Core & Advanced Java Notes

✅ 1. OOP Concepts

Java is based on Object-Oriented Programming.
The main pillars are:

- **Encapsulation**: Bind data with code (class).
- **Inheritance**: Reuse code from parent class.
- **Polymorphism**: Same method, different behaviors.
- **Abstraction**: Hide internal details, show only essential features.

✅ 2. Data Types and Variables

Java supports **primitive** (int, char, boolean, etc.) and **non-primitive** (String, arrays) data types.
Variables are containers to store data.
Java is strongly typed — you must declare the type.
Example: int age = 25;
Use **type casting** to convert between types.

✅ 3. Control Flow Statements

Control statements manage program flow.

- **if / if-else**: Conditional execution.
- **switch**: Multi-branch control.
- **for / while / do-while**: Looping structures.
- **break**: Exit loop early, **continue**: skip current iteration.

✅ 4. Arrays and Strings

An **array** is a fixed-size collection of elements of the same type.
Syntax: int[] arr = new int[5];

A **String** is a sequence of characters and is immutable.
Use methods like .length(), .charAt(), .equals(), etc.
Use .equals() to compare values, not ==.

✅ 5. Constructors

Constructors are special methods to initialize objects.
They have the same name as the class and no return type.
You can overload constructors with different parameters.
Java provides a **default constructor** if none is written.
Called automatically when you create an object.

✅ 6. Inheritance

Inheritance allows one class to reuse another class's code using extends.
The **child class** inherits fields and methods of the **parent class**.
Helps with code reusability and method overriding.
Java supports **single**, **multilevel**, and **hierarchical** inheritance.
Use super to access parent methods or constructors.

✅ 7. Polymorphism

Polymorphism means “many forms.”

- **Compile-time (overloading)**: Same method name, different parameters.
- **Runtime (overriding)**: Subclass modifies parent method.
  Enhances flexibility and readability.
  Helps in dynamic method dispatch (object-based behavior).

✅ 8. Abstraction

Abstraction hides complex internal logic and shows only important features.
Achieved using **abstract classes** and **interfaces**.
An abstract class may contain both abstract and regular methods.
Interfaces provide full abstraction (Java 7 and earlier).
Encourages clean and maintainable code.

✅ 9. Encapsulation

Encapsulation binds data and code into a single unit (class).
Use **private** access for fields and **public getters/setters** for access.
Protects data from unauthorized access.
Improves modularity and maintainability.
Supports data hiding and secure programming.

✅ 10. Exception Handling

Java uses exceptions to handle runtime errors gracefully.
Keywords: try, catch, finally, throw, throws.

- **Checked exceptions**: handled at compile-time.
- **Unchecked exceptions**: occur at runtime.
  Avoids program crashes and allows error recovery.

✅ 11. Multithreading

Java supports multithreading for concurrent execution.
Create threads using Thread class or Runnable interface.
Thread methods: start(), run(), sleep(), join().
Use **synchronization** to avoid data conflicts.
Improves performance in multi-core systems.

✅ 12. Java Collections Framework

Collections are used to store and manipulate groups of data.
Includes interfaces like List, Set, Map, and classes like ArrayList, HashSet, HashMap.
Collections replace arrays for dynamic data handling.
Supports searching, sorting, inserting, and deleting data efficiently.
All classes are in java.util package.

✅ 13. Wrapper Classes

Wrapper classes convert primitives to objects: int → Integer, char → Character, etc.
Useful in collections that require objects.
Supports **autoboxing** (primitive to object) and **unboxing** (object to primitive).
Each primitive has a corresponding wrapper.
Example: Integer num = 10;

✅ 14. File Handling (I/O)

Java uses classes like File, FileReader, BufferedReader for file input.
Use FileWriter, BufferedWriter for writing.
I/O classes are in java.io package.
Always handle file exceptions with try-catch.
Supports reading text, binary files, and streams.

✅ 15. Java 8 Features

Java 8 introduced modern programming features:

- **Lambda expressions**: Anonymous methods
- **Stream API**: Functional-style data processing

- **Method references**: Shorter syntax to call methods
- **Default methods**: Method body in interfaces
  Boosts code readability and conciseness.

✅ 16. Access Modifiers

Java uses access modifiers to control visibility.

- private: within class
- (default): within package
- protected: within package and subclasses
- public: everywhere
  Helps with data hiding and design security.

✅ 17. Static Keyword

static belongs to the class, not the object.
Used for variables, methods, and blocks.
Accessed using ClassName.staticMember.
Useful for utility methods and constants.
Reduces memory usage when shared across instances.

✅ 18. final Keyword

Used to declare constants, prevent method overriding or inheritance.

- final variable: value cannot change
- final method: cannot be overridden
- final class: cannot be extended
  Improves security and design stability.

✅ 19. this and super Keyword

- this: Refers to the current object. Used to avoid variable name conflicts.
- super: Refers to the parent class. Used to access parent members.
  Helpful in constructor chaining and method overriding.

✅ 20. Packages and Imports

Packages are used to group related classes.
Use package to define, and import to access them.
Helps in avoiding class name conflicts.
Java has built-in packages like java.util, java.io, etc.
You can create your own packages for better organization.

21. Interfaces vs Abstract Classes

Both are used to achieve abstraction in Java.
Interface: All methods are abstract by default (till Java 7).
Abstract class: Can have both abstract and non-abstract methods.
A class can **implement multiple interfaces** but can **extend only one abstract class**.
Use interfaces when multiple inheritance is needed.

✅ 22. Method Overloading and Overriding

- **Overloading**: Same method name, different parameters (compile-time polymorphism).
- **Overriding**: Subclass changes the behavior of superclass method (runtime polymorphism).
  Overloading happens in the **same class**, overriding in **child-parent relationship**.
  Java uses @Override annotation for clarity.

✅ 23. Anonymous Inner Class

An inner class without a name, defined and instantiated at the same time.
Mostly used in **event handling** and **functional interfaces**.
Declared using:

```
Runnable r = new Runnable() { public void run() { ... } };
```

✅ 24. Enum in Java

Enums are special classes to define constants.
They are more powerful than final static variables.
You can define methods and constructors in enums.
Example:

```
enum Day { MON, TUE, WED }
```

✅ 25. Varargs (Variable Arguments)

Allows a method to accept **0 or more** arguments.
Declared with ... syntax:

```
void print(String... names) { }
```

Internally treated as an array. Only **one varargs** allowed per method.

✅ 26. Serialization and Deserialization

Serialization converts an object to a byte stream to save to a file or send over a network.
Use ObjectOutputStream for serialization, ObjectInputStream for

deserialization.
Class must implement Serializable interface.
transient keyword is used to skip fields during serialization.

✅ 27. Singleton Design Pattern

Restricts the instantiation of a class to **only one object**.
Used for logging, configuration classes, etc.
Typical implementation:

```
private static Singleton instance;
public static Singleton getInstance() { ... }
```

✅ 28. Immutable Class

An immutable class's state (data) cannot be changed once created.
String is the best example.
To make a class immutable:

- Make fields final and private
- Don't provide setters
- Return copies of mutable objects

✅ 29. Java Annotations

Metadata that provides information to the compiler or runtime.
Examples: @Override, @Deprecated, @SuppressWarnings
Used heavily in frameworks like Spring, Hibernate.
You can create your own custom annotations.

✅ 30. Functional Interfaces

An interface with **only one abstract method**.
Used with lambda expressions and method references.
Examples: Runnable, Comparable, Predicate<T>.
Declared with @FunctionalInterface annotation.

✅ 31. Lambda Expressions

Introduced in Java 8 to make code concise.
Syntax: (a, b) -> a + b
Used to implement functional interfaces.
Replaces anonymous inner classes in many cases.

✅ 32. Method References

A shorter way to refer to a method.
Syntax: ClassName::methodName
Example: System.out::println
Used with streams and lambda expressions.

✅ 33. Streams API

Processes collections in a **functional and pipeline** style.
Supports operations like filter(), map(), collect(), forEach()
Introduced in Java 8, great for working with large data sets.

✅ 34. Default and Static Methods in Interfaces

From Java 8 onwards:

- **default**: Methods with body inside interface.

- **static**: Belong to the interface, not instance.
  Helps in adding new methods to interfaces without breaking existing code.

✅ 35. JVM Architecture

JVM components:

- **Class Loader**
- **Memory Area (Heap, Stack, etc.)**
- **Execution Engine**
- **Garbage Collector**
  Responsible for executing Java bytecode on any machine.

✅ 36. Garbage Collection

Java handles memory automatically using the **Garbage Collector (GC)**.
It removes unreferenced objects to free up memory.
You can request GC using System.gc(), but it's not guaranteed.
Helps avoid memory leaks and crashes.

✅ 37. JIT Compiler

Just-In-Time (JIT) compiler is part of JVM.
It compiles bytecode into native machine code at runtime.
Improves performance by reducing interpretation overhead.

✅ 38. ClassLoader in Java

Loads classes into memory during runtime.
Types: **Bootstrap**, **Extension**, **Application** ClassLoader.
Enables dynamic loading of classes.

✅ 39. Transient Keyword

Used to prevent fields from being serialized.
Transient variables are skipped during object serialization.
Useful for sensitive data like passwords.

✅ 40. Volatile Keyword

Tells JVM that a variable may be changed by different threads.
Ensures visibility of changes to variables across threads.
Used in multithreaded environments.

41. Reflection API

The Reflection API allows you to inspect and modify classes, methods, and fields at runtime.
Used in frameworks (like Spring, Hibernate) for dependency injection and object inspection.
Example:

```
Class<?> c = Class.forName("MyClass");
Method[] methods = c.getDeclaredMethods();
```

✅ 42. Java Memory Model (JMM)

Defines how threads interact through memory and what behaviors are allowed in multithreading.
Ensures **visibility, ordering, and atomicity** of operations.

Important for understanding volatile, synchronized, and **thread safety**.

✅ 43. Thread Lifecycle

States: New → Runnable → Running → Blocked/Waiting → Terminated.
Each thread goes through these states managed by the JVM.
Understanding lifecycle helps debug performance and concurrency issues.

✅ 44. Thread Safety and Synchronization

Ensures that shared data is accessed by only one thread at a time.
Use synchronized, ReentrantLock, and atomic classes.
Avoids **race conditions** and inconsistent data.

✅ 45. Executor Framework

Provides thread pool management (ExecutorService) for better multithreading.
Avoids manually creating threads.
Supports Callable, Future, and task scheduling.

✅ 46. Java Generics

Allows you to write **type-safe**, reusable code.
Example: List<String> instead of List.
Supports compile-time checks and eliminates type casting.

✅ 47. Wildcard in Generics (?)

Used when the exact type is unknown.

- <? extends T>: accepts T or its subclasses.
- <? super T>: accepts T or its superclasses.
  Enhances flexibility in generic methods/classes.

✅ 48. Java Modules (Java 9+)

Modularizes large Java applications.
Introduced module-info.java for defining dependencies.
Improves encapsulation and reduces startup time.

✅ 49. Optional Class (Java 8)

A container for possibly-null values to avoid NullPointerException.
Methods: isPresent(), ifPresent(), orElse(), map().
Example:

Optional<String> name = Optional.ofNullable(null);

✅ 50. Comparable vs Comparator

Both are used to sort objects.

- Comparable: implemented in the class itself (compareTo()).
- Comparator: external custom sorting (compare()).
  Allows sorting by multiple fields dynamically.

✅ 51. Annotations Processing (APT)

Used to process custom annotations at compile time.
Helps generate code, configuration files, etc.
Frameworks like Lombok and Dagger use it heavily.

✅ 52. Java Records (Java 14+)

A compact syntax for immutable data classes.
Auto-generates constructors, getters, toString(), equals(), hashCode().
Example:

```
record Person(String name, int age) { }
```

✅ 53. Pattern Matching (Java 16+)

Simplifies instanceof checks and type casting.
Example:

```
if (obj instanceof String s) {
    System.out.println(s.length());
}
```

✅ 54. Sealed Classes (Java 17)

Restricts which classes can extend or implement a class/interface.
Improves control over class hierarchies.
Example:

```
sealed class Shape permits Circle, Square { }
```

✅ 55. Memory Leaks in Java

Memory that is no longer needed but not collected by GC.
Common in long-lived collections, listeners, static fields.
Use profiling tools like **VisualVM** or **JProfiler** to detect leaks.

✅ 56. Class.forName vs newInstance

- Class.forName() loads class at runtime.
- .newInstance() creates object using default constructor.
  Useful for dynamic loading in frameworks and JDBC drivers.

✅ 57. Java WeakReference & SoftReference

Used in caching and memory-sensitive applications.

- **WeakReference**: collected eagerly
- **SoftReference**: collected only when memory is low
  Helps in implementing custom memory management strategies.

✅ 58. Proxy Classes (Dynamic Proxy)

Creates objects at runtime that implement interfaces.
Used in **AOP (Aspect-Oriented Programming)**, like Spring.
Implemented via java.lang.reflect.Proxy.

✅ 59. JDBC Architecture

Used to connect Java to relational databases.
Consists of DriverManager, Connection, Statement, ResultSet.
Supports prepared statements and batch processing.

✅ 60. Java Native Interface (JNI)

Allows Java to call C/C++ native code.
Used in performance-sensitive operations and legacy systems.
Requires writing native libraries using javah and .dll or .so files.