**Section 1: Java Introduction**

**1. What is Java?**Java is a class-based, object-oriented, platform-independent programming language developed by Sun Microsystems (now Oracle).

* Write Once, Run Anywhere (WORA): Achieved via bytecode and JVM.
* Automatic memory management: Handled by Garbage Collection.
* Strongly typed & compiled: Ensures reliability.
* Multi-threaded & secure: Supports concurrent execution and robust security.

**2. If Java is platform-independent, why do we still need different JVMs for Windows, Linux, and Mac?**

* Java code compiles into bytecode, which is universal.
* However, the JVM implementation differs per OS so it can interact with system libraries.

👉 Think of it as:

* Bytecode = Universal language
* JVM = Translator for each OS

**3. What is the difference between JDK, JRE, and JVM?**

* JVM (Java Virtual Machine): Executes Java bytecode, abstracts hardware/OS.
* JRE (Java Runtime Environment): JVM + core libraries (needed to run apps).
* JDK (Java Development Kit): JRE + compiler (javac) + tools (for development).

**4. If you compile a Java program using JDK 17 and run it on JRE 8, will it work?**❌ No. Newer compiled code may contain bytecode instructions that the older JVM doesn’t understand.

**5. Is Java 100% Object-Oriented?**No. Java has primitives (int, char, boolean, etc.) for efficiency.

* But Java also provides wrapper classes (Integer, Character, Boolean) to use primitives as objects.

**6. If primitives aren’t objects, how can we use them in collections like ArrayList<Integer>?**  
Java uses autoboxing: primitives are automatically converted to their wrapper classes.

**7. How does Java achieve platform independence?**

1. Write .java source code.
2. Compile with javac → generates .class bytecode.
3. JVM (OS-specific) loads bytecode.
4. JIT compiler converts bytecode → machine code.
5. OS executes native code.

**8. What happens when you run a Java program?**

* Compilation: javac → .class (bytecode).
* Class Loading: JVM loads bytecode using ClassLoader.
* Bytecode Verification: Security & correctness check.
* Execution: Interpreter + JIT compilation.

**9. What is a ClassLoader in Java?**It is a subsystem of JVM that loads .class files into memory.

* Follows parent delegation model.
* Ensures class uniqueness.

**10. Java Compilation & Execution Flow**

.java (Source Code)

↓ javac (Compiler)

.class (Bytecode)

↓ JVM

Class Loader → Bytecode Verifier → Runtime Data Areas

↓ Execution Engine (Interpreter / JIT)

Native Machine Code → Program Execution

**Section 2 : Java Variables & Data Types**

**A. Basics of Variables**

**1. What is a variable in Java?**  
A variable is a name given to a memory location that stores data. It allows programs to store, retrieve, and manipulate values during execution.  
Example:

int age = 25;

**2. How do you declare and initialize a variable in Java?**

* Declaration: Specify data type + variable name
* Initialization: Assign a value  
  Example:

int x; // declaration

x = 10; // initialization

int y = 20; // declaration + initialization

**3. What are the rules and conventions for naming variables in Java?**

* Must begin with a letter, $, or \_
* Cannot be a keyword
* Case sensitive
* Convention: use **camelCase** for variables  
  Example: firstName, totalAmount

**4. What are the different types of variables in Java?**

* **Local variables** – declared inside a method/constructor/block, no default values
* **Instance variables** – declared in a class, not static, each object has its own copy
* **Static variables** – declared with static, shared among all objects of a class

**5. What is the difference between instance, class (static), and local variables?**

* **Local**: inside methods, no default value, scope is method/block
* **Instance**: tied to an object, default values exist, scope is object
* **Static**: one copy shared across all objects, stored in class memory

**6. What is the default value of instance variables in Java?**

* int, byte, short, long → 0
* float, double → 0.0
* char → \u0000
* boolean → false
* Object references → null

**7. Do local variables get default values in Java?**  
No. Local variables **must be initialized** before use, otherwise it results in a compilation error.

**8. Can a local variable be declared without initialization?**  
Yes, but you must assign a value before use.

int x;

System.out.println(x); // ERROR

**9. What is the scope of a variable in Java?**

* Local → within method/block
* Instance → tied to object instance
* Static → throughout program class

**10. What is the lifetime of local, instance, and static variables?**

* Local → created when method is called, destroyed after exit
* Instance → exists as long as object exists
* Static → exists for the lifetime of the class (JVM process)

**11. What is variable shadowing in Java?**  
When a local variable has the same name as an instance/static variable, the local one “shadows” it.

class A {

int x = 10;

void show() {

int x = 20;

System.out.println(x); // prints 20

}

}

**12. What is the difference between a reference variable and a primitive variable?**

* Primitive variable → stores actual value
* Reference variable → stores memory address of object

**13. Can two variables have the same name in Java?**  
Yes, if they are in different scopes.  
Example: a local variable can have the same name as an instance variable (shadowing).

**14. What is the difference between final, static, and transient variables?**

* final → value cannot change after initialization
* static → shared across all objects
* transient → ignored during serialization

**15. What is the use of the volatile keyword in Java variables?**  
Ensures changes made by one thread are immediately visible to other threads. Prevents caching.

**16. How are variables stored in memory (Stack vs Heap)?**

* Local (primitives) → Stack
* Object references → Stack (reference) + Heap (object data)
* Static variables → Method area (special memory)

**17. What is the difference between declaring a variable and defining a variable?**

* Declaration: only tells the compiler about variable name & type
* Definition: allocates memory and may assign a value

**18. Can we change the data type of a variable after declaration?**  
No. Once declared, the type is fixed.

**19. What is the difference between primitive variables and objects?**

* Primitive → lightweight, stored directly in stack, faster
* Object → stored in heap, accessed via reference, heavier

**20. What are compile-time constants in Java?**  
Variables declared as static final. They must be assigned at declaration or in static block.

**B. Primitive Data Types**

**1. How many primitive data types are there in Java?**  
8 → byte, short, int, long, float, double, char, boolean

**2. What are the default values of all primitive data types?**

* byte/short/int/long → 0
* float/double → 0.0
* char → \u0000
* boolean → false

**3. Why does Java have both float and double?**

* float (32-bit) → less precision, saves memory
* double (64-bit) → more precision, default for decimals

**4. Why is char in Java a 16-bit data type?**  
To support Unicode (international characters), not just ASCII (8-bit).

**5. Why is boolean not given a size in Java?**  
Because it depends on JVM implementation. Only values: true or false.

**6. Can you store a char in an int variable?**  
Yes. char is implicitly promoted to int (ASCII/Unicode value).

**7. What is the range of byte?**  
-128 to +127 (1 byte, signed two’s complement).

**8. Why does Java use int as the default for integers?**

* Balanced size (32-bit)
* Efficient for most processors (native word size)

**9. What is the difference between float and double in terms of precision?**

* float → 6–7 decimal digits
* double → 15–16 decimal digits

**10. Can a long store a float value without casting?**  
No. Requires explicit casting because of possible precision loss.

**11. What happens if you assign a value outside the range of byte?**  
Compilation error unless cast. Casting results in overflow (wrap-around).

**12. How are negative numbers stored in Java?**  
Using **Two’s complement** representation.

**13. Why does Java not support unsigned integers?**  
To keep language simple and portable across architectures.

**14. Can you assign null to a primitive type?**  
No. Only reference types can hold null.

**15. Why do you need f or F for float literals?**  
By default decimals are double. Example:

float x = 3.14f;

**16. What is type promotion in Java?**  
When smaller data types are automatically promoted to larger ones in expressions.  
Example: byte + byte → int

**17. What is type casting in Java?**

* **Widening (automatic)**: int → long
* **Narrowing (manual)**: double → int

**18. What happens when you add an int and a double?**  
Result will be double (wider type wins).

**19. How does Java handle integer overflow?**  
Wraps around silently (no error).

**20. What are literals in Java?**  
Constant values directly written in code.  
Example: 10, 3.14f, 'A', "Hello"

**C. Wrapper Classes & Autoboxing**

**1. What are wrapper classes in Java?**  
Classes that convert primitives into objects. Example: Integer, Double.

**2. How do you convert primitive to wrapper?**  
Using valueOf() or autoboxing:

Integer i = Integer.valueOf(10);

Integer j = 10; // autoboxing

**3. How do you convert wrapper to primitive?**  
Using xxxValue() or unboxing:

int x = i.intValue();

int y = j; // unboxing

**4. What is autoboxing and unboxing?**

* Autoboxing: primitive → wrapper automatically
* Unboxing: wrapper → primitive automatically

**5. Why are wrapper classes immutable?**  
To ensure caching, thread-safety, and reliable behavior.

**6. What is the difference between == and .equals() for wrappers?**

* == → compares references
* .equals() → compares values

**7. Why is Integer.valueOf() preferred over new Integer()?**  
Because it uses caching for small integers, saving memory.

**8. What is the difference between Integer.parseInt() and Integer.valueOf()?**

* parseInt() → returns primitive int
* valueOf() → returns Integer object

**9. How does caching work in wrapper classes?**  
Integer, Short, Byte, Character (-128 to 127) values are cached.

**10. What happens when comparing two Integer objects using ==?**  
If values are in cache range, == may return true; otherwise false.

**D. Advanced / Tricky Questions**

**1. What is the difference between final, const, and readonly?**

* final → Java keyword, variable can’t be reassigned
* const → reserved but unused in Java
* readonly → used in C#, not in Java

**2. Why doesn’t Java support unsigned data types?**  
To avoid confusion and keep the language portable.

**3. Is String a data type in Java?**  
No. String is a class in java.lang package.

**4. Why is String immutable?**  
For security, caching, and thread-safety.

**5. What is the size of an object reference variable?**  
Depends on JVM (usually 4 bytes on 32-bit, 8 bytes on 64-bit).

**6. What is the difference between static and non-static variables?**

* Static → one copy shared by all objects
* Non-static → separate copy per object

**7. Can we declare variables inside an interface?**  
Yes. They are implicitly public static final.

**8. What is a transient variable?**  
Excluded from serialization. Used to hide sensitive data.

**9. What is the difference between a constant (final) and immutable object?**

* Constant → reference cannot change
* Immutable → object’s content cannot change

**10. How do variables behave in multithreaded environments?**

* Each thread gets its own stack (local variables)
* Shared variables need synchronization (volatile, synchronized)

**11. What is shallow vs deep copy?**

* Shallow → copies references only
* Deep → copies actual objects

**12. Can we have uninitialized variables in Java?**

* Instance variables → default values
* Local variables → must be initialized

**13. What is difference between class variable and global variable?**  
Java does not have global variables. Class variable = static variable in a class.

**14. Why can’t local variables be static?**  
Because static variables belong to a class, not a block/method.

**15. Primitive vs Reference equality?**

* Primitive → compared by value
* Reference → compared by reference (unless .equals() is overridden)

**Section 3: Java Operators & Expressions  
  
A. Basics of Operators**

**Q1. What are operators in Java?**  
Operators are special symbols in Java that perform specific operations on operands (variables and values). For example, +, -, \*, / are arithmetic operators.

**Q2. What are expressions in Java?**  
An expression is a combination of variables, operators, and values that evaluates to a single result.  
Example: int x = 5 + 3 \* 2; → expression is 5 + 3 \* 2, result is 11.

**Q3. What are the categories of operators in Java?**

1. Arithmetic Operators
2. Relational Operators
3. Logical Operators
4. Bitwise Operators
5. Assignment Operators
6. Unary Operators
7. Conditional (Ternary) Operator
8. Shift Operators
9. Instanceof Operator

**B. Arithmetic Operators**

**Q4. What are arithmetic operators in Java?**  
They perform basic mathematical operations:

* Addition (+)
* Subtraction (-)
* Multiplication (\*)
* Division (/)
* Modulus (%)

**Q5. What is the difference between / and % operator?**

* / → returns quotient. Example: 10 / 3 = 3
* % → returns remainder. Example: 10 % 3 = 1

**Q6. What happens if you divide an integer by zero in Java?**

* For integers → ArithmeticException: / by zero
* For floating-point numbers (float/double) → results in Infinity or NaN.

**Q7. What is operator precedence and associativity?**

* **Precedence** → decides which operator executes first. Example: \* has higher precedence than +.
* **Associativity** → decides order when precedence is same. Most are left-to-right, but = and ternary ?: are right-to-left.

**C. Unary Operators**

**Q8. What are unary operators in Java?**  
Operators that work on a single operand. Examples:

* Unary plus (+)
* Unary minus (-)
* Increment (++)
* Decrement (--)
* Logical NOT (!)
* Bitwise complement (~)

**Q9. What is the difference between pre-increment and post-increment?**

* **Pre-increment (++x)** → increments first, then uses the value.
* **Post-increment (x++)** → uses the value first, then increments.

Example:

int x = 5;

System.out.println(++x); // 6

System.out.println(x++); // 6 (then x becomes 7)

**D. Relational Operators**

**Q10. What are relational operators in Java?**  
They compare two values and return a boolean result:

* ==, !=, <, >, <=, >=

**Q11. What is the difference between == and .equals()?**

* == → compares **references** for objects, values for primitives.
* .equals() → compares **content** of objects (if overridden).

Example:

String a = new String("test");

String b = new String("test");

System.out.println(a == b); // false

System.out.println(a.equals(b)); // true

**E. Logical Operators**

**Q12. What are logical operators in Java?**

* && → Logical AND
* || → Logical OR
* ! → Logical NOT

**Q13. What is short-circuiting in logical operators?**

* && and || use short-circuiting.
* If the left operand decides the result, the right operand is not evaluated.  
  Example:

if (x != 0 && 10/x > 1) {...}

→ If x == 0, second condition never executes (prevents divide by zero).

**Q14. What is the difference between & and &&?**

* & → Bitwise AND (also works as logical AND without short-circuit).
* && → Logical AND with short-circuit evaluation.

**F. Bitwise & Shift Operators**

**Q15. What are bitwise operators?**  
Work on individual bits:

* & → AND
* | → OR
* ^ → XOR
* ~ → Complement

**Q16. What are shift operators in Java?**

* << → Left shift (multiplies by 2^n)
* >> → Right shift (divides by 2^n, keeps sign)
* >>> → Unsigned right shift (fills with zero, ignores sign)

**Q17. What is the difference between >> and >>>?**

* >> → Sign-propagating right shift (preserves negative sign).
* >>> → Zero-fill right shift (fills with 0 regardless of sign).

**G. Assignment Operators**

**Q18. What are assignment operators in Java?**

* = (basic assignment)
* Compound assignment: +=, -=, \*=, /=, %=, &=, |=, ^=, <<=, >>=, >>>=

**Q19. What is the difference between x = x + y and x += y?**  
Functionally similar, but:

* x += y includes implicit type casting.  
  Example:

byte b = 10;

// b = b + 1; // Compilation error (int result)

// b += 1; // Works (auto cast)

**H. Conditional / Miscellaneous Operators**

**Q20. What is the ternary operator in Java?**

* Syntax: condition ? expr1 : expr2
* If condition is true → returns expr1, else → returns expr2.

**Q21. What is the instanceof operator?**  
Checks if an object is an instance of a class or subclass.  
Example:

String s = "hello";

System.out.println(s instanceof String); // true

**Q22. What is the difference between instanceof and getClass()?**

* instanceof → checks type with inheritance considered.
* getClass() → returns the exact runtime class, does not consider inheritance.

**I. Advanced / Tricky Questions**

**Q23. What happens when you add two char variables in Java?**  
They are promoted to int before operation.  
Example: 'A' + 'B' → 65 + 66 = 131.

**Q24. What is operator overloading? Is it supported in Java?**  
Operator overloading means redefining operators for custom use. Java does not support it (except + for String concatenation).

**Q25. What is the difference between ++i + i++ in an expression?**  
They behave differently depending on evaluation order. Example:

int i = 5;

System.out.println(i++ + ++i); // 5 + 7 = 12

**Q26. How does Java handle integer overflow with operators?**  
Java does not throw an error; it wraps around.  
Example:

int x = Integer.MAX\_VALUE;

System.out.println(x + 1); // -2147483648

**Q27. What is the difference between && and & when used with booleans?**

* && → logical AND with short-circuiting.
* & → evaluates both sides always (slower, but sometimes required).

**Q28. What is the difference between == and =?**

* = → assignment operator.
* == → comparison operator.

**Q29. Can we use operators with null in Java?**

* Primitive operators (+, -, etc.) cannot be applied to null.
* But instanceof with null always returns false.

**Q30. What happens if you concatenate null with a String using +?**  
 It results in "null" as text.  
Example: "abc" + null = "abcnull".

**Section 4: Strings**

**1. What is a String in Java?**

**Answer:**

* In Java, a String is a sequence of characters.
* Unlike other languages, Strings in Java are **objects** of the String class (in java.lang package).
* Example:
* String name = "Alice"; // using string literal
* String city = new String("Paris"); // using new keyword

**2. Why are Strings immutable in Java?**

**Answer:**

* Once a String object is created, its value **cannot be changed**.
* Any modification results in the creation of a **new String object**.
* Reasons:
  1. **Security:** Prevents altering sensitive data like database URLs, usernames, or passwords.
  2. **Caching:** Enables reusability via the String pool.
  3. **Thread-safety:** Multiple threads can safely share a String object.
  4. **HashCode stability:** Useful when Strings are used as keys in hash-based collections.

**3. What is the String Constant Pool (SCP)?**

**Answer:**

* A special memory region inside the Java Heap where string literals are stored.
* When you create a string literal, JVM first checks SCP:
  + If present → reference is reused.
  + If not → a new object is created in the pool.
* Example:
* String s1 = "Java";
* String s2 = "Java";
* System.out.println(s1 == s2); // true → both point to same pool object

**4. Difference between String, StringBuilder, and StringBuffer?**

**Answer:**

| **Feature** | **String** | **StringBuilder** | **StringBuffer** |
| --- | --- | --- | --- |
| **Mutability** | Immutable | Mutable | Mutable |
| **Thread-safety** | Not thread-safe | Not thread-safe | Thread-safe (synchronized) |
| **Performance** | Slower for frequent modifications | Faster | Slower (due to synchronization) |

**5. How do you compare Strings in Java?**

**Answer:**

1. equals() → Compares **content**.
2. String a = "Hello";
3. String b = "Hello";
4. System.out.println(a.equals(b)); // true
5. == → Compares **reference (memory address)**.
6. String x = new String("Test");
7. String y = new String("Test");
8. System.out.println(x == y); // false (different objects)
9. compareTo() → Lexicographic comparison.
10. "abc".compareTo("abd"); // -1

**6. What are common String methods in Java?**

**Answer:**  
Some frequently used methods:

* length() → Returns length of String
* charAt(int index) → Returns char at position
* substring(begin, end) → Extracts part of string
* equals(), equalsIgnoreCase() → Content comparison
* indexOf(), lastIndexOf() → Finds positions
* toLowerCase(), toUpperCase() → Case conversion
* trim() → Removes leading/trailing spaces
* split() → Splits into an array
* replace(), replaceAll() → Replaces characters or regex

Example:

String str = " Java ";

System.out.println(str.trim().toUpperCase()); // "JAVA"

**7. What is String concatenation?**

**Answer:**

* Strings can be joined using + operator or concat() method.
* String s1 = "Hello";
* String s2 = "World";
* System.out.println(s1 + " " + s2); // "Hello World"
* Internally, + is converted to StringBuilder.append() in compiled code.

**8. What is the difference between String.valueOf() and toString()?**

**Answer:**

* toString() → Used to convert object into string (if class overrides it).
* String.valueOf() → Converts any primitive or object to String safely (null becomes "null").

Example:

int num = 50;

String s1 = String.valueOf(num); // "50"

Object obj = null;

String s2 = String.valueOf(obj); // "null"

**9. What are mutable alternatives to String?**

**Answer:**

* **StringBuilder** and **StringBuffer** allow modifications without creating new objects.
* Example:
* StringBuilder sb = new StringBuilder("Hi");
* sb.append(" there!");
* System.out.println(sb); // "Hi there!"

**10. What is intern() in String?**

**Answer:**

* intern() adds the string to SCP (if not already present).
* Ensures reference from pool is used.  
  Example:

String s1 = new String("Hello");

String s2 = s1.intern();

String s3 = "Hello";

System.out.println(s2 == s3); // true

**11. How to convert between String and other types?**

**Answer:**

* **To primitive:**
* int num = Integer.parseInt("123");
* double d = Double.parseDouble("12.34");
* **From primitive:**
* String s = String.valueOf(100);
* **To char array:**
* char[] arr = "Hello".toCharArray();
* **From char array:**
* String str = new String(arr);

**12. What is the difference between replace() and replaceAll()?**

**Answer:**

* replace(char, char) or replace(CharSequence, CharSequence) → Replaces characters or substrings (no regex).
* replaceAll(regex, replacement) → Uses **regular expressions**.

Example:

String s = "abc123";

System.out.println(s.replace("a","x")); // xbc123

System.out.println(s.replaceAll("\\d","")); // abc

**13. How do you reverse a String in Java?**

**Answer:**  
Options:

1. Using StringBuilder:
2. String str = "Hello";
3. String reversed = new StringBuilder(str).reverse().toString();
4. Manual loop with char array.

**14. What is difference between substring() and subSequence()?**

**Answer:**

* substring(int, int) → Returns a String.
* subSequence(int, int) → Returns a CharSequence (interface).

Example:

String str = "abcdef";

System.out.println(str.substring(2,5)); // "cde"

System.out.println(str.subSequence(2,5)); // "cde"

**15. How do you handle Unicode and special characters in Strings?**

**Answer:**

* Java Strings are internally stored as **UTF-16**.
* Supports Unicode characters like emojis and symbols.
* Example:
* String smiley = "\u263A";
* System.out.println(smiley); // ☺

**16. What are String Joiners in Java 8?**

**Answer:**

* Java 8 introduced String.join() and StringJoiner to build strings with delimiters.  
  Example:

String joined = String.join(", ", "A", "B", "C");

System.out.println(joined); // A, B, C

**17. What is difference between split() and tokenizer?**

**Answer:**

* split() → Returns array, uses regex.
* StringTokenizer → Legacy class, splits tokens based on delimiters.  
  Example:

String str = "A-B-C";

String[] arr = str.split("-");

System.out.println(Arrays.toString(arr)); // [A, B, C]

**18. Can Strings be null or empty? Difference between both?**

**Answer:**

* null → Reference does not point to any object.
* Empty String ("") → Object exists but contains no characters.  
  Example:

String a = null;

String b = "";

System.out.println(a == null); // true

System.out.println(b.isEmpty()); // true

**Section 5: Conditions in Java**

**1. What are conditional statements in Java? Why are they important?**

**Answer:**  
Conditional statements are decision-making constructs in Java that allow a program to execute certain blocks of code depending on whether a condition is true or false.  
They are important because they help control the program’s flow and allow branching logic instead of executing code sequentially only.

**2. What are the different types of conditional statements in Java?**

**Answer:**  
Java provides the following types of conditional statements:

1. **if statement**
2. **if-else statement**
3. **if-else-if ladder**
4. **nested if statements**
5. **switch statement** (alternative to multiple if-else)

**3. How does the if statement work in Java?**

**Answer:**  
The if statement checks a condition, and if the condition evaluates to **true**, the associated block of code executes.  
If false, the block is skipped.

**Example:**

int age = 20;

if (age >= 18) {

System.out.println("Eligible to vote");

}

**4. Explain the difference between if and if-else.**

**Answer:**

* **if**: Executes a block only if the condition is true.
* **if-else**: Executes one block if the condition is true, and another block if the condition is false.

**Example:**

int num = 5;

if (num % 2 == 0) {

System.out.println("Even number");

} else {

System.out.println("Odd number");

}

**5. What is an if-else-if ladder? When do we use it?**

**Answer:**  
An **if-else-if ladder** is used when multiple conditions need to be checked sequentially.  
As soon as one condition is true, its block executes and the rest are skipped.

**Example:**

int marks = 75;

if (marks >= 90) {

System.out.println("Grade A");

} else if (marks >= 75) {

System.out.println("Grade B");

} else if (marks >= 60) {

System.out.println("Grade C");

} else {

System.out.println("Grade D");

}

**6. What are nested if statements? Give an example.**

**Answer:**  
A **nested if** is an if statement inside another if.  
It is used when a decision depends on another condition.

**Example:**

int age = 25;

boolean hasID = true;

if (age >= 18) {

if (hasID) {

System.out.println("Entry allowed");

} else {

System.out.println("ID required");

}

}

**7. What is a switch statement in Java?**

**Answer:**  
A **switch statement** allows multi-branching decisions based on the value of an expression.  
It is an alternative to multiple if-else statements for better readability.

**Example:**

int day = 3;

switch (day) {

case 1: System.out.println("Monday"); break;

case 2: System.out.println("Tuesday"); break;

case 3: System.out.println("Wednesday"); break;

default: System.out.println("Invalid day");

}

**8. What are the rules for using switch statements in Java?**

**Answer:**

* The expression inside switch must evaluate to **byte, short, char, int, String, or enum** (Java 7+ allows String).
* **case** values must be constants or literals, not variables.
* **break** is used to prevent fall-through (optional but recommended).
* A **default** block is optional but used as a fallback.

**9. What is fall-through in switch statements?**

**Answer:**  
Fall-through occurs when a case executes but does not contain a break statement, causing subsequent cases to execute until a break is found.

**Example:**

int day = 2;

switch(day) {

case 1: System.out.println("Monday");

case 2: System.out.println("Tuesday");

case 3: System.out.println("Wednesday");

}

// Output:

// Tuesday

// Wednesday

**10. Difference between if-else-if ladder and switch statement?**

**Answer:**

| **Feature** | **if-else-if ladder** | **switch statement** |
| --- | --- | --- |
| Data types allowed | Any boolean condition | Only byte, short, int, char, String, enum |
| Readability | Complex for many conditions | More readable for multi-branching |
| Fall-through | Not possible | Possible if break is missing |
| Performance | Slower for many conditions | Faster with jump tables (compiler optimized) |

**11. Can we use multiple conditions in switch?**

**Answer:**  
No, switch only evaluates one expression.  
If multiple conditions are needed (e.g., ranges), if-else-if should be used.

**12. Can we use logical operators in if conditions?**

**Answer:**  
Yes, if conditions accept any boolean expression including logical operators (&&, ||, !).

**Example:**

int age = 20;

boolean hasID = true;

if (age >= 18 && hasID) {

System.out.println("Entry allowed");

}

**13. Can we use relational operators inside conditions?**

**Answer:**  
Yes, relational operators (>, <, ==, !=, >=, <=) are often used to evaluate conditions.

**14. What happens if we don’t write a break in switch?**

**Answer:**  
The program will execute all subsequent cases until it encounters a break or the switch ends (fall-through behavior).

**15. Can we nest switch statements in Java?**

**Answer:**  
Yes, switch statements can be nested inside other switches or conditions.

**Example:**

int num = 2;

switch(num) {

case 1: System.out.println("One"); break;

case 2:

int subCase = 1;

switch(subCase) {

case 1: System.out.println("SubCase One"); break;

}

break;

}

**16. What is the difference between conditional statements and loops?**

**Answer:**

* **Conditional statements** decide **which block of code** to execute.
* **Loops** decide **how many times a block of code executes**.

**17. What is the ternary operator in Java? Is it related to conditions?**

**Answer:**  
Yes. The **ternary operator** (?:) is a shorthand for if-else.  
It returns one of two values depending on a condition.

**Example:**

int age = 20;

String result = (age >= 18) ? "Adult" : "Minor";

System.out.println(result);

**Section 6: Loop Control Instructions in Java**

**1. What is a loop in Java? Why are loops used?**

**Answer:**  
A loop in Java is a control structure that allows repeated execution of a block of code until a condition is met.

* **Why used?** To avoid code duplication and make programs concise and maintainable.
* Instead of writing the same statement multiple times, loops allow dynamic repetition.

**2. What are the different types of loops in Java?**

**Answer:**  
Java provides **three main types of loops**:

1. **for loop** → When number of iterations is known.
2. **while loop** → When condition-based repetition is required.
3. **do-while loop** → When code must execute at least once before checking the condition.

**3. Explain the syntax of a for loop with an example.**

**Answer:**

for (initialization; condition; update) {

// body of loop

}

* **Initialization**: executed once before the loop starts.
* **Condition**: checked before every iteration. If false, loop exits.
* **Update**: executed after every iteration.

**Example:**

for (int i = 1; i <= 5; i++) {

System.out.println(i);

}

Output: 1 2 3 4 5

**4. What is the difference between for, while, and do-while loops?**

**Answer:**

| **Feature** | **for loop** | **while loop** | **do-while loop** |
| --- | --- | --- | --- |
| Condition check | Before execution | Before execution | After execution |
| Execution guarantee | May not run | May not run | Runs at least once |
| Usage | Known iteration count | Condition-driven repetition | Must execute once |

**5. Explain the while loop with an example.**

**Answer:**

while (condition) {

// body of loop

}

* Condition is checked first. If true → loop executes, else → exits.

**Example:**

int i = 1;

while (i <= 5) {

System.out.println(i);

i++;

}

Output: 1 2 3 4 5

**6. Explain the do-while loop with an example.**

**Answer:**

do {

// body of loop

} while (condition);

* Executes body **once** even if condition is false.

**Example:**

int i = 6;

do {

System.out.println(i);

i++;

} while (i <= 5);

Output: 6 (runs once even though condition fails).

**7. What is an infinite loop? Give an example.**

**Answer:**  
An infinite loop runs endlessly because the termination condition is never met.

**Example using for:**

for (;;) {

System.out.println("Infinite Loop");

}

**Example using while:**

while (true) {

System.out.println("Infinite Loop");

}

**8. What is the difference between break and continue statements?**

**Answer:**

* **break**: Terminates the loop immediately and control moves outside the loop.
* **continue**: Skips the current iteration and jumps to the next iteration.

**Example (break):**

for (int i = 1; i <= 5; i++) {

if (i == 3) break;

System.out.println(i);

}

Output: 1 2

**Example (continue):**

for (int i = 1; i <= 5; i++) {

if (i == 3) continue;

System.out.println(i);

}

Output: 1 2 4 5

**9. Can we use break and continue in nested loops?**

**Answer:**  
Yes. In nested loops,

* break exits only the innermost loop by default.
* We can use **labeled break/continue** to control outer loops.

**Example (labeled break):**

outer: for (int i = 1; i <= 3; i++) {

for (int j = 1; j <= 3; j++) {

if (i == 2 && j == 2) break outer;

System.out.println(i + " " + j);

}

}

Output:

1 1

1 2

1 3

2 1

(Loop exits completely when i=2, j=2)

**10. What is the use of the return statement inside loops?**

**Answer:**

* return ends the method execution entirely (not just the loop).
* Once return executes, control goes back to the caller function.

**Example:**

public static void testLoop() {

for (int i = 1; i <= 5; i++) {

if (i == 3) return;

System.out.println(i);

}

System.out.println("End");

}

Output:

1

2

(Method exits completely before printing "End").

**11. What are nested loops? Give an example.**

**Answer:**  
A loop inside another loop is called a nested loop. Commonly used for working with **matrices, patterns, grids**.

**Example:**

for (int i = 1; i <= 3; i++) {

for (int j = 1; j <= 3; j++) {

System.out.print(i + "," + j + " ");

}

System.out.println();

}

Output:

1,1 1,2 1,3

2,1 2,2 2,3

3,1 3,2 3,3

**12. Can loops be used without curly braces {}?**

**Answer:**  
Yes, if there is only **one statement** inside the loop.

**Example:**

for (int i = 1; i <= 3; i++)

System.out.println(i);

Output:

1

2

3

**13. Which loop is best when the number of iterations is known/unknown?**

**Answer:**

* **Known iterations** → for loop.
* **Unknown iterations** → while loop.
* **At least one execution required** → do-while loop.

**14. Can we modify the loop variable inside the loop body?**

**Answer:**  
Yes, but it’s generally not recommended as it may cause **unexpected behavior** or infinite loops.

**Example:**

for (int i = 0; i < 5; i++) {

i += 2; // modifies loop variable

System.out.println(i);

}

**15. What are enhanced for-each loops? When are they used?**

**Answer:**

* A **for-each loop** is used to iterate over collections/arrays without using an index.
* Provides cleaner syntax.

**Example:**

int[] arr = {1, 2, 3, 4};

for (int num : arr) {

System.out.println(num);

}

Output: 1 2 3 4

**Section 7: Arrays**

**Q1. What is an array in Java?**

**Answer:**

* An **array** is a data structure in Java that stores multiple values of the **same data type** in a contiguous memory location.
* Each element in an array is accessed using an **index**, starting from 0.

**Example:**

int[] numbers = {10, 20, 30, 40};

System.out.println(numbers[0]); // 10

System.out.println(numbers[3]); // 40

**Q2. How do you declare and initialize arrays in Java?**

**Answer:**  
There are multiple ways:

1. **Declaration + Initialization separately:**

int[] arr = new int[5]; // size 5, default values 0

arr[0] = 10;

arr[1] = 20;

1. **Declaration with initialization:**

int[] arr = {1, 2, 3, 4, 5};

1. **Using new keyword with values:**

int[] arr = new int[]{10, 20, 30};

**Q3. What are the default values of array elements in Java?**

**Answer:**

* For numeric types (int, double, etc.) → 0
* For boolean → false
* For char → \u0000 (null character)
* For reference types (e.g., String) → null

**Example:**

int[] nums = new int[3];

System.out.println(nums[0]); // 0

boolean[] flags = new boolean[2];

System.out.println(flags[1]); // false

**Q4. What is the length of an array in Java? Can it change after initialization?**

**Answer:**

* The length of an array is fixed once it is created.
* You can access it using the **array.length** property.
* It **cannot** be changed dynamically.

**Example:**

int[] arr = {1, 2, 3};

System.out.println(arr.length); // 3

// arr.length = 5; ❌ Not allowed

**Q5. How do you iterate over an array in Java?**

**Answer:**  
There are multiple ways:

1. **For loop:**

int[] arr = {10, 20, 30};

for (int i = 0; i < arr.length; i++) {

System.out.println(arr[i]);

}

1. **Enhanced for loop (for-each):**

for (int num : arr) {

System.out.println(num);

}

1. **Using Arrays.toString():**

import java.util.Arrays;

System.out.println(Arrays.toString(arr)); // [10, 20, 30]

**Q6. What is the difference between array.length and arr.length()?**

**Answer:**

* array.length → Property of an array (no parentheses).
* arr.length() → Method used in Strings and collections, **not in arrays**.
* If you try arr.length(), it will give a compilation error.

**Q7. Can arrays store objects in Java?**

**Answer:**  
Yes, arrays can store objects since objects are reference types.

**Example:**

String[] names = {"Alice", "Bob", "Charlie"};

System.out.println(names[1]); // Bob

**Q8. What happens if you access an index outside the array size?**

**Answer:**

* Java throws a **ArrayIndexOutOfBoundsException**.

**Example:**

int[] arr = {1, 2, 3};

System.out.println(arr[5]); // Exception

**Q9. What is the difference between 1D and 2D arrays?**

**Answer:**

* **1D array:** Linear collection of elements.

int[] arr = {1, 2, 3};

* **2D array:** Array of arrays (matrix-like).

int[][] matrix = {

{1, 2, 3},

{4, 5, 6}

};

System.out.println(matrix[1][2]); // 6

**Q10. How do you create a jagged array in Java?**

**Answer:**  
A **jagged array** is an array of arrays with different column sizes.

**Example:**

int[][] jagged = new int[3][];

jagged[0] = new int[2];

jagged[1] = new int[4];

jagged[2] = new int[1];

System.out.println(jagged[1].length); // 4

**Q11. How do you copy an array in Java?**

**Answer:**

1. **Using loop:**

int[] arr1 = {1, 2, 3};

int[] arr2 = new int[arr1.length];

for (int i = 0; i < arr1.length; i++) {

arr2[i] = arr1[i];

}

1. **Using Arrays.copyOf():**

import java.util.Arrays;

int[] arr2 = Arrays.copyOf(arr1, arr1.length);

1. **Using System.arraycopy():**

System.arraycopy(arr1, 0, arr2, 0, arr1.length);

**Q12. How do you sort an array in Java?**

**Answer:**  
Use Arrays.sort():

import java.util.Arrays;

int[] nums = {5, 2, 8, 1};

Arrays.sort(nums);

System.out.println(Arrays.toString(nums)); // [1, 2, 5, 8]

**Q13. What are Arrays.equals() and Arrays.deepEquals()?**

**Answer:**

* Arrays.equals(arr1, arr2) → Compares elements of **1D arrays**.
* Arrays.deepEquals(arr1, arr2) → Compares elements of **multi-dimensional arrays**.

**Example:**

int[][] a = {{1,2}, {3,4}};

int[][] b = {{1,2}, {3,4}};

System.out.println(Arrays.equals(a, b)); // false

System.out.println(Arrays.deepEquals(a, b)); // true

**Q14. Can you increase the size of an array in Java?**

**Answer:**

* No, arrays are **fixed-size**.
* To "resize," create a new array with larger size and copy elements.
* Alternative: Use **ArrayList** which is dynamic.

**Q15. Difference between Array and ArrayList in Java?**

**Answer:**

| **Feature** | **Array** | **ArrayList** |
| --- | --- | --- |
| Size | Fixed | Dynamic |
| Data Types | Can store primitives & objects | Only objects |
| Performance | Faster | Slower (due to resizing, boxing/unboxing) |
| Utility methods | Limited (length) | Rich API (add, remove, contains) |

**Section 8: Methods (Functions in Java)**

**1. What is a method in Java?**

**Answer:**

* A **method** in Java is a block of code that performs a specific task.
* It improves **code reusability**, **readability**, and **modularity**.
* Methods are invoked (called) whenever we need to perform that specific task.

**Example:**

public class Example {

public static void greet() {

System.out.println("Hello, Welcome to Java!");

}

public static void main(String[] args) {

greet(); // method call

}

}

**2. What are the types of methods in Java?**

**Answer:**

1. **Predefined methods** – Already defined in Java libraries (e.g., length(), sqrt()).
2. **User-defined methods** – Created by the programmer.

**Example:**

public class Example {

public static void main(String[] args) {

// Predefined method

String s = "Hello";

System.out.println(s.length()); // length()

// User-defined method

greet();

}

static void greet() {

System.out.println("Hi User!");

}

}

**3. What is the difference between a method declaration and method definition?**

**Answer:**

* **Method Declaration**: Specifies method name, return type, and parameters (signature).
* **Method Definition**: Includes the method body (actual implementation).

**Example:**

// Declaration

int add(int a, int b);

// Definition

int add(int a, int b) {

return a + b;

}

**4. What is the syntax of a method in Java?**

**Answer:**

modifier returnType methodName(parameters) {

// method body

return value; // if returnType is not void

}

**Example:**

public int add(int a, int b) {

return a + b;

}

**5. What are parameters and arguments in Java methods?**

**Answer:**

* **Parameter**: Variable defined in the method signature.
* **Argument**: Actual value passed when calling the method.

**Example:**

public class Example {

static void greet(String name) { // parameter

System.out.println("Hello " + name);

}

public static void main(String[] args) {

greet("Alice"); // argument

}

}

**6. What is the difference between void and return type in methods?**

**Answer:**

* **void method**: Does not return any value.
* **Return type method**: Must return a value of the declared type.

**Example:**

void display() {

System.out.println("This is a void method.");

}

int add(int a, int b) {

return a + b;

}

**7. What is method overloading in Java?**

**Answer:**

* **Method Overloading** occurs when two or more methods have the same name but different parameter lists (different type or number of parameters).
* Achieved at **compile-time (compile-time polymorphism)**.

**Example:**

class Calculator {

int add(int a, int b) {

return a + b;

}

double add(double a, double b) {

return a + b;

}

}

**8. Can methods be overloaded by changing the return type only?**

**Answer:**

* **No.** Method overloading cannot be achieved by just changing the return type.
* The compiler must differentiate methods based on **parameter list** (not return type).

**9. What is method overriding in Java?**

**Answer:**

* **Method Overriding** occurs when a subclass provides its own implementation of a method that is already defined in its superclass.
* Achieved at **runtime (runtime polymorphism)**.
* The method must have the same **name, parameters, and return type**.

**Example:**

class Animal {

void sound() {

System.out.println("Animal makes a sound");

}

}

class Dog extends Animal {

@Override

void sound() {

System.out.println("Dog barks");

}

}

public class Main {

public static void main(String[] args) {

Animal a = new Dog();

a.sound(); // Runtime polymorphism

}

}

**10. What are static methods in Java?**

**Answer:**

* Declared using the static keyword.
* Can be called **without creating an object**.
* Can only access **static variables** and **static methods**.

**Example:**

public class Example {

static void greet() {

System.out.println("Hello from static method!");

}

public static void main(String[] args) {

greet(); // no object needed

}

}

**11. What are the rules of method overriding?**

**Answer:**

1. Must have the same name and parameter list.
2. Must have the same or covariant return type.
3. Cannot override static, final, or private methods.
4. Access level cannot be more restrictive than the overridden method.

**12. What are recursive methods in Java?**

**Answer:**

* A method that **calls itself** until a base condition is met.
* Used in problems like factorial, Fibonacci, etc.

**Example (factorial):**

int factorial(int n) {

if (n == 0) return 1; // base case

return n \* factorial(n - 1);

}

**13. What is the difference between call by value and call by reference in Java?**

**Answer:**

* **Call by Value**: Java passes a **copy of the variable value**. Changes inside the method do not affect the original variable.
* **Call by Reference**: Java does not support true call by reference, but objects are passed by reference internally (reference copy is passed).

**Example:**

void modify(int x) {

x = x + 10;

}

public static void main(String[] args) {

int a = 5;

modify(a);

System.out.println(a); // still 5

}

**14. What is the difference between main() method and user-defined methods?**

**Answer:**

* **main()**: Entry point of Java application, must be public static void main(String[] args).
* **User-defined methods**: Created by developers to perform specific tasks, invoked inside main().

**15. What are the advantages of using methods in Java?**

**Answer:**

* **Code Reusability** – Avoids duplicate code.
* **Modularity** – Breaks down complex programs.
* **Maintainability** – Easy to debug/update.
* **Abstraction** – Hides implementation details.

**Section 9: Object-Oriented Programming (OOP)**

**Part A: Basics of OOP**

**Q1. What is Object-Oriented Programming (OOP)? Why is it used in Java?**

**Answer:**

* **Object-Oriented Programming (OOP)** is a programming paradigm where software is designed using **objects** that represent real-world entities.
* Java is purely OOP (except primitives) and organizes code into **classes** and **objects**.
* Benefits:
  + **Reusability** via inheritance
  + **Maintainability** via encapsulation
  + **Flexibility** via polymorphism
  + **Security** via abstraction and access modifiers

**Q2. What are the main principles of OOP in Java?**

**Answer:**

1. **Encapsulation** – Wrapping data (fields) and methods into a single unit (class).
2. **Inheritance** – Reusing properties/behaviors from a parent class.
3. **Polymorphism** – Same method behaving differently (overloading/overriding).
4. **Abstraction** – Hiding implementation details and showing only functionality.

**Q3. What is a class and an object in Java? Give an example.**

**Answer:**

* **Class** → Blueprint/template for creating objects.
* **Object** → Instance of a class, representing a real-world entity.

**Example:**

class Car {

String brand;

int speed;

void drive() {

System.out.println(brand + " is driving at " + speed + " km/h");

}

}

public class Main {

public static void main(String[] args) {

Car car1 = new Car(); // object creation

car1.brand = "Tesla";

car1.speed = 120;

car1.drive();

}

}

**Part B: Encapsulation**

**Q4. What is encapsulation in Java?**

**Answer:**

* Encapsulation = Wrapping **fields (data)** and **methods** into a class.
* Achieved using:
  + **Private variables**
  + **Public getters & setters**

**Example:**

class BankAccount {

private double balance; // private data

public double getBalance() { // getter

return balance;

}

public void deposit(double amount) { // setter with validation

if(amount > 0) balance += amount;

}

}

**Q5. What are the advantages of encapsulation?**

**Answer:**

* Data hiding → Security
* Controlled access → via setters/getters
* Improved maintainability
* Loose coupling

**Part C: Inheritance**

**Q6. What is inheritance in Java?**

**Answer:**

* Inheritance allows one class (**child**) to acquire properties and behaviors of another class (**parent**).
* Achieved using extends keyword.

**Example:**

class Animal {

void eat() { System.out.println("Eating..."); }

}

class Dog extends Animal {

void bark() { System.out.println("Barking..."); }

}

public class Main {

public static void main(String[] args) {

Dog d = new Dog();

d.eat(); // inherited method

d.bark();

}

}

**Q7. What are the types of inheritance supported in Java?**

**Answer:**

1. **Single Inheritance** – One class inherits another.
2. **Multilevel Inheritance** – Child inherits parent, grandchild inherits child.
3. **Hierarchical Inheritance** – Multiple child classes inherit one parent.  
   *(Java does NOT support multiple inheritance with classes, only via interfaces).*

**Q8. What is the super keyword in Java?**

**Answer:**

* Refers to the **immediate parent class**.
* Uses:
  + Call parent class constructor
  + Access parent class method/variable

**Example:**

class Animal {

String type = "Animal";

Animal() { System.out.println("Animal created"); }

}

class Dog extends Animal {

Dog() {

super(); // calls parent constructor

System.out.println("Dog created");

}

}

**Part D: Polymorphism**

**Q9. What is polymorphism in Java?**

**Answer:**

* **Polymorphism = many forms.**
* Two types:
  + **Compile-time (Method Overloading)**
  + **Runtime (Method Overriding)**

**Q10. What is method overloading in Java?**

**Answer:**

* Same method name with different **parameters** (number/type).
* Happens at compile time.

**Example:**

class MathUtil {

int add(int a, int b) { return a + b; }

double add(double a, double b) { return a + b; }

}

**Q11. What is method overriding in Java?**

**Answer:**

* Child class provides a new implementation of a parent class method.
* Rules:
  + Same method name, parameters, return type
  + Cannot override private, static, final methods

**Example:**

class Animal {

void sound() { System.out.println("Animal sound"); }

}

class Dog extends Animal {

@Override

void sound() { System.out.println("Bark"); }

}

**Q12. What is dynamic method dispatch?**

**Answer:**

* Process where the **call to an overridden method is resolved at runtime**.
* Achieved via method overriding.

**Example:**

Animal a = new Dog(); // reference type: Animal, object type: Dog

a.sound(); // Bark (runtime decision)

**Part E: Abstraction**

**Q13. What is abstraction in Java?**

**Answer:**

* **Hiding implementation details** and showing only essential features.
* Achieved via:
  + **Abstract classes** (abstract keyword)
  + **Interfaces**

**Q14. What is an abstract class?**

**Answer:**

* Class declared with abstract keyword.
* Can have:
  + Abstract methods (without body)
  + Concrete methods (with body)

**Example:**

abstract class Shape {

abstract void draw(); // abstract method

void info() { System.out.println("Shape info"); }

}

class Circle extends Shape {

void draw() { System.out.println("Drawing Circle"); }

}

**Q15. What is an interface in Java?**

**Answer:**

* **Interface = contract** containing abstract methods (Java 8+ supports default & static methods).
* A class implements an interface using implements.
* Supports multiple inheritance.

**Example:**

interface Vehicle {

void start();

}

class Car implements Vehicle {

public void start() { System.out.println("Car started"); }

}

**Q16. Difference between abstract class and interface?**

**Answer:**

| **Feature** | **Abstract Class** | **Interface** |
| --- | --- | --- |
| Methods | Abstract + Concrete | Abstract (Java 8+ → default & static too) |
| Variables | Instance + static | Public static final only |
| Inheritance | Single only | Multiple possible |
| Use case | Partial abstraction | Full abstraction |

**Part F: Constructors**

**Q17. What is a constructor in Java?**

**Answer:**

* Special method that initializes objects.
* Same name as class, no return type.

**Types:**

1. **Default Constructor** – No arguments
2. **Parameterized Constructor** – Takes arguments
3. **Copy Constructor** – Copy data from another object (not built-in, must be coded manually)

**Q18. Can a constructor be overloaded in Java?**

**Answer:**

* Yes. Multiple constructors with different parameter lists.

**Example:**

class Student {

Student() { System.out.println("Default Constructor"); }

Student(String name) { System.out.println("Name: " + name); }

}

**Part G: this, static, and memory concepts**

**Q19. What is the this keyword in Java?**

**Answer:**

* Refers to the current object.
* Uses:
  + Differentiate instance variables from parameters
  + Call another constructor (this())
  + Pass current object

**Q20. What is the difference between static and instance variables/methods?**

**Answer:**

* **Static** → belongs to class, shared by all objects.
* **Instance** → belongs to each object.

**Example:**

class Student {

static String school = "ABC School";

String name;

}

**Q21. What is the difference between == and .equals() in objects?**

**Answer:**

* == → compares references (memory location).
* .equals() → compares values (if overridden, e.g., String class).

**Part H: Advanced OOP**

**Q22. What is multiple inheritance? Does Java support it?**

**Answer:**

* Multiple inheritance = one class inherits multiple classes.
* Java **does not support multiple inheritance with classes** to avoid ambiguity (diamond problem).
* Achieved via **interfaces**.

**Q23. What are inner classes in Java?**

**Answer:**

* A class defined inside another class.
* Types:
  + **Member Inner Class**
  + **Static Nested Class**
  + **Local Inner Class**
  + **Anonymous Inner Class**

**Q24. What is an immutable class in Java?**

**Answer:**

* Class whose objects cannot be modified once created.
* Example: String class.
* Custom immutable class → mark fields final + no setters + deep copy in getters.

**Section 10: Exception Handling in Java**

**1. What is Exception Handling in Java?**

**Answer:**

* Exception Handling in Java is a mechanism to handle runtime errors so that the flow of the program is not disrupted.
* It uses the **try-catch-finally-throw-throws** keywords.

**Example:**

public class ExceptionExample {

public static void main(String[] args) {

try {

int result = 10 / 0; // ArithmeticException

} catch (ArithmeticException e) {

System.out.println("Error: Division by zero is not allowed!");

} finally {

System.out.println("Finally block always executes.");

}

}

}

**2. What is the difference between Error and Exception in Java?**

**Answer:**

* **Error:** Represents serious problems that a program should not try to handle (e.g., OutOfMemoryError, StackOverflowError).
* **Exception:** Represents conditions that a program can handle (e.g., NullPointerException, IOException).

**3. What are the types of Exceptions in Java?**

**Answer:**

1. **Checked Exceptions** – Checked at compile time (e.g., IOException, SQLException).
2. **Unchecked Exceptions** – Occur at runtime, not checked at compile time (e.g., NullPointerException, ArithmeticException).
3. **Errors** – Serious issues not meant to be handled (e.g., OutOfMemoryError).

**4. Explain try, catch, finally, throw, and throws keywords.**

**Answer:**

* **try** → block that contains code which may throw an exception.
* **catch** → block that handles the exception.
* **finally** → block that always executes (cleanup code).
* **throw** → used to explicitly throw an exception.
* **throws** → used in method signature to declare exceptions a method may throw.

**Example:**

public void readFile() throws IOException {

throw new IOException("File not found!");

}

**5. What is the difference between throw and throws?**

**Answer:**

* **throw** → used to actually throw an exception inside a method.
* **throws** → used in method declaration to specify possible exceptions.

**Example:**

// Using throw

throw new ArithmeticException("Divide by zero");

// Using throws

public void divide() throws ArithmeticException { ... }

**6. Can we have multiple catch blocks in Java?**

**Answer:**

* Yes, multiple catch blocks can be used to handle different types of exceptions.
* The order must go from **specific to general** (child exceptions first).

**Example:**

try {

int arr[] = new int[5];

arr[10] = 50;

} catch (ArrayIndexOutOfBoundsException e) {

System.out.println("Array index error!");

} catch (Exception e) {

System.out.println("General exception caught.");

}

**7. Can we have multiple exceptions in one catch block?**

**Answer:**

* Yes, from Java 7 onwards, we can use **multi-catch** with | operator.

**Example:**

try {

int result = 10 / 0;

} catch (ArithmeticException | NullPointerException e) {

System.out.println("Caught exception: " + e);

}

**8. What happens if an exception is not handled?**

**Answer:**

* If not handled, the JVM terminates the program and prints the **stack trace**.

**9. What is the difference between final, finally, and finalize()?**

**Answer:**

* **final** → keyword used with classes, methods, and variables.
* **finally** → block that always executes in exception handling.
* **finalize()** → method called by Garbage Collector before destroying an object.

**10. What is the difference between Checked and Unchecked exceptions?**

**Answer:**

| **Aspect** | **Checked Exceptions** | **Unchecked Exceptions** |
| --- | --- | --- |
| Compile-time check | Checked at compile time | Not checked at compile time |
| Example | IOException, SQLException | NullPointerException, RuntimeException |
| Handling | Must be handled using try-catch or throws | Handling optional |

**11. Can we catch multiple exceptions with one catch block?**

**Answer:**

* Yes, using **multi-catch** syntax introduced in Java 7.

**12. What is the role of throw keyword?**

**Answer:**

* The throw keyword is used to explicitly throw an exception inside a method.

**Example:**

throw new IllegalArgumentException("Invalid input!");

**13. What is the difference between System.exit(0) and finally block execution?**

**Answer:**

* Normally, finally always executes.
* But if System.exit(0) is called, the program terminates immediately and finally will not execute.

**14. Can we write try without catch?**

**Answer:**

* Yes, we can use try with only finally (no catch).
* Useful when you always want cleanup code to run.

**Example:**

try {

int data = 50 / 0;

} finally {

System.out.println("Cleanup executed!");

}

**15. Can we write multiple finally blocks?**

**Answer:**

* No, only one finally block is allowed per try.

**16. What is the difference between throw new Exception() and e.printStackTrace()?**

**Answer:**

* **throw new Exception()** → actually throws a new exception.
* **e.printStackTrace()** → prints the stack trace but does not throw a new exception.

**17. Can constructors throw exceptions?**

**Answer:**

* Yes, constructors can throw both checked and unchecked exceptions.

**Example:**

class Test {

Test() throws IOException {

throw new IOException("Constructor exception");

}

}

**18. What are custom exceptions in Java?**

**Answer:**

* We can create our own exception class by extending Exception (checked) or RuntimeException (unchecked).

**Example:**

class MyException extends Exception {

public MyException(String message) {

super(message);

}

}

**19. What is Exception Propagation?**

**Answer:**

* If an exception is not handled in a method, it is **propagated** (passed) to the caller method until it is handled.
* Runtime exceptions are propagated automatically, but checked exceptions must be declared with throws.

**20. Best Practices for Exception Handling?**

**Answer:**

* Don’t use exceptions for flow control.
* Catch only specific exceptions.
* Always clean up resources in finally or use **try-with-resources**.
* Create custom exceptions only when necessary.

**Section 11: Multithreading in Java**

**1. What is multithreading in Java?**

**Answer:**  
Multithreading is a process of executing multiple threads simultaneously within a single program.

* **Thread:** A lightweight sub-process, the smallest unit of execution.
* Multithreading allows concurrent execution, which improves performance, especially on multicore CPUs.

**2. Difference between process and thread?**

**Answer:**

| **Feature** | **Process** | **Thread** |
| --- | --- | --- |
| Definition | Independent execution unit | Smallest execution unit within process |
| Memory | Has its own memory space | Shares memory of the process |
| Context switch | Expensive (more resources) | Cheaper, lightweight |
| Dependency | Independent | Dependent on process |

**3. How do you create a thread in Java?**

**Answer:**  
Two ways:

1. **By extending Thread class**

class MyThread extends Thread {

public void run() {

System.out.println("Thread running...");

}

}

public class Test {

public static void main(String[] args) {

MyThread t1 = new MyThread();

t1.start();

}

}

1. **By implementing Runnable interface**

class MyRunnable implements Runnable {

public void run() {

System.out.println("Thread running...");

}

}

public class Test {

public static void main(String[] args) {

Thread t1 = new Thread(new MyRunnable());

t1.start();

}

}

**4. Difference between start() and run() method in Java threads?**

**Answer:**

* run(): Executes as a normal method call, no new thread is created.
* start(): Creates a new thread and internally calls run().

**5. What is thread lifecycle in Java?**

**Answer:**

Thread states:

1. **New** – Thread object created but not started.
2. **Runnable** – After start() is called, waiting for CPU scheduling.
3. **Running** – When CPU executes the thread.
4. **Waiting/Timed Waiting** – Thread waits indefinitely or for a specified time.
5. **Terminated** – Thread has finished execution.

**6. What is thread priority in Java?**

**Answer:**

* Each thread has a priority (1–10).
* MIN\_PRIORITY = 1, NORM\_PRIORITY = 5 (default), MAX\_PRIORITY = 10.
* Higher-priority threads get preference in CPU scheduling, but it is **not guaranteed**.

**7. What is synchronization in Java multithreading?**

**Answer:**

* Synchronization ensures that only one thread accesses a shared resource at a time.
* Prevents **race conditions**.

Example:

class Counter {

int count = 0;

public synchronized void increment() {

count++;

}

}

**8. What are race conditions in multithreading?**

**Answer:**  
When two or more threads access a shared resource concurrently and the final output depends on the order of execution, it leads to inconsistent results.  
**Solution:** Use synchronization, locks, or concurrent collections.

**9. Difference between synchronized method and synchronized block?**

**Answer:**

* **Synchronized Method:** Locks the entire method → less efficient.
* **Synchronized Block:** Locks only a part of code → more efficient.

**10. What is deadlock in Java multithreading?**

**Answer:**  
Deadlock occurs when two or more threads are waiting indefinitely for each other to release locks.  
Example:

* Thread1 locks A, waits for B.
* Thread2 locks B, waits for A.  
  → Both are stuck forever.

**11. How can you prevent deadlock?**

**Answer:**

* Acquire locks in a consistent order.
* Use tryLock() with timeout (from ReentrantLock).
* Minimize synchronized code.

**12. What is the difference between concurrency and parallelism?**

**Answer:**

* **Concurrency:** Multiple tasks make progress at the same time (may not execute simultaneously).
* **Parallelism:** Tasks execute literally at the same time (requires multicore CPU).

**13. Difference between wait(), notify(), and notifyAll()?**

**Answer:**

* wait(): Causes current thread to release the lock and wait.
* notify(): Wakes up one waiting thread.
* notifyAll(): Wakes up all waiting threads.

All three must be called inside a synchronized block.

**14. What is volatile keyword in Java multithreading?**

**Answer:**

* Declares a variable as being stored in **main memory**.
* Ensures **visibility**: All threads read the latest updated value.
* Does not guarantee atomicity (use synchronized or AtomicInteger for atomic operations).

**15. What are join() and sleep() methods in threads?**

**Answer:**

* sleep(ms): Makes the thread pause for given milliseconds.
* join(): Waits for another thread to finish execution.

**16. What is the difference between Callable and Runnable?**

**Answer:**

* Runnable: Does not return result, cannot throw checked exception.
* Callable: Returns a result (Future object), can throw exceptions.

**17. What is Executor Framework in Java?**

**Answer:**

* Introduced in Java 5 (java.util.concurrent).
* Provides a thread pool to manage threads efficiently.

Example:

ExecutorService executor = Executors.newFixedThreadPool(3);

executor.execute(new MyRunnable());

executor.shutdown();

**18. What are Concurrent Collections in Java?**

**Answer:**  
Thread-safe collections that handle synchronization internally.  
Examples:

* ConcurrentHashMap
* CopyOnWriteArrayList
* BlockingQueue

**19. What is a daemon thread in Java?**

**Answer:**

* A background thread that runs in service of other threads.
* Example: Garbage Collector is a daemon thread.
* Created using setDaemon(true).

**20. What are some best practices for multithreading in Java?**

**Answer:**

* Minimize shared mutable data.
* Prefer concurrent collections over synchronized blocks.
* Use higher-level APIs (ExecutorService, ForkJoinPool).
* Always handle deadlocks, race conditions, and thread safety.

**Section 12: Collection Framework in Java**

**1. What is the Java Collection Framework (JCF)?**

**Answer:**  
The **Java Collection Framework (JCF)** is a set of **interfaces, classes, and algorithms** that provide a standard way to store, manipulate, and retrieve groups of objects.  
It provides **ready-to-use data structures** like **List, Set, Queue, Map**, and utilities in **Collections class**.

Example:

import java.util.\*;

public class Example {

public static void main(String[] args) {

List<String> list = new ArrayList<>();

list.add("Apple");

list.add("Banana");

System.out.println(list); // [Apple, Banana]

}

}

**2. What are the main interfaces in the Collection Framework?**

**Answer:**  
The key interfaces are:

* **Collection (root interface)**
  + **List** (ArrayList, LinkedList, Vector, Stack)
  + **Set** (HashSet, LinkedHashSet, TreeSet)
  + **Queue** (PriorityQueue, ArrayDeque)
* **Map (separate hierarchy)**
  + HashMap, LinkedHashMap, TreeMap, Hashtable, ConcurrentHashMap

**3. Difference between List, Set, and Map?**

**Answer:**

| **Feature** | **List** | **Set** | **Map** |
| --- | --- | --- | --- |
| Duplicates | Allowed | Not Allowed | Keys not allowed, values allowed |
| Order | Maintains insertion order | May/may not maintain order | Key-value pairs, order depends on type |
| Examples | ArrayList, LinkedList, Vector | HashSet, LinkedHashSet, TreeSet | HashMap, TreeMap, LinkedHashMap |

**4. Difference between ArrayList and LinkedList?**

**Answer:**

| **Feature** | **ArrayList** | **LinkedList** |
| --- | --- | --- |
| Data Structure | Dynamic array | Doubly linked list |
| Access (get) | O(1) fast random access | O(n) slower |
| Insert/Delete | Slow (shifting elements) | Fast (just re-linking nodes) |
| Use Case | Frequent read operations | Frequent insert/delete ops |

Example:

List<Integer> arr = new ArrayList<>();

arr.add(10);

arr.add(20);

List<Integer> linked = new LinkedList<>();

linked.add(10);

linked.add(20);

**5. What is the difference between HashSet and TreeSet?**

**Answer:**

| **Feature** | **HashSet** | **TreeSet** |
| --- | --- | --- |
| Ordering | No order maintained | Sorted in ascending order |
| Null Elements | Allows one null | Does not allow null |
| Performance | Faster (O(1)) | Slower (O(log n)) |

**6. Explain HashMap vs Hashtable vs ConcurrentHashMap.**

**Answer:**

| **Feature** | **HashMap** | **Hashtable** | **ConcurrentHashMap** |
| --- | --- | --- | --- |
| Thread-Safe | No | Yes (synchronized) | Yes (concurrent lock partitioning) |
| Null Keys/Values | 1 null key, multiple null values | No null keys/values allowed | 1 null key, multiple null values |
| Performance | Faster (non-synchronized) | Slower | High performance in multi-threading |

**7. What is the difference between Comparable and Comparator?**

**Answer:**

* **Comparable** → Natural ordering using compareTo() (only one sort sequence).
* **Comparator** → Custom ordering using compare() (multiple sorting logics possible).

Example:

class Student implements Comparable<Student> {

int age;

Student(int age) { this.age = age; }

public int compareTo(Student s) {

return this.age - s.age; // Natural order by age

}

}

class AgeComparator implements Comparator<Student> {

public int compare(Student s1, Student s2) {

return s2.age - s1.age; // Custom order descending

}

}

**8. Difference between Iterator and ListIterator?**

**Answer:**

| **Feature** | **Iterator** | **ListIterator** |
| --- | --- | --- |
| Direction | Forward only | Forward + Backward |
| Modify Elements | Only remove | Add, remove, replace |
| Applicable On | All collections | Only List |

**9. Fail-Fast vs Fail-Safe Iterators?**

**Answer:**

* **Fail-Fast** (e.g., ArrayList, HashMap iterators) → Throws ConcurrentModificationException if modified during iteration.
* **Fail-Safe** (e.g., ConcurrentHashMap, CopyOnWriteArrayList) → Works on a **clone** of the collection, no exception.

**10. What is the difference between Arrays and Collections?**

**Answer:**

| **Feature** | **Arrays** | **Collections** |
| --- | --- | --- |
| Size | Fixed | Dynamic resizing |
| Data Type | Can store primitives/objects | Only objects |
| Performance | Faster (less overhead) | More flexible, rich APIs |

**11. What is the Collections Utility Class?**

**Answer:**  
The **Collections** class (java.util.Collections) contains static utility methods like:

* sort() → Sorts a list
* reverse() → Reverses order
* max(), min() → Finds max/min element
* synchronizedList() → Makes a list thread-safe

Example:

import java.util.\*;

public class CollectionsExample {

public static void main(String[] args) {

List<Integer> nums = Arrays.asList(5, 1, 8, 2);

Collections.sort(nums);

System.out.println(nums); // [1, 2, 5, 8]

}

}

**12. How does HashMap work internally?**

**Answer:**

* HashMap stores elements in **buckets (array of nodes)**.
* Each key is hashed using hashCode().
* Index = (hashCode & (n-1)).
* If collision occurs → elements are stored in a **linked list / balanced tree (Java 8+)**.
* Performance → O(1) average, O(log n) worst case.

**13. Difference between ConcurrentHashMap and SynchronizedMap?**

**Answer:**

* **SynchronizedMap** → Locks the **entire map** → lower performance.
* **ConcurrentHashMap** → Uses **segment-level locking** (bucket-based), better concurrency.

**14. How to remove duplicates from a List in Java?**

**Answer:**  
Using **Set**:

List<Integer> list = Arrays.asList(1, 2, 2, 3, 4, 4);

Set<Integer> set = new HashSet<>(list);

System.out.println(set); // [1, 2, 3, 4]

Or **Streams (Java 8+):**

list.stream().distinct().forEach(System.out::println);

**Section 13: Packages in Java**

**Q1. What is a package in Java? Why is it used?**  
A **package** in Java is a way of grouping related classes, interfaces, and sub-packages together. It acts like a folder in a file system.  
Packages are used to:

* **Organize code**: Makes large projects manageable.
* **Avoid name conflicts**: Classes with the same name can exist in different packages.
* **Access protection**: Packages provide access modifiers (public, protected, default, private) at a package level.
* **Reusability**: Classes in a package can be reused in other programs.

**Q2. What are the types of packages in Java?**  
There are two types of packages:

1. **Built-in packages (predefined):** Provided by Java API (e.g., java.util, java.sql, java.io, etc.).
2. **User-defined packages:** Created by developers to organize their own classes.

**Q3. How do you create a package in Java?**  
We use the **package** keyword.

**Example:**

package mypack; // defining package

public class MyClass {

public void display() {

System.out.println("Hello from MyClass in mypack!");

}

}

**Q4. How do you use (import) a package in Java?**  
Packages are imported using the **import** keyword.

* **Import single class:**

import mypack.MyClass;

* **Import entire package:**

import mypack.\*;

**Q5. What is the difference between import mypack.\* and import mypack.MyClass?**

* import mypack.\* → Imports all classes/interfaces of the package (but not sub-packages).
* import mypack.MyClass → Imports only the specified class.

**Q6. What are some commonly used built-in packages in Java?**

* java.lang → Contains fundamental classes (String, Math, Object, etc.)
* java.util → Data structures & collections (ArrayList, HashMap, etc.)
* java.io → Input/Output operations
* java.sql → Database connectivity (JDBC)
* java.time → Date and time API

**Q7. What is the default package in Java?**  
If no package is specified, the class is placed in the **default package**.

* Default package has **no name**.
* Classes in the default package **cannot be imported** into other packages.

**Q8. What is the difference between a package and an import statement?**

* package keyword → Defines a package for a class.
* import keyword → Brings classes/packages into scope for usage.

**Q9. Can you access a class from another package without using import?**  
Yes, by using the **fully qualified name (FQN)**.

**Example:**

public class Test {

public static void main(String[] args) {

mypack.MyClass obj = new mypack.MyClass();

obj.display();

}

}

**Q10. Can sub-packages be created in Java?**  
Yes. Java supports **hierarchical packages**.

**Example:**

package mypack.subpack;

Here, subpack is a subpackage inside mypack.

**Q11. What is the difference between a package and a module in Java (Java 9+)?**

* **Package:** Groups related classes/interfaces.
* **Module:** A higher-level grouping that contains packages and provides stronger encapsulation (introduced in Java 9 with the module-info.java file).

**Q12. What is the difference between a package and a classpath?**

* **Package:** Logical grouping of classes.
* **Classpath:** The location (path) where the Java runtime looks for .class files and packages.

**Q13. How do access modifiers work with packages?**

* **public:** Accessible everywhere.
* **protected:** Accessible within the package and in subclasses (even outside package).
* **default (no modifier):** Accessible only within the same package.
* **private:** Accessible only within the same class.

**Q14. Can interfaces and enums be placed inside packages?**  
Yes. Packages can contain **classes, interfaces, enums, and sub-packages**.

**Q15. What are the advantages of using packages in Java?**

1. Organized code structure.
2. Avoids class name conflicts.
3. Controlled access via access modifiers.
4. Promotes reusability.
5. Easier maintenance in large applications.

**Section 14: Generics in Java**

**Q1. What are Generics in Java?**  
Generics in Java allow you to create **classes, interfaces, and methods with type parameters**. This enables **type safety**, **code reusability**, and **compile-time checking**, reducing the need for typecasting.

Example:

// Without Generics

List list = new ArrayList();

list.add("Hello");

String s = (String) list.get(0); // Typecasting required

// With Generics

List<String> list2 = new ArrayList<>();

list2.add("Hello");

String s2 = list2.get(0); // No typecasting

**Q2. Why do we use Generics in Java?**

**Answer:**

1. **Type Safety** – Ensures only specific type of objects are added.
2. **Eliminates Type Casting** – No need to cast objects.
3. **Code Reusability** – Same code works with different data types.
4. **Compile-time Checking** – Errors are caught early.

**Q3. Can you explain Generic Classes with an example?**

**Answer:**  
A **generic class** is a class that can work with different data types.

class Box<T> { // T is a type parameter

private T value;

public void setValue(T value) {

this.value = value;

}

public T getValue() {

return value;

}

}

public class Main {

public static void main(String[] args) {

Box<String> strBox = new Box<>();

strBox.setValue("Hello");

System.out.println(strBox.getValue());

Box<Integer> intBox = new Box<>();

intBox.setValue(100);

System.out.println(intBox.getValue());

}

}

**Q4. What are Generic Methods in Java?**  
Generic methods allow **type parameters** for methods, making them flexible for different data types.

Example:

class Util {

public static <T> void printArray(T[] array) {

for (T element : array) {

System.out.print(element + " ");

}

System.out.println();

}

}

public class Main {

public static void main(String[] args) {

Integer[] intArr = {1, 2, 3};

String[] strArr = {"A", "B", "C"};

Util.printArray(intArr); // Works for Integer

Util.printArray(strArr); // Works for String

}

}

**Q5. What are Bounded Type Parameters in Generics?**  
Sometimes we want to **restrict the types** used in generics using the extends keyword.

Example:

class Calculator<T extends Number> {

public double add(T a, T b) {

return a.doubleValue() + b.doubleValue();

}

}

public class Main {

public static void main(String[] args) {

Calculator<Integer> calc1 = new Calculator<>();

System.out.println(calc1.add(10, 20)); // 30.0

Calculator<Double> calc2 = new Calculator<>();

System.out.println(calc2.add(10.5, 20.5)); // 31.0

}

}

Here, only **subclasses of Number** (Integer, Double, etc.) are allowed.

**Q6. What are Wildcards in Generics?**  
Wildcards (?) are used when we don’t know the exact type.

1. **Unbounded Wildcard** – <?> → accepts any type.

public void printList(List<?> list) {

for (Object obj : list) {

System.out.println(obj);

}

}

1. **Upper Bounded Wildcard** – <? extends Number> → accepts Number or its subclasses.

public double sum(List<? extends Number> list) {

double total = 0.0;

for (Number n : list) {

total += n.doubleValue();

}

return total;

}

1. **Lower Bounded Wildcard** – <? super Integer> → accepts Integer or its superclasses.

public void addNumbers(List<? super Integer> list) {

list.add(10);

list.add(20);

}

**Q7. What is the difference between <? extends T> and <? super T>?**

* <? extends T> → **Upper bound**, can read items safely but can’t add (except null).
* <? super T> → **Lower bound**, can safely add items but when reading you only get Object.

**Q8. Can Generics work with primitive data types in Java?**  
No, Generics only work with **objects**.  
Primitives (int, double, etc.) must be used with their **Wrapper classes** (Integer, Double).

Example:

List<int> list = new ArrayList<>(); // ❌ Not allowed

List<Integer> list = new ArrayList<>(); // ✅ Allowed

**Q9. What is Type Erasure in Generics?**  
Java implements generics using **Type Erasure** at runtime. This means the type parameter information (<T>) is removed during compilation and replaced with **Object or their bounds**.

Example:

List<String> list1 = new ArrayList<>();

List<Integer> list2 = new ArrayList<>();

System.out.println(list1.getClass() == list2.getClass()); // true

Both lists are treated as **ArrayList** at runtime.

**Q10. Can we overload methods with different generic parameters?**  
Yes, we can overload, but the compiler may not differentiate them after **type erasure**.

Example:

class Test {

// Overloading works if signatures are different

public <T> void display(T data) {

System.out.println(data);

}

public <T extends Number> void displayNumber(T num) {

System.out.println(num);

}

}

**Q11. What are some real-life use cases of Generics in Java?**

1. **Collections Framework** (List<T>, Set<T>, Map<K,V>).
2. **Utility methods** (Collections.sort(List<T>)).
3. **Frameworks** – Selenium uses Generics in List<WebElement>.
4. **Reusable classes** like DAO layer in automation frameworks.