1. Fundamental reason for evolution of computer language:

* To adapt to changing computing environment.
* To implement new ideas of computing and programming.

1. **Evolution of CL**: Invention of computer -> Binary instructions -> Assembly language -> High level language (FORTRAN, COBOL) -> Structured language (C) -> Object oriented language (C++, Java)
2. Fundamental reason for evolution of java:
   1. Need of portable code due to introduction of WWW and consumer electronics.
   2. Need for of PL with in build support multithreading and internet related libraries.
3. **Java features**:
   1. **Simple**: Easy to learn.
   2. **Object oriented**: Java is OO while primitives are kept non-objects for high performance
   3. **Robust**: Inbuild memory management and elegant exception handling capabilities.
   4. **Multithreaded**: Easy to create program that does multiple operations simultaneously.
   5. **Architecture neutral**: Support write once run anywhere approach.
   6. **Distributed**: Support Remote Method Invocation (RMI)
   7. **Dynamic**: Java carry lot of information at runtime that allows small fragment of code to be updated dynamically at runtime. This lead to robustness.
4. **Evolution of Java**:
   1. Java 1.0: Base release.
   2. Java 1.1: New libraries, Event handling
   3. Java 1.2: Swing, Collection, Deprecated thread api’s
   4. Java 1.3:
   5. Java 1.4: NIO, Chained exceptions, assert
   6. Java 5: Generics, Annotations, Autoboxing, Enum, for-each, varags, static import, Concurrent utilities
   7. Java 6:
   8. Java 7: try-with-resources, Type inference in generics, multi-catch, @SafeVarags, NIO.2
   9. Java 8: Lambda expressions, method references, Stream api, interface methods
   10. Java 9: JShell, Module system(removed rt.jar), Security enhancements(SHA-3), Concurrency updates(), GC updates(G1 GC), Interface changes(Private method)
   11. Java 10: better docker support
5. **Syntax**: It’s about the structure or the grammar of the language. It answers the question: how do I construct a valid sentence?

* some semantics cannot be determined at compile-time and must therefore must be evaluated at run-time. In the ++ operator example, if x is already at the maximum value for its data type.

1. **Portability type**:
   1. **Source code portability**: A given Java program should produce identical results regardless of the underlying CPU, operating system, or Java compiler.
      * C and C++ also provide numerous opportunities to create non-portable code as well. Unless programs written in C and C++ are designed to be portable from the beginning, the ability to move to different machines is more theoretical than practical. C and C++ leave undefined details such as the size and endianism of atomic data types, the behavior of floating-point math, the value of uninitialized variables, and the behavior when freed memory is accessed.
      * although the syntax of C and C++ is well defined, the semantics are not. This semantic looseness allows a single block of C or C++ source code to compile to programs that give different results when run on different CPUs, operating systems, compilers, and even on a single compiler/CPU/OS combination, depending on various compiler settings.
      * the features that make Java so portable have a downside. Java assumes a 32-bit machine with 8-bit bytes and IEEE754 floating-point math. Machines that don't fit this model, including 8-bit microcontrollers and Cray supercomputers, can't run Java efficiently.
   2. **CPU portability**:  Java compilers are different. Instead of producing output for each different CPU family on which the Java program is intended to run, the current Java compilers produce object code-Byte code for a CPU that does not yet exist.
      * Producing output for an imaginary CPU is not new with Java: (Pascal, Limbo etc. ). The Internet-savvy JVM distinguishes itself from these other virtual CPU implementations by intentionally being designed to allow the generation of protably safe, virus-free code. Prior to the Internet, there was no need for virtual machines to prove programs safe and virus-free. This safety feature, combined with a much better understanding of how to quickly execute programs for imaginary CPUs, has led to rapid, widespread acceptance of the JVM.
   3. **OS portability**: Java solves this problem by providing a set of library functions (contained in Java-supplied libraries such as awt, util, and lang) that talk to an imaginary OS and imaginary GUI. Just like the JVM presents a virtual CPU, the Java libraries present a virtual OS/GUI. Every Java implementation provides libraries implementing this virtual OS/GUI.
      * Java has provided a least-common-denominator functionality in its OS/GUI libraries. Features available on only one OS/GUI, such as tabbed dialog boxes, were omitted.
      * Another approach, not followed by Java, is to pick a single OS/GUI as the master and provide wrapper libraries supporting this master OS/GUI on all machines to which you wish to port. The problem with the master OS/GUI approach is that the ported applications often look alien on the other machines.
2. **Class data sharing**: The class sharing feature offers the transparent and dynamic sharing of data between multiple Java™ Virtual Machines (JVMs). When enabled, JVMs use shared memory to obtain and store data, including information about: loaded classes, Ahead-Of-Time (AOT) compiled code, commonly used UTF-8 strings, and Java Archive (JAR) file indexes.
   1. You can enable shared classes with the **-Xshareclasses** command-line option.
   2. When enabled, shared classes share the immutable parts of a class between JVMs, which has the following benefits:
      * The amount of physical memory used can be significantly less when using more than one JVM instance.
      * Loading classes from a populated cache is faster than loading classes from disk, because classes are already partially verified and are possibly already loaded in memory. Therefore, class sharing also benefits applications that regularly start new JVM instances doing similar tasks.
3. **JIT compiler**: The Just-In-Time (JIT) compiler is a component of the Java™ Runtime Environment. It improves the performance of Java applications by compiling bytecodes to native machine code at run time.
   1. JIT compilation does require processor time and memory usage. When the JVM first starts up, thousands of methods are called. Compiling methods can significantly affect startup time, even if the program eventually achieves very good peak performance.
   2. For each method, the JVM maintains a call count, which is incremented every time the method is called. The JVM interprets a method until its call count exceeds a JIT compilation threshold.
      * JIT compilation does require processor time and memory usage. When the JVM first starts up, thousands of methods are called. Compiling methods can significantly affect startup time, even if the program eventually achieves very good peak performance. The JIT compilation threshold helps the JVM start quickly and still have improved performance.
   3. After a method is compiled, its call count is reset to zero and subsequent calls to the method continue to increment its count. When the call count of a method reaches a JIT recompilation threshold, the JIT compiler compiles it a second time, applying a larger selection of optimizations than on the previous compilation. This process is repeated until the maximum optimization level is reached.
   4. bytecodes are first reformulated in an internal representation called *trees*, which resembles machine code more closely than bytecodes. Analysis and optimizations are then performed on the trees of the method. The JIT compiler can use more than one compilation thread to perform JIT compilation tasks. Using multiple threads can potentially help Java™ applications to start faster. In practice, multiple JIT compilation threads show performance improvements only where there are unused processing cores in the system.
      * you can override the JVM decision by using the **-XcompilationThreads.**
   5. The compilation consists of the following phases:
      * [inlining](https://www.ibm.com/support/knowledgecenter/SSYKE2_8.0.0/com.ibm.java.zos.80.doc/diag/understanding/jit_optimize_inline.html?view=kc): Inlining is the process by which the trees of smaller methods are merged, or "inlined", into the trees of their callers. This speeds up frequently executed method calls.
      * [local optimizations](https://www.ibm.com/support/knowledgecenter/SSYKE2_8.0.0/com.ibm.java.zos.80.doc/diag/understanding/jit_optimize_local.html?view=kc): Local optimizations analyze and improve a small section of the code at a time.
      * [control flow optimizations](https://www.ibm.com/support/knowledgecenter/SSYKE2_8.0.0/com.ibm.java.zos.80.doc/diag/understanding/jit_optimize_flow.html?view=kc): Control flow optimizations analyze the flow of control inside a method (or specific sections of it) and rearrange code paths to improve their efficiency.
      * [global optimizations](https://www.ibm.com/support/knowledgecenter/SSYKE2_8.0.0/com.ibm.java.zos.80.doc/diag/understanding/jit_optimize_global.html?view=kc): Global optimizations work on the entire method at once. They are more "expensive", requiring larger amounts of compilation time, but can provide a great increase in performance.
      * [native code generation](https://www.ibm.com/support/knowledgecenter/SSYKE2_8.0.0/com.ibm.java.zos.80.doc/diag/understanding/jit_optimize_native.html?view=kc): Native code generation processes vary, depending on the platform architecture. Generally, during this phase of the compilation, the trees of a method are translated into machine code instructions.
4. **AOT compiler**: Ahead-Of-Time compilation allows the compilation of Java™ classes into native code for subsequent executions of the same program. The AOT compiler works with the class data sharing framework.
   1. The AOT compiler is enabled by default, but is only active when shared classes are enabled.