Java: Basic to Advance

1. Introduction to Java, JVM, JDK, JRE

Theory:

- **Java**: Platform-independent, object-oriented programming language that follows "write once, run anywhere" principle
- **JVM** (Java Virtual Machine): Runtime environment that executes Java bytecode and provides platform independence
- **JRE** (Java Runtime Environment): Complete environment to run Java applications (JVM + core libraries)
- **JDK** (Java Development Kit): Complete development kit containing JRE + development tools (compiler, debugger, etc.)

Key Points:

java

```java
// Compilation process: .java → javac → .class → JVM → Machine Code
// Relationship: JDK ⊃ JRE ⊃ JVM
```

---

2. Data Types, Variables, Operators

Theory:

Java has two categories of data types: **Primitive** (stored in stack, hold actual values) and **Reference** (stored in heap, hold memory addresses). Variables are containers that store data values. Operators are symbols that perform operations on operands.

Code:

java

```java
// Primitive Data Types
byte b = 127;        // 8-bit (-128 to 127)
short s = 32000;     // 16-bit (-32,768 to 32,767)
int i = 100;         // 32-bit (-2^31 to 2^31-1)
long l = 100L;       // 64-bit (-2^63 to 2^63-1)
float f = 3.14f;     // 32-bit floating point
double d = 3.14159;  // 64-bit floating point
char c = 'A';        // 16-bit Unicode character
boolean flag = true; // true or false
```

```java
// Operators
int a = 10, b = 3;
int sum = a + b;       // Arithmetic: +, -, *, /, %
boolean result = a > b; // Relational: ==, !=, <, >, <=, >=
boolean logic = (a > 5) && (b < 5); // Logical: &&, ||, !
a += 5;              // Assignment: =, +=, -=, *=, /=
a++;                 // Increment/Decrement: ++, --
```

---

3. Control Statements

Theory:

Control statements alter the flow of program execution. **If-else** provides conditional execution, **switch** handles multiple conditions efficiently, and **loops** enable repetitive execution. Each serves different scenarios for program flow control.

Code:

java

```java
// If Statement
if (condition) { /* code */ }
else if (condition2) { /* code */ }
else { /* code */ }

// Switch Statement (efficient for multiple exact matches)
switch (variable) {
  case value1: /* code */ break;
  case value2: /* code */ break;
  default: /* code */
}

// Loops
for (int i = 0; i < 10; i++) { /* code */ }     // For loop
while (condition) { /* code */ }               // While loop
do { /* code */ } while (condition);           // Do-while loop
for (int num : array) { /* code */ }          // Enhanced for loop
```

---

4. Arrays

Theory:

Arrays are containers that hold multiple values of the same data type in contiguous memory locations. **1D arrays** store elements in a linear sequence, while **2D arrays** store elements in a matrix format (array of arrays). Arrays have fixed size and zero-based indexing.

Code:

java

```java
// 1D Arrays
int[] arr = new int[5];              // Declaration with size
int[] arr2 = {1, 2, 3, 4, 5};        // Declaration with initialization
arr[0] = 10;                         // Assignment (index 0-4)
int length = arr.length;             // Get length

// 2D Arrays
int[][] matrix = new int[3][4];      // 3 rows, 4 columns
int[][] matrix2 = {{1,2}, {3,4}, {5,6}}; // Initialization
matrix[0][1] = 5;                    // Assignment [row][column]
int rows = matrix.length;            // Number of rows
int cols = matrix[0].length;         // Number of columns
```

---

5. Strings

Theory:

String class creates immutable objects (cannot be modified after creation). Every string operation creates a new object, making it inefficient for multiple concatenations. **StringBuilder** creates mutable objects, allowing efficient string manipulations without creating new objects.

Code:

java

```java
// String Class (Immutable)
String str = "Hello";                // String literal (stored in string pool)
str = str + " World";                // Creates new object
int len = str.length();              // Get length
char ch = str.charAt(0);             // Get character at index
String sub = str.substring(0, 5);    // Extract substring
```

```java
boolean equals = str.equals("Hello");    // Compare strings (content)
boolean same = str == "Hello";          // Compare references

// StringBuilder (Mutable)
StringBuilder sb = new StringBuilder();  // Mutable string
sb.append("Hello");                     // Append (modifies same object)
sb.append(" World");
String result = sb.toString();          // Convert to String
```

---

6. Type Casting

Theory:

Type casting converts one data type to another. **Implicit casting** (widening) happens automatically when converting smaller to larger data types. **Explicit casting** (narrowing) requires manual casting when converting larger to smaller types. **Wrapper classes** provide object representation of primitives.

Code:

java

```java
// Implicit Casting (Widening) - No data loss
int i = 10;
double d = i;                    // int to double (automatic)

// Explicit Casting (Narrowing) - Possible data loss
double d = 10.5;
int i = (int) d;                 // double to int (manual) - loses decimal

// Wrapper Classes
Integer obj = Integer.valueOf(10);      // Boxing (primitive to object)
int primitive = obj.intValue();        // Unboxing (object to primitive)
Integer auto = 10;                     // Auto-boxing
int val = auto;                        // Auto-unboxing
```

---

7. Classes and Objects

Theory:

Class is a blueprint/template that defines properties (variables) and behaviors (methods) of objects. **Object** is an instance of a class that occupies memory and has actual values. Classes provide abstraction and encapsulation, fundamental OOP concepts.

Code:

java

```java
// Class Definition
class Student {
    String name;                    // Instance variable
    int age;                        // Instance variable
    static String school = "ABC School";  // Class variable

    void display() {                // Instance method
        System.out.println(name + " " + age);
    }

    static void showSchool() {      // Class method
        System.out.println(school);
    }
}

// Object Creation and Usage
Student s1 = new Student();         // Object creation
s1.name = "John";                   // Set instance variables
s1.age = 20;
s1.display();                       // Call instance method
Student.showSchool();               // Call class method
```

---

8. Constructors

Theory:

Constructors are special methods that initialize objects during creation. They have the same name as the class and no return type. **Default constructor** has no parameters, **parameterized constructor** accepts parameters. Java provides a default constructor if none is defined.

Code:

java

```java
class Student {
    String name;
    int age;

    // Default constructor
    Student() {
        name = "Unknown";
        age = 0;
    }

    // Parameterized constructor
    Student(String n, int a) {
        name = n;
        age = a;
    }

    // Constructor overloading
    Student(String n) {
        name = n;
        age = 18; // default age
    }
}

// Usage
Student s1 = new Student();            // Calls default constructor
Student s2 = new Student("Alice", 22); // Calls parameterized constructor
Student s3 = new Student("Bob");       // Calls single parameter constructor
```

---

9. Inheritance

Theory:

Inheritance allows a class to acquire properties and methods of another class. **Parent class** (superclass) provides common features, **child class** (subclass) extends parent and can add new features. Promotes code reusability and establishes IS-A relationship.

Code:

```java
// Parent class
class Animal {
    String name;

    void eat() {
        System.out.println("Animal is eating");
    }

    void sleep() {
        System.out.println("Animal is sleeping");
    }
}

// Child class
class Dog extends Animal {
    String breed;

    void bark() {
        System.out.println("Dog is barking");
    }
}

// Usage
Dog d = new Dog();
d.name = "Buddy";          // Inherited property
d.eat();                   // Inherited method
d.bark();                  // Own method

// Types: Single, Multilevel, Hierarchical (Multiple inheritance not supported)
```

---

10. Method Overloading & Overriding

Theory:

Method Overloading (compile-time polymorphism) allows multiple methods with same name but different parameters in the same class. **Method Overriding** (runtime polymorphism) allows child class to provide specific implementation of parent class method.

Code:

```java
// Method Overloading
class Calculator {
    int add(int a, int b) { return a + b; }
    double add(double a, double b) { return a + b; }
    int add(int a, int b, int c) { return a + b + c; }
}

// Method Overriding
class Animal {
    void makeSound() { System.out.println("Animal makes sound"); }
}

class Dog extends Animal {
    @Override
    void makeSound() { System.out.println("Dog barks"); }
}

// Usage
Calculator calc = new Calculator();
calc.add(5, 3);        // Calls int version
calc.add(5.5, 3.2);    // Calls double version

Animal animal = new Dog();
animal.makeSound();    // Calls Dog's version (runtime polymorphism)
```

---

11. `this` and `super` Keywords

Theory:

`this` keyword refers to the current object instance, used to differentiate between instance variables and parameters, or call other constructors. **`super`** keyword refers to the parent class, used to access parent class variables, methods, or constructors.

Code:

```java
class Animal {
    String name = "Animal";
```

```java
    Animal() { System.out.println("Animal constructor"); }

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

class Dog extends Animal {
    String name = "Dog";

    Dog() {
        super();                    // Call parent constructor
        System.out.println("Dog constructor");
    }

    void display() {
        System.out.println(this.name);    // Current class variable
        System.out.println(super.name);   // Parent class variable
        super.display();                  // Parent class method
    }

    void setName(String name) {
        this.name = name;                 // Distinguish parameter from instance variable
    }
}
```

---

12. Access Modifiers

Theory:

Access modifiers control the visibility and accessibility of classes, methods, and variables. They implement encapsulation by restricting access to class members from different parts of the program.

Code:

java

```java
class Example {
    public int publicVar;       // Accessible everywhere
    private int privateVar;     // Accessible only within same class
    protected int protectedVar; // Accessible within package and subclasses
    int defaultVar;             // Accessible within same package (package-private)
```

```java
    public void publicMethod() { }
    private void privateMethod() { }
    protected void protectedMethod() { }
    void defaultMethod() { }
}

// Access from different class
class Test {
    void accessExample() {
        Example ex = new Example();
        ex.publicVar = 10;       // ✓ Allowed
        // ex.privateVar = 20;   // ✗ Not allowed
        ex.protectedVar = 30;    // ✓ Allowed (same package)
        ex.defaultVar = 40;      // ✓ Allowed (same package)
    }
}
```

---

13. Static and Final Keywords

Theory:

`static` keyword creates class-level members that belong to the class rather than instances. Static members are shared among all objects and can be accessed without creating objects. **`final`** keyword creates constants and prevents inheritance/overriding.

Code:

java

```java
class Student {
    static String schoolName = "ABC School";  // Static variable (shared)
    final int maxAge = 25;                    // Final variable (constant)
    static final String COUNTRY = "India";   // Static final (constant)

    String name;                          // Instance variable

    static void displaySchool() {         // Static method
        System.out.println(schoolName);
        // System.out.println(name);       // ✗ Cannot access instance variable
    }
```

```java
    final void finalMethod() {           // Final method (cannot be overridden)
        System.out.println("Final method");
    }
}

// Final class (cannot be extended)
final class FinalClass {
    // class content
}

// Usage
Student.schoolName = "XYZ School";        // Access static variable
Student.displaySchool();                  // Call static method
```

---

14. Abstract Classes & Interfaces

Theory:

Abstract classes contain abstract methods (without implementation) and concrete methods. Cannot be instantiated but can be extended. **Interfaces** contain only abstract methods (Java 8+ allows default/static methods). A class can implement multiple interfaces but extend only one abstract class.

Code:

java
```java
// Abstract Class
abstract class Shape {
    String color;                    // Concrete variable

    abstract void draw();            // Abstract method

    void setColor(String color) {    // Concrete method
        this.color = color;
    }
}

class Circle extends Shape {
    void draw() {                    // Must implement abstract method
```

```java
        System.out.println("Drawing circle");
    }
}

// Interface
interface Drawable {
    int MAX_SIZE = 100;              // public static final by default

    void draw();                     // public abstract by default

    default void print() {           // Default method (Java 8+)
        System.out.println("Printing...");
    }
}

class Rectangle implements Drawable {
    public void draw() {             // Must implement interface method
        System.out.println("Drawing rectangle");
    }
}
```

---

15. Encapsulation & Polymorphism

Theory:

Encapsulation bundles data and methods together and restricts direct access to data using private access modifiers and public getter/setter methods. **Polymorphism** allows objects of different types to be treated as objects of a common base type, enabling method overriding and dynamic method dispatch.

Code:

java

```java
// Encapsulation
class Student {
    private String name;             // Private data
    private int age;

    // Public getter methods
    public String getName() { return name; }
```

```java
    public int getAge() { return age; }

    // Public setter methods with validation
    public void setName(String name) {
        if (name != null && !name.isEmpty()) {
            this.name = name;
        }
    }

    public void setAge(int age) {
        if (age > 0 && age < 100) {
            this.age = age;
        }
    }
}

// Polymorphism
class Animal {
    void makeSound() { System.out.println("Animal sound"); }
}

class Dog extends Animal {
    void makeSound() { System.out.println("Woof"); }
}

class Cat extends Animal {
    void makeSound() { System.out.println("Meow"); }
}

// Runtime polymorphism
Animal[] animals = {new Dog(), new Cat()};
for (Animal animal : animals) {
    animal.makeSound();             // Calls respective overridden methods
}
```

---

16. Exception Handling

Theory:

Exception handling manages runtime errors gracefully without crashing the program. **try-catch** blocks handle exceptions, **finally** block executes regardless of exceptions, **throw** manually throws exceptions, and **throws** declares exceptions that a method might throw.

Code:

java

```java
// Basic Exception Handling
try {
    int result = 10 / 0;              // May throw ArithmeticException
    int[] arr = new int[5];
    arr[10] = 100;                    // May throw ArrayIndexOutOfBoundsException
} catch (ArithmeticException e) {
    System.out.println("Division by zero: " + e.getMessage());
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("Array index error: " + e.getMessage());
} catch (Exception e) {               // Generic exception handler
    System.out.println("General error: " + e.getMessage());
} finally {
    System.out.println("Finally block always executes");
}

// Throwing exceptions
void validateAge(int age) throws IllegalArgumentException {
    if (age < 0) {
        throw new IllegalArgumentException("Age cannot be negative");
    }
}

// Exception hierarchy: Throwable → Exception → RuntimeException
// Checked exceptions: IOException, SQLException (must be handled)
// Unchecked exceptions: RuntimeException, NullPointerException (optional handling)
```

---

17. Custom Exceptions

Theory:

Custom exceptions are user-defined exception classes that extend existing exception classes. They provide specific error handling for application-specific scenarios. Can extend **Exception** (checked) or **RuntimeException** (unchecked) based on requirements.

Code:

java

```java
// Custom Checked Exception
class InvalidAgeException extends Exception {
    public InvalidAgeException(String message) {
        super(message);              // Call parent constructor
    }
}

// Custom Unchecked Exception
class InsufficientBalanceException extends RuntimeException {
    private double currentBalance;

    public InsufficientBalanceException(String message, double balance) {
        super(message);
        this.currentBalance = balance;
    }

    public double getCurrentBalance() {
        return currentBalance;
    }
}

// Usage
class BankAccount {
    private double balance;

    void withdraw(double amount) throws InsufficientBalanceException {
        if (amount > balance) {
            throw new InsufficientBalanceException(
                "Insufficient balance", balance);
        }
        balance -= amount;
    }

    void setAge(int age) throws InvalidAgeException {
        if (age < 18 || age > 100) {
            throw new InvalidAgeException("Age must be between 18-100");
```

```
        }
    }
}

// Handling custom exceptions
try {
    BankAccount account = new BankAccount();
    account.withdraw(1000);
} catch (InsufficientBalanceException e) {
    System.out.println("Error: " + e.getMessage());
    System.out.println("Current balance: " + e.getCurrentBalance());
}
```

18. Collections (List, Set, Map, Queue)

Theory:

- Collections Framework is hierarchy of interfaces and classes for storing/manipulating groups of objects
- **List:** Ordered collection, allows duplicates, indexed access. Implementations: ArrayList (resizable array), LinkedList (doubly-linked), Vector (synchronized)
- **Set:** No duplicate elements. HashSet (hash table), LinkedHashSet (maintains insertion order), TreeSet (sorted)
- **Map:** Key-value pairs, no duplicate keys. HashMap (hash table), LinkedHashMap (insertion order), TreeMap (sorted by keys)
- **Queue:** FIFO operations. LinkedList, PriorityQueue (heap-based), ArrayDeque (resizable array)

Code:

java

```java
// List operations
List<String> arrayList = new ArrayList<>();
arrayList.add("first"); arrayList.add(0, "start"); // Add at index
String item = arrayList.get(0); // Access by index

// Set operations
Set<String> hashSet = new HashSet<>();
hashSet.add("unique"); boolean contains = hashSet.contains("unique");

// Map operations
Map<String, Integer> hashMap = new HashMap<>();
```

```java
hashMap.put("key1", 100); Integer value = hashMap.get("key1");
hashMap.putIfAbsent("key2", 200); // Only if key doesn't exist

// Queue operations
Queue<String> queue = new LinkedList<>();
queue.offer("first"); String head = queue.poll(); // Remove and return head
```

19. Iterator, Comparable vs Comparator

Theory:

- **Iterator:** Safe way to traverse collections, supports remove during iteration
- **ListIterator:** Bidirectional iteration for Lists, supports add/set operations
- **Comparable:** Natural ordering, implement in the class itself, single sorting logic
- **Comparator:** Custom ordering, external comparison, multiple sorting strategies possible

Code:

java

```java
// Iterator usage
List<String> list = Arrays.asList("a", "b", "c");
Iterator<String> it = list.iterator();
while(it.hasNext()) { String s = it.next(); if(s.equals("b")) it.remove(); }

// Comparable implementation
class Student implements Comparable<Student> {
  int marks;
  public int compareTo(Student other) { return Integer.compare(this.marks, other.marks); }
}

// Comparator usage
Collections.sort(students, (s1, s2) -> s1.name.compareTo(s2.name)); // By name
Collections.sort(students, Comparator.comparing(Student::getMarks).reversed()); // By marks desc
```

20. Wrapper Classes, Autoboxing

Theory:

- **Wrapper Classes:** Object representation of primitives (Integer, Double, Boolean, Character, etc.)
- **Autoboxing:** Automatic conversion from primitive to wrapper object
- **Unboxing:** Automatic conversion from wrapper to primitive
- **valueOf():** Returns wrapper object, may cache values (-128 to 127 for Integer)

- **parseXxx():** Converts String to primitive

Code:

java

```java
// Autoboxing and Unboxing
Integer obj1 = 100; // Autoboxing: int -> Integer
int primitive = obj1; // Unboxing: Integer -> int
Integer obj2 = Integer.valueOf(100); // Same object due to caching (-128 to 127)

// String conversions
int num = Integer.parseInt("123"); // String to primitive
Integer obj = Integer.valueOf("456"); // String to wrapper
String str = Integer.toString(789); // primitive/wrapper to String

// Null safety with wrapper classes
Integer nullableInt = null;
// int result = nullableInt; // NullPointerException during unboxing
```

21. Static & Instance Blocks

Theory:

- **Static Block:** Executes once when class is loaded into memory, before any instance creation
- **Instance Block:** Executes before constructor, every time object is created
- **Execution Order:** Static blocks → Instance blocks → Constructor
- Used for complex initialization logic that can't be done in single line

Code:

java

```java
class InitializationDemo {
  static int staticVar;
  int instanceVar;

  // Static block - runs once when class loads
  static {
    System.out.println("Static block executed");
    staticVar = 100;
  }

  // Instance block - runs before each constructor call
```

```
    {
        System.out.println("Instance block executed");
        instanceVar = 50;
    }

    public InitializationDemo() {
        System.out.println("Constructor executed");
    }
}
```

22. Inner Classes (basic)

Theory:

- **Member Inner Class:** Has access to outer class instance variables and methods
- **Static Nested Class:** No access to outer instance members, can access only static members
- **Local Inner Class:** Defined inside method, has access to final/effectively final local variables
- **Anonymous Inner Class:** Unnamed class implementing interface or extending class inline

Code:

java

```java
class Outer {
    private int outerVar = 10;
    static int staticVar = 20;

    // Member inner class
    class Inner {
        void display() { System.out.println("Outer var: " + outerVar); }
    }

    // Static nested class
    static class StaticNested {
        void display() { System.out.println("Static var: " + staticVar); }
    }

    void method() {
        // Local inner class
        class LocalInner {
```

```java
        void show() { System.out.println("Local inner"); }
      }
      LocalInner local = new LocalInner();
    }
}

// Usage
Outer outer = new Outer();
Outer.Inner inner = outer.new Inner(); // Member inner class object
Outer.StaticNested nested = new Outer.StaticNested(); // Static nested class object
```

23. Thread Class & Runnable

Theory:

- **Thread Class:** Provides thread functionality, extends it to create custom thread
- **Runnable Interface:** Defines task to execute, implement run() method
- **Best Practice:** Use Runnable (composition) over Thread (inheritance) for better design
- **Thread Methods:** start(), run(), sleep(), join(), interrupt(), isAlive()

Code:

java

```java
// Extending Thread class
class MyThread extends Thread {
  public void run() { System.out.println("Thread: " + Thread.currentThread().getName()); }
}

// Implementing Runnable (Preferred)
class MyTask implements Runnable {
  public void run() {
    for(int i = 0; i < 5; i++) {
      System.out.println(Thread.currentThread().getName() + ": " + i);
      try { Thread.sleep(1000); } catch(InterruptedException e) { }
    }
  }
}

// Usage
MyThread t1 = new MyThread(); t1.start();
Thread t2 = new Thread(new MyTask()); t2.start();
```

```
Thread t3 = new Thread(() -> System.out.println("Lambda task")); t3.start();
```

24. Thread Lifecycle & Synchronization

Theory:

- **Thread States:** NEW → RUNNABLE → BLOCKED/WAITING/TIMED_WAITING → TERMINATED
- **Synchronization:** Prevents race conditions when multiple threads access shared resources
- **synchronized keyword:** Method level or block level synchronization
- **wait(), notify(), notifyAll():** Inter-thread communication methods (must be in synchronized context)

Code:

java

```java
class Counter {
    private int count = 0;

    // Synchronized method
    public synchronized void increment() { count++; }

    // Synchronized block
    public void decrement() {
        synchronized(this) { count--; }
    }

    public synchronized int getCount() { return count; }
}

// Producer-Consumer with wait/notify
class SharedResource {
    private boolean hasData = false;

    public synchronized void produce() throws InterruptedException {
        while(hasData) wait(); // Wait if data already exists
        System.out.println("Produced data");
        hasData = true;
        notify(); // Notify consumer
    }
```

```java
    public synchronized void consume() throws InterruptedException {
        while(!hasData) wait(); // Wait if no data
        System.out.println("Consumed data");
        hasData = false;
        notify(); // Notify producer
    }
}
```

25. JDBC

Theory:

- **JDBC:** Java Database Connectivity API for database operations
- **Steps:** Load driver → Create connection → Create statement → Execute query → Process results → Close resources
- **Statement Types:** Statement (static SQL), PreparedStatement (parameterized), CallableStatement (stored procedures)
- **ResultSet:** Contains query results, cursor-based navigation

Code:

java

```java
// Database connection and operations
String url = "jdbc:mysql://localhost:3306/testdb";
String username = "root", password = "admin";

try (Connection con = DriverManager.getConnection(url, username, password)) {

    // PreparedStatement for parameterized queries
    String sql = "SELECT * FROM users WHERE age > ? AND city = ?";
    PreparedStatement ps = con.prepareStatement(sql);
    ps.setInt(1, 25);
    ps.setString(2, "Mumbai");

    ResultSet rs = ps.executeQuery();
    while(rs.next()) {
        System.out.println("Name: " + rs.getString("name") + ", Age: " + rs.getInt("age"));
    }

    // Insert operation
    String insertSql = "INSERT INTO users(name, age, city) VALUES(?, ?, ?)";
    PreparedStatement insertPs = con.prepareStatement(insertSql);
```

```java
    insertPs.setString(1, "John");
    insertPs.setInt(2, 30);
    insertPs.setString(3, "Delhi");
    int rowsAffected = insertPs.executeUpdate();

} catch(SQLException e) { e.printStackTrace(); }
```

26. Lambda Expressions

Theory:

- **Lambda:** Anonymous function that can be passed around as parameter
- **Syntax:** (parameters) -> expression or (parameters) -> { statements; }
- **Used with:** Functional interfaces (interfaces with single abstract method)
- **Benefits:** Concise code, functional programming support, better readability
- **Method References:** Shorthand for lambdas (Class::method)

Code:

java

```java
// Basic lambda syntax
Runnable task = () -> System.out.println("Hello Lambda");
Comparator<String> comp = (s1, s2) -> s1.length() - s2.length();

// Lambda with collections
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
numbers.forEach(n -> System.out.println(n * 2)); // Print double of each

// Filtering and processing
List<String> names = Arrays.asList("John", "Jane", "Jack", "Jill");
names.stream()
    .filter(name -> name.startsWith("J"))
    .map(String::toUpperCase)
    .forEach(System.out::println);

// Method references
List<String> words = Arrays.asList("apple", "banana", "cherry");
words.sort(String::compareToIgnoreCase); // Method reference
words.forEach(System.out::println); // Method reference
```

27. Stream API

Theory:

- **Stream:** Sequence of elements supporting sequential/parallel operations
- **Characteristics:** Not a data structure, functional approach, lazy evaluation
- **Operations:** Intermediate (filter, map, sorted) and Terminal (collect, forEach, reduce)
- **Pipeline:** Source → Intermediate operations → Terminal operation
- **Parallel Streams:** Use parallelStream() for parallel processing

Code:

java

```java
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

// Filtering and collecting
List<Integer> evenNumbers = numbers.stream()
    .filter(n -> n % 2 == 0)
    .collect(Collectors.toList());

// Mapping and statistics
List<String> words = Arrays.asList("apple", "banana", "cherry");
int totalLength = words.stream()
    .mapToInt(String::length)
    .sum();

// Complex operations
Map<Boolean, List<Integer>> partitioned = numbers.stream()
    .collect(Collectors.partitioningBy(n -> n % 2 == 0));

// Grouping
List<Person> people = Arrays.asList(/* person objects */);
Map<String, List<Person>> groupedByCity = people.stream()
    .collect(Collectors.groupingBy(Person::getCity));

// Reduction operations
Optional<Integer> max = numbers.stream().max(Integer::compareTo);
int sum = numbers.stream().reduce(0, Integer::sum);
```

28. Functional Interfaces

Theory:

- **Functional Interface:** Interface with exactly one abstract method (SAM - Single Abstract Method)
- **@FunctionalInterface:** Annotation to ensure interface has only one abstract method
- **Built-in Interfaces:** Predicate<T>, Function<T,R>, Consumer<T>, Supplier<T>, UnaryOperator<T>, BinaryOperator<T>
- **Used with:** Lambda expressions and method references

Code:

java

```java
// Built-in functional interfaces
Predicate<Integer> isEven = n -> n % 2 == 0;
Function<String, Integer> stringLength = String::length;
Consumer<String> printer = System.out::println;
Supplier<String> stringSupplier = () -> "Hello World";

// Custom functional interface
@FunctionalInterface
interface Calculator {
  int calculate(int a, int b);
  // Can have default and static methods
  default void info() { System.out.println("Calculator interface"); }
}

// Usage
Calculator add = (a, b) -> a + b;
Calculator multiply = (a, b) -> a * b;
System.out.println(add.calculate(5, 3)); // 8

// Chaining functional interfaces
Predicate<String> startsWithA = s -> s.startsWith("A");
Predicate<String> lengthGreaterThan3 = s -> s.length() > 3;
Predicate<String> combined = startsWithA.and(lengthGreaterThan3);
```

29. Date and Time API

Theory:

- **Java 8+ API:** Modern, immutable, thread-safe date/time handling
- **Main Classes:** LocalDate (date only), LocalTime (time only), LocalDateTime (both), ZonedDateTime (with timezone)
- **Formatting:** DateTimeFormatter for custom formats
- **Parsing:** Parse strings to date/time objects

- **Calculations:** Plus/minus operations, between calculations

Code:

java

```java
// Creating date/time objects
LocalDate today = LocalDate.now();
LocalDate specificDate = LocalDate.of(2024, Month.JANUARY, 15);
LocalTime currentTime = LocalTime.now();
LocalDateTime dateTime = LocalDateTime.now();

// Formatting and parsing
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd-MM-yyyy");
String formattedDate = today.format(formatter);
LocalDate parsedDate = LocalDate.parse("15-01-2024", formatter);

// Calculations
LocalDate tomorrow = today.plusDays(1);
LocalDate lastWeek = today.minusWeeks(1);
long daysBetween = ChronoUnit.DAYS.between(specificDate, today);

// Working with zones
ZonedDateTime zonedNow = ZonedDateTime.now(ZoneId.of("Asia/Kolkata"));
ZonedDateTime utcTime = zonedNow.withZoneSameInstant(ZoneOffset.UTC);

// Period and Duration
Period period = Period.between(specificDate, today);
Duration duration = Duration.between(LocalTime.of(9, 0), LocalTime.of(17, 30));
```

30. Generics

Theory:

- **Generics:** Provide type safety at compile time, eliminate casting
- **Type Parameters:** T (Type), E (Element), K (Key), V (Value), N (Number)
- **Wildcards:** ? (unknown), ? extends T (upper bound), ? super T (lower bound)
- **Type Erasure:** Generic type information removed at runtime
- **Bounded Types:** <T extends SomeClass> restricts type parameter

Code:

java

```java
// Generic class
```

```java
class Box<T> {
    private T content;
    public void set(T content) { this.content = content; }
    public T get() { return content; }
}

// Generic methods
public class GenericMethods {
    public static <T> void swap(T[] array, int i, int j) {
        T temp = array[i];
        array[i] = array[j];
        array[j] = temp;
    }

    // Bounded type parameter
    public static <T extends Number> double average(T[] numbers) {
        return Arrays.stream(numbers)
                .mapToDouble(Number::doubleValue)
                .average().orElse(0.0);
    }
}

// Wildcards usage
List<? extends Number> numbers = new ArrayList<Integer>(); // Upper bound
List<? super Integer> integers = new ArrayList<Number>(); // Lower bound

// Generic interface
interface Repository<T, ID> {
    void save(T entity);
    T findById(ID id);
    List<T> findAll();
}
```

31. Enums

Theory:

- **Enum:** Special class type for defining named constants
- **Benefits:** Type safety, namespace, can have methods/constructors/fields
- **Implicit Methods:** values(), valueOf(), ordinal(), name()
- **Can implement interfaces** and have abstract methods

- **Each enum constant is public, static, final instance**

Code:

java

```java
// Simple enum
enum Status { ACTIVE, INACTIVE, PENDING }

// Enum with fields and methods
enum Planet {
    MERCURY(3.303e+23, 2.4397e6),
    VENUS(4.869e+24, 6.0518e6),
    EARTH(5.976e+24, 6.37814e6);

    private final double mass;
    private final double radius;

    Planet(double mass, double radius) {
        this.mass = mass;
        this.radius = radius;
    }

    public double getMass() { return mass; }
    public double getRadius() { return radius; }
    public double surfaceGravity() { return 6.67300E-11 * mass / (radius * radius); }
}

// Usage
Status currentStatus = Status.ACTIVE;
System.out.println(currentStatus.name()); // "ACTIVE"
System.out.println(currentStatus.ordinal()); // 0

// Switch with enum
switch(currentStatus) {
    case ACTIVE: System.out.println("System is running"); break;
    case INACTIVE: System.out.println("System is down"); break;
    default: System.out.println("Unknown status");
}
```

32. Maven (basic)

Theory:

- **Maven:** Build automation and dependency management tool
- **POM (Project Object Model):** pom.xml contains project configuration
- **Standard Directory Layout:** src/main/java, src/test/java, src/main/resources
- **Lifecycle Phases:** validate → compile → test → package → verify → install → deploy
- **Dependencies:** Managed through Maven Central Repository

Code:

xml

```xml
<!-- Basic pom.xml structure -->
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.company</groupId>
  <artifactId>my-app</artifactId>
  <version>1.0.0</version>
  <packaging>jar</packaging>

  <properties>
    <maven.compiler.source>11</maven.compiler.source>
    <maven.compiler.target>11</maven.compiler.target>
  </properties>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.13.2</version>
      <scope>test</scope>
    </dependency>

    <dependency>
      <groupId>org.apache.commons</groupId>
      <artifactId>commons-lang3</artifactId>
      <version>3.12.0</version>
    </dependency>
  </dependencies>
</project>
```

Commands:

bash
```bash
mvn clean compile      # Clean and compile
mvn test               # Run tests
mvn package            # Create JAR/WAR
mvn install            # Install to local repository
mvn dependency:tree    # Show dependency tree
```

33. Spring Boot (basic)

Theory:

- **Spring Boot:** Framework for rapid application development with minimal configuration
- **Auto-configuration:** Automatically configures application based on dependencies
- **Starter Dependencies:** Pre-configured dependency groups (spring-boot-starter-web, spring-boot-starter-data-jpa)
- **Embedded Server:** Built-in Tomcat/Jetty/Undertow
- **Key Annotations:** @SpringBootApplication, @RestController, @Service, @Repository, @Autowired

Code:

java
```java
// Main application class
@SpringBootApplication  // Combines @Configuration, @EnableAutoConfiguration, @ComponentScan
public class MyApplication {
    public static void main(String[] args) {
        SpringApplication.run(MyApplication.class, args);
    }
}

// REST Controller
@RestController
@RequestMapping("/api/users")
public class UserController {

    @Autowired
    private UserService userService;

    @GetMapping
    public List<User> getAllUsers() {
```

```
        return userService.findAll();
    }

    @GetMapping("/{id}")
    public User getUserById(@PathVariable Long id) {
        return userService.findById(id);
    }

    @PostMapping
    public User createUser(@RequestBody User user) {
        return userService.save(user);
    }

    @PutMapping("/{id}")
    public User updateUser(@PathVariable Long id, @RequestBody User user) {
        user.setId(id);
        return userService.save(user);
    }

    @DeleteMapping("/{id}")
    public void deleteUser(@PathVariable Long id) {
        userService.deleteById(id);
    }
}

// Service layer
@Service
public class UserService {
    @Autowired
    private UserRepository userRepository;

    public List<User> findAll() { return userRepository.findAll(); }
    public User findById(Long id) { return userRepository.findById(id).orElse(null); }
    public User save(User user) { return userRepository.save(user); }
    public void deleteById(Long id) { userRepository.deleteById(id); }
}
```

34. REST API (basic)

Theory:

- **REST:** Representational State Transfer architectural style
- **HTTP Methods:** GET (read), POST (create), PUT (update/replace), PATCH (partial update), DELETE (remove)
- **Status Codes:** 200 (OK), 201 (Created), 400 (Bad Request), 404 (Not Found), 500 (Internal Server Error)
- **Stateless:** Each request contains all information needed to process it
- **Resource-based URLs:** /users, /users/123, /users/123/orders

Code:

java

```java
@RestController
@RequestMapping("/api/products")
public class ProductController {

  @Autowired
  private ProductService productService;

  // GET /api/products - Get all products
  @GetMapping
  public ResponseEntity<List<Product>> getAllProducts(
      @RequestParam(defaultValue = "0") int page,
      @RequestParam(defaultValue = "10") int size) {
    List<Product> products = productService.findAll(page, size);
    return ResponseEntity.ok(products);
  }

  // GET /api/products/123 - Get product by ID
  @GetMapping("/{id}")
  public ResponseEntity<Product> getProductById(@PathVariable Long id) {
    Product product = productService.findById(id);
    if (product != null) {
      return ResponseEntity.ok(product);
    }
    return ResponseEntity.notFound().build();
  }

  // POST /api/products - Create new product
  @PostMapping
  public ResponseEntity<Product> createProduct(@Valid @RequestBody Product product) {
    Product savedProduct = productService.save(product);
```

```java
        return ResponseEntity.status(HttpStatus.CREATED).body(savedProduct);
    }

    // PUT /api/products/123 - Update product
    @PutMapping("/{id}")
    public ResponseEntity<Product> updateProduct(@PathVariable Long id,
                                                  @Valid @RequestBody Product product) {
        if (!productService.existsById(id)) {
            return ResponseEntity.notFound().build();
        }
        product.setId(id);
        Product updatedProduct = productService.save(product);
        return ResponseEntity.ok(updatedProduct);
    }

    // DELETE /api/products/123 - Delete product
    @DeleteMapping("/{id}")
    public ResponseEntity<Void> deleteProduct(@PathVariable Long id) {
        if (!productService.existsById(id)) {
            return ResponseEntity.notFound().build();
        }
        productService.deleteById(id);
        return ResponseEntity.noContent().build();
    }

    // Exception handling
    @ExceptionHandler(ProductNotFoundException.class)
    public ResponseEntity<String> handleProductNotFound(ProductNotFoundException e) {
        return ResponseEntity.status(HttpStatus.NOT_FOUND).body(e.getMessage());
    }
}
```

------------x-----------x----------x-----------x------------x-----------x------------x------------x------------x-----------x-------