** When the JVM first starts up, thousands of methods are called. Compiling all of these methods can significantly affect startup time, even if the program eventually achieves very good peak performance.

In practice, methods are not compiled the first time they are called. For each method, the JVM maintains an invocation count, which starts at a predefined compilation threshold value and is decremented every time the method is called. When the invocation count reaches zero, a just-in-time compilation for the method is triggered. Therefore, often-used methods are compiled soon after the JVM has started, and less-used methods are compiled much later, or not at all. The JIT compilation threshold helps the JVM start quickly and still have improved performance. The threshold value was selected to obtain an optimal balance between startup times and long-term performance.

When a method is chosen for compilation, the JVM feeds its bytecodes to the Just-In-Time compiler (JIT). The JIT needs to understand the semantics and syntax of the bytecodes before it can compile the method correctly.

**JIT stands for "Just-In-Time," and a JIT compiler is a type of compiler used in the execution of computer programs during runtime.** Unlike traditional compilers that translate the entire source code into machine code before execution (ahead-of-time compilation), JIT compilers translate code into machine code as the program is running. Here's a brief overview of how a **JIT compiler works:**

1. **Interpretation and Compilation**: Initially, a program might be interpreted by an interpreter, which reads and executes the source code line by line. While interpreting, the JIT compiler identifies portions of the code that are frequently executed (hotspots).
2. **Dynamic Compilation:** When a hotspot is identified, the JIT compiler translates the corresponding source code into machine code, optimizing it for the specific architecture of the computer running the program.
3. **Caching:** The generated machine code is often cached, so if the same portion of code is encountered again, the cached machine code can be used instead of re-interpreting and recompiling.
4. **Execution:** The machine code generated by the JIT compiler is executed directly by the CPU, providing performance improvements compared to interpretation.

JIT compilation offers several **advantages:**

1. **Optimized Execution:** Since the compilation is done at runtime and based on actual usage patterns, the JIT compiler can make optimizations that are not possible during ahead-of-time compilation.
2. **Adaptability:** The compiled code can be optimized for the specific hardware and runtime conditions, leading to improved performance.
3. **Reduced Startup Time:** Interpretation allows for quick program startup, and JIT compilation helps improve performance as the program continues to run.

Java is a notable example of a language that uses a JIT compiler. Java code is initially compiled into an intermediate form called bytecode, and then the Java Virtual Machine (JVM) uses a JIT compiler to translate the bytecode into machine code for the specific platform on which the program is running. This approach allows Java programs to be platform-independent and provides a good balance between portability and performance.

**Ans 4.**

**The Java Runtime Environment (JRE)** is software that Java programs require to run correctly. Java is a computer language that powers many current web and mobile applications. The JRE is the underlying technology that communicates between the Java program and the operating system.

The Java Runtime Environment (JRE) is **a set of software tools and libraries** that allows Java applications to run on a computer. It provides the necessary runtime support for executing Java programs. The JRE includes the Java Virtual Machine (JVM), libraries, and other components needed for running Java applications.

Here are the **key components of the Java Runtime Environment:**

1. **Java Virtual Machine (JVM):** The JVM is a crucial part of the JRE. It interprets and executes Java bytecode, which is an intermediate form of the Java code generated during the compilation process. The JVM is platform-dependent, and there are different implementations for various operating systems.
2. **Java Class Library:** The JRE includes a set of standard class libraries and APIs (Application Programming Interfaces) that provide essential functionality to Java applications. These libraries cover a wide range of tasks, including input/output operations, networking, data structures, graphical user interfaces (GUIs), and more. Developers can leverage these libraries to build robust and cross-platform applications.
3. **Java Runtime Environment Core Components:** Besides the JVM and class libraries, the JRE also includes other essential components, such as native libraries, executable binaries, and configuration files needed to support Java applications during runtime.

It's important to note that the **JRE is distinct from the Java Development Kit (JDK).** The JDK includes everything in the JRE but also includes additional tools and utilities necessary for Java development, such as compilers, debuggers, and development-related libraries.

When a user wants to run a Java application on their machine, they typically need to have the JRE installed. The JRE allows them to execute Java programs without having to install the full development environment provided by the JDK. In contrast, developers working on Java applications will typically use the JDK for compiling, debugging, and testing their code.

As of my knowledge cutoff in January 2022, the modular structure of Java starting from Java 9 includes the concept of the Java Platform Module System (JPMS), which divides the JRE into modules for better organization and modularity. The general principles, however, remain the same.

The Java Runtime Environment, or JRE, is a software layer that runs on top of a computer’s operating system software and provides the class libraries and other resources that a specific Java program needs to run.

The **JRE is one of three interrelated components** for developing and running Java programs. The other two components are as follows:

* The Java Development Kit, or JDK, is a set of tools for developing Java applications. Developers choose JDKs by Java version and by package or edition—Java Enterprise Edition (Java EE), Java Special Edition (Java SE), or Java Mobile Edition (Java ME). Every JDK always includes a compatible JRE, because running a Java program is part of the process of developing a Java program.
* The Java Virtual Machine, or JVM, executes live Java applications. Every JRE includes a default JRE, but developers are free to choose another that meets the specific resource needs of their applications.

The JRE combines Java code created using the JDK with the necessary libraries required to run it on a JVM and then creates an instance of the JVM that executes the resulting program. JVMs are available for multiple operating systems, and programs created with the JRE will run on all of them. In this way, the Java Runtime Environment is what enables a Java program to run in any operating system without modification.