1 Features of Java

Q1. Why is Java platform-independent but the JVM platform-dependent?

Java source code is compiled into **bytecode**, which is independent of the underlying OS or CPU.
The JVM, however, converts this bytecode into **native machine instructions**, so each platform requires its own JVM implementation.

Q2. What does “architecture neutral” mean in Java?

Java bytecode uses **fixed-size data types** (for example, `int` is always 4 bytes), regardless of the underlying hardware.
This ensures consistent behavior across different CPU architectures like x86 and ARM.

Q3. Why is Java considered a robust language?

Java enforces **strong type checking at compile time**, uses **automatic garbage collection**, and provides **exception handling**.
These features prevent memory corruption and reduce runtime failures.

Q4. How does Java ensure security?

Java uses **bytecode verification**, **class loaders**, and restricted memory access.
It does not allow direct pointer manipulation, preventing illegal memory access and malicious code execution.

Q5. Why is Java not a pure object-oriented language?

Java uses **primitive data types** like `int` and `boolean`, which are not objects.
Also, it does not support multiple inheritance of classes or operator overloading (except `+` for String).

Q6. Explain Java’s hybrid execution model.

Java code is first **compiled into bytecode**, then executed by the JVM using an **interpreter** and **JIT compiler**.
Frequently executed code paths are optimized and compiled into native code at runtime.

Q7. How does JIT compilation improve performance?

The JIT compiler identifies **hot code paths** and converts them into optimized native machine instructions.
This removes interpretation overhead and significantly improves runtime performance.

Q8. Why is Java suitable for distributed systems?

Java provides strong **networking APIs**, **serialization**, and mature frameworks like Spring and Jakarta EE.
Its platform independence makes it ideal for large-scale distributed and cloud systems.

Q9. What does "dynamic" mean in Java?

Java supports **dynamic class loading**, meaning classes are loaded only when required.
This enables plugin architectures, modular applications, and reduced memory usage.

Q10. Explain the difference between JVM, JRE, and JDK.

The **JVM** executes bytecode, the **JRE** provides JVM + core libraries, and the **JDK** includes development tools like `javac`.
Developers need the JDK, while end users typically need only the JRE.

2️⃣ Java Data Types

Q1. Why is Java called a strongly typed language?

Every variable and expression in Java has a **declared type**, which is checked at compile time.
This prevents invalid assignments and improves program reliability.

Q2. Why is `int` the default integer type?

`int` provides the best balance between **range, performance, and memory usage**.
Smaller types like `byte` and `short` are promoted to `int` during arithmetic operations anyway.

Q3. Why is `boolean` not treated as numeric in Java?

Java strictly separates logical and numeric values to avoid C-style bugs.
This enforces clarity and prevents misuse in conditions and arithmetic expressions.

Q4. Why is `char` unsigned in Java?

Java uses **Unicode (UTF-16)** for character representation.
This allows support for international characters and avoids negative character values.

Q5. What happens when a `double` is cast to `int`?

The **decimal part is truncated**, not rounded.
This can cause data loss and must be handled explicitly by the programmer.

Q6. Why must `float` literals use the `f` suffix?

Floating-point literals are treated as `double` by default.
The `f` suffix explicitly tells the compiler to treat the value as a `float`.

Q7. Why are underscores allowed in numeric literals?

Underscores improve **readability** of large numbers and are ignored by the compiler.
They cannot appear at the beginning, end, or next to decimal points.

Q8. Why does `byte + byte` result in `int`?

Java promotes operands smaller than `int` to `int` before performing arithmetic.
This prevents overflow and simplifies JVM instruction handling.

Q9. Where do default values apply in Java?

Default values are assigned only to **instance and static variables**.
Local variables must be explicitly initialized before use.

Q10. Why is `long l = 10` valid but `int i = 10L` invalid?

Assigning `int` to `long` is widening and happens implicitly.
Assigning `long` to `int` is narrowing and requires an explicit cast.

3️⃣ Types of Variables

Q1. Why don’t local variables get default values?

Local variables are stored on the **stack** and must be explicitly initialized.
This avoids accidental use of uninitialized memory and improves program correctness.

Q2. Why can’t static methods access instance variables directly?

Static methods belong to the class, not any specific object.
Instance variables require an object reference, which static methods do not have.

Q3. Explain the lifecycle difference between instance and static variables.

Instance variables are created when an object is created and destroyed during GC.
Static variables are created during class loading and exist until class unloading.

Q4. Why are local variables thread-safe by default?

Each thread has its own stack, so local variables are not shared across threads.
This eliminates concurrency issues for local variables.

Q5. Explain Java’s pass-by-value behavior for objects.

Java passes a **copy of the reference**, not the object itself.
Changes to the object are visible, but reassignment of the reference is not.

Q6. Why is `final` the only modifier allowed for local variables?

Local variables do not support access control or concurrency semantics.
`final` is allowed to enforce immutability within a method or block.

Q7. What happens if an array reference is not initialized?

The array reference defaults to `null`.
Accessing elements results in a `NullPointerException`.

Q8. Why is accessing static variables via object reference discouraged?

It compiles, but it misleads readers into thinking the variable is instance-specific.
Using the class name improves clarity and maintainability.

Q9. When does an instance variable become eligible for garbage collection?

When the object it belongs to becomes unreachable.
Instance variables cannot outlive their owning object.

Q10. Can a variable be both static and a reference?

Yes. A static reference variable points to a heap object shared across all instances.
Only the reference is static, not the object itself.

4️⃣ main() Method & Command Line Arguments

Q1. Why doesn't the compiler check for the presence of `main()`?

The compiler checks syntax, not execution entry points.
The JVM validates the `main()` method at runtime.

Q2. Why must `main()` be static?

The JVM must call `main()` without creating an object.
Making it static avoids object-creation dependencies.

Q3. Why does an incorrect `main()` signature compile but fail at runtime?

Signature validation is performed by the JVM, not the compiler.
Any mismatch causes a runtime error.

Q4. Can the `main()` method be overloaded?

Yes, but the JVM only invokes `main(String[] args)`.
Other overloads are ignored unless called explicitly.

Q5. Why are modifiers like `final` or `synchronized` allowed on `main()`?

The JVM checks only the method signature.
Other modifiers do not affect JVM invocation.

Q6. Why don't static blocks execute without `main()` in Java 7+?

Java 7 enforces an explicit entry point.
Static blocks execute only after `main()` is found.

Q7. Is `String... args` a valid `main()` signature?

Yes. Var-args are treated as a `String[]` by the JVM.
It is fully compatible.

Q8. Why are command-line arguments always Strings?

The JVM does not perform type conversion.
All arguments are passed as raw text.

Q9. Why is `args` never null?

The JVM initializes `args` automatically.
If no arguments are passed, it is an empty array.

Q10. Which `main()` method does the JVM choose if multiple exist?

The JVM always selects `public static void main(String[] args)`.
All other variations are ignored.

---

5 Operators & Assignments

Q1. Why does `b++` work but `b = b + 1` fail for byte?

`++` performs implicit casting back to `byte`.
Arithmetic operators promote operands to `int`, requiring explicit casting.

Q2. Why is `NaN != NaN` true?

NaN represents an undefined numeric result.
By IEEE-754 rules, NaN is not equal to any value, including itself.

Q3. Why does `"Java"+10+20` differ from `10+20+"Java"`?

Evaluation is left-to-right.
Once a String is encountered, all subsequent `+` operations perform concatenation.

Q4. Why is `&&` safer than `&`?

`&&` short-circuits evaluation.
It prevents unnecessary evaluation and runtime exceptions.

Q5. Why does `==` behave differently for objects?

For objects, `==` compares references, not content.
Content comparison requires `equals()`.

Q6. Why does `instanceof` return false for `null`?

`null` has no runtime type.
Therefore, it cannot be an instance of any class.

Q7. Why does `~4` produce `-5`?

The bitwise NOT operator inverts all bits.
 Two’s complement representation results in `-5`.

Q8. Why is `Class.newInstance()` deprecated?

It ignores checked exceptions and requires a no-arg constructor.
 Modern reflection APIs are safer and more flexible.

Q9. Why can’t relational operators be used on objects?

Objects do not have a natural ordering.
 Comparison must be done using `Comparable` or `Comparator`.

Q10. Difference between operator precedence and evaluation order?

Precedence decides grouping of operators.
 Evaluation order determines the sequence of operand execution.

---

6 Flow Control

Q1. Why must conditions be boolean in Java?

Java enforces strict logical expressions.
This avoids ambiguous or error-prone conditions.

Q2. Explain the dangling-else problem.

An `else` always associates with the nearest unmatched `if`.
Braces should be used to avoid ambiguity.

Q3. Why does Java disallow unreachable code?

It ensures deterministic execution paths.
Unreachable statements indicate logical errors.

Q4. Why is `while(false)` valid but its body invalid?

The loop condition is evaluated at compile time.
The loop body becomes unreachable.

Q5. Why does `switch` not support `long`?

JVM switch instructions operate on `int`.
Supporting `long` would add complexity.

Q6. Why must case labels be constants?

They must be resolved at compile time.
This enables efficient jump tables.

Q7. Why does `switch` on String use `equals()`?

String comparison is content-based, not reference-based.
`equals()` ensures correct matching.

Q8. Difference between `break` and `return`?

`break` exits a loop or switch.
`return` exits the method entirely.

Q9. Why can’t enhanced for modify array elements?

It works on a copy of values, not indices.
Structural modification is not allowed.

Q10. Why are assertions disabled by default?

Assertions are meant for debugging, not production logic.
Disabling them improves performance.

---

7 Arrays

Q1. Why are arrays objects in Java?

Arrays are stored on the heap and carry runtime type information.
 They inherit from `Object`.

Q2. Why does printing an array show `[I@...`?

It uses `Object.toString()`.
 This prints the internal class name and hashcode.

Q3. Why are zero-length arrays allowed?

They prevent null checks and support safe iteration.
 They are valid objects.

Q4. Why does negative array size throw a runtime exception?

Array size is evaluated at runtime.
 Negative sizes violate JVM constraints.

Q5. Why are Java multidimensional arrays jagged?

Java implements them as arrays of arrays.
 This improves memory efficiency and flexibility.

Q6. Why does `a = b` not copy array elements?

Only the reference is reassigned.
 Both variables point to the same array.

Q7. Why can `Object[]` store `String` but not vice versa?

Arrays are covariant.
A parent type array can store child objects, not the reverse.

Q8. Why doesn’t array-level promotion exist?

Allowing it would break type safety.
Java enforces strict array type matching.

Q9. When do arrays become eligible for GC?

When no reachable references exist.
Reassigning references can trigger eligibility.

Q10. Why is `length` a field, not a method?

Array size is fixed metadata.
Using a field is faster and simpler.

---

8 Declaration & Access Modifiers

Q1. Why can’t top-level classes be private?

They must be accessible at the package or global level.
Private visibility is only meaningful within classes.

Q2. Why is `protected` access tricky across packages?

Outside the package, access is allowed only via subclass references.
Parent references cannot access protected members.

Q3. Why are private members not inherited?

Private members are class-specific.
They are not visible to subclasses.

Q4. Why can’t abstract methods be static?

Static methods cannot be overridden.
Abstract methods require overriding.

Q5. Why doesn’t `volatile` ensure thread safety?

It guarantees visibility, not atomicity.
Race conditions can still occur.

Q6. Why can’t constructors be static?

Constructors initialize objects.
Static context exists without objects.

Q7. Why is `final` different from immutability?

`final` prevents reference reassignment.
Object state may still change.

Q8. Why is `native` incompatible with `synchronized`?

Native code handles synchronization externally.
Java synchronization cannot be enforced.

Q9. Why is `strictfp` rarely used today?

Modern hardware already follows IEEE standards.
Its use is mostly legacy.

Q10. Why does `transient` not affect garbage collection?

It only controls serialization behavior.
GC is based on reachability, not serialization.

---

9 Strings & Regular Expressions

Q1. Why are Strings immutable?

Immutability improves security, thread safety, and String Pool reuse.
It also enables caching and hashing optimizations.

Q2. How many objects are created by `new String("Java")`?

Two objects: one in the String Constant Pool and one in the heap.
The heap object references the SCP value.

Q3. Why doesn't `StringBuilder` override `equals()`?

It is mutable.
Content-based equality would break hash-based collections.

Q4. Why is `StringBuffer` slower than `StringBuilder`?

Its methods are synchronized.
This adds locking overhead.

Q5. Why is `matches()` stricter than `find()`?

`matches()` requires the entire string to match the regex.
`find()` searches for any matching substring.

Q6. Why should regex be compiled using `Pattern`?

Compiling once avoids repeated parsing.
This significantly improves performance.

Q7. Why does `substring()` exclude the end index?

It follows half-open interval design.
This simplifies length calculation and chaining.

Q8. Why is `intern()` risky in large applications?

It stores strings in the String Pool.
Excessive usage can increase memory pressure.

Q9. Why does `replaceAll()` use regex instead of literals?

It performs pattern-based replacement.
For literal replacement, `replace()` should be used.

Q10. Difference between `find()`, `matches()`, and `lookingAt()`?

`find()` matches anywhere, `matches()` matches entire input, and `lookingAt()` matches only the prefix.
Each serves a different matching purpose.