Java CheatSheet

JAVA CHEATSHEET

Intro to Java:
OOP language • Platform Independent • JVM, JRE, JDK • Editions: JSE, JEE, JME

Basic Syntax:
class > main() • Data Types • Scanner • println • Comments • Type Casting

Operators:
/ % • == != > < • && || ! • ?: • = += -=

Control Statements:
if, else, else if • switch-case • Nested if

Loops:
for • while • do...while • for-each • break • continue

Arrays:
1D, 2D Arrays • arr.length • Arrays.sort()

Strings:
length() • charAt() • concat() • equals() • compareTo() • StringBuilder

Methods:
void, return • Overloading • static • Recursion

OOP:
Class, Object • Constructor • this • Inheritance • super • Polymorphism • Abstraction • Encapsulation • Interface

Exception Handling:
try-catch-finally • throw • throws

Packages & Imports:
package • import • static import

Collections:
List • Set • Map • ArrayList • HashSet • HashMap • Queue • for-each • Iterator

File Handling:
FileReader/Writer • BufferedReader • Scanner • File Class

Multithreading:
Thread & Runnable • Lifecycle • sync • wait/notify

Java 8+ Features:
Lambda • Stream API • Optional • Functional Interfaces • forEach

Generics:
<T> • Wildcards <?>, <? extends>, <? super>

Annotations:
@Override • @Deprecated • Custom

Wrapper Classes:
int ↔ Integer • parseInt(), valueOf()

Enums:
enum Days {MON, TUE} • name() • ordinal()

Date & Time API:
LocalDate • DateTimeFormatter • Calendar

Utility Classes:
Math • Random • Arrays • Collections

Mini Projects:
Calculator, ATM, To-Do App, File Encryptor

TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS

1. INTRODUCTION TO JAVA

1.1 What is Java?

Java is a computer language that helps us tell computers what to do. It can be used to make:

- Games
- Mobile apps
- Websites
- Big computer programs

- Just like how we use Gujarati, Hindi, or English to talk, computers use Java to understand us.

1.2 History and Features

- Java was created in 1995 by a company named Sun Microsystems.
- It was later bought by Oracle (a big tech company).

Cool Features of Java:

- Write once, run anywhere: You write Java code once and it can work on many computers.
- Object-Oriented: Java organizes code in reusable blocks called "objects".
- Secure: Java is safe from hackers.
- Simple and powerful: Easy to learn, but also very strong.

1.3 Java Editions (JSE, JEE, JME)

- Java comes in 3 types (or editions), like 3 types of chocolate:

JSE (Java Standard Edition):

- Used for basic things like games, calculators, and simple apps.

JEE (Java Enterprise Edition):

- Used to make big programs for companies like Amazon or Flipkart.

JME (Java Micro Edition):

- Used in small devices like old phones, washing machines, etc.

1.4 JVM, JRE, and JDK

- These are like parts of a machine that help Java work.

JDK (Java Development Kit):

- A box of tools that helps you create and run Java programs.

JRE (Java Runtime Environment):

- A smaller box that lets you only run Java programs (not create them).

JVM (Java Virtual Machine):

- The brain of Java. It reads your Java program and makes the computer understand it.

1. INTRODUCTION TO JAVA

1.5 How Java Works

- You write your program in Java.
- Java turns it into something called Bytecode.
- The JVM reads the bytecode and tells the computer what to do.

It's like:

- You write a letter in Java.
- Java translates it to bytecode.
- JVM reads that letter and tells the computer.

2. BASIC SYNTAX

2.1 Structure of a Java Program

- A simple program in Java looks like this:

```java
public class Hello {
  public static void main(String[] args) {
    System.out.println("Hello, world!");
  }
}
```

What it does:

- It prints "Hello, world!" on the screen.

2.2 Comments

- Comments are notes for humans reading the code. Computers ignore comments.

```java
// This is a single-line comment

/* This is
   a multi-line comment */
```

2.3 Main Method Explained

```java
public static void main(String[] args) {
  // code goes here
}
```

- This is the starting point of every Java program. It's like the "Start" button in a game.

2.4 Data Types and Variables

- Variables are like boxes to store information.
- Data types tell us what kind of information is in the box.

Example:

```java
int age = 10;          // number
float price = 5.5f;    // decimal number
char grade = 'A';      // single letter
String name = "Ravi"; // group of letters (text)
boolean isCool = true; // true or false
```

2. BASIC SYNTAX

2.5 Type Casting

- Changing from one type to another:

Example:

```
int a = 10;
double b = a;   // auto type casting
```

- Or manually:

Example:

```
double x = 5.6;
int y = (int)x; // manual type casting
```

2.6 Keywords

- Java has special words that are already reserved. You can't use them as names.
- Examples: class, int, if, else, while

2.7 Input/Output (Scanner, System.out)

- System.out.println() is used to show output on screen.

Example:

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

- Scanner is used to take input from the user:

Example:

```
import java.util.Scanner;

Scanner sc = new Scanner(System.in);
int age = sc.nextInt(); // user enters a number
```

3. OPERATORS

3.1 Arithmetic Operators

- Used to do math:

Example:

```
+  // add
-  // subtract
*  // multiply
/  // divide
%  // remainder
```

Example:

```
int x = 10 + 5; // x is 15
```

3.2 Assignment Operators

- Used to give values to variables:

Example:

```
=   // assign
+=  // add and assign
-=  // subtract and assign
```

Example:

```
x += 3;  // same as x = x + 3;
```

3.3 Comparison Operators

- Used to compare things:

Example:

```
==  // is equal?
≠   // not equal?
>   // greater than
<   // less than
≥   // greater or equal
≤   // less or equal
```

Example:

```
if (age ≥ 18) {
  System.out.println("You can vote");
}
```

3. OPERATORS

3.4 Logical Operators

- Used to combine conditions:

Example:

```
&&  // AND (both must be true)
||  // OR (any one can be true)
!   // NOT (opposite)
```

3.5 Bitwise Operators

- Used in advanced calculations using 0s and 1s.

Example:

```
&  |  ^  ~  <<  >>
```

- (Not needed for beginners.)

3.6 Ternary Operator

- Shortcut for if...else:

Example:

```
int max = (a > b) ? a : b;
```

- It means: if a > b, then max = a, else max = b.

3.7 Operator Precedence

- Some operators happen before others.

Example:

```
int x = 5 + 3 * 2; // result is 11, not 16
```

- Because multiplication happens before addition.

4. CONTROL STATEMENTS

4.1 if, else if, else

- Used to make decisions:

Example:

```
if (age < 10) {
  System.out.println("You are a child");
} else if (age < 18) {
  System.out.println("You are a teenager");
} else {
  System.out.println("You are an adult");
}
```

4.2 switch Statement

- Checks many values:

Example:

```
int day = 2;
switch (day) {
  case 1: System.out.println("Monday"); break;
  case 2: System.out.println("Tuesday"); break;
  default: System.out.println("Other day");
}
```

4.3 Nested if Statements

- One if inside another:

Example:

```
if (age > 5) {
  if (age < 10) {
    System.out.println("You are in school");
  }
}
```

5. LOOPS

5.1 for, while, do...while

- for loop:

Example:

```
for (int i = 1; i <= 5; i++) {
  System.out.println(i);
}
```

- while loop:

Example:

```
int i = 1;
while (i <= 5) {
  System.out.println(i);
  i++;
}
```

- do...while loop:

Example:

```
int i = 1;
do {
  System.out.println(i);
  i++;
} while (i <= 5);
```

5.2 Enhanced for-each Loop

- Used for arrays (list of items):

Example:

```
int[] numbers = {10, 20, 30};
for (int num : numbers) {
  System.out.println(num);
}
```

5.3 Loop Control (break, continue)

- break = stop the loop
- continue = skip to the next loop

5. LOOPS

5.3 Loop Control (break, continue)

Example:

```
for (int i = 1; i <= 5; i++) {
  if (i == 3) break;      // stops at 2
  System.out.println(i);
}

for (int i = 1; i <= 5; i++) {
  if (i == 3) continue;   // skips 3
  System.out.println(i);
}
```

6. ARRAYS

6.1 Single-Dimensional Arrays

- An array is like a row of boxes where each box holds a value (like numbers, names, etc.).

Example:

```
int[] numbers = {10, 20, 30, 40};
System.out.println(numbers[0]); // shows 10
```

- You can also create an empty array:

Example:

```
int[] marks = new int[5]; // 5 boxes
marks[0] = 90;
```

6.2 Multi-Dimensional Arrays

- These are like tables or grids (rows and columns).

Example:

```
int[][] matrix = {
  {1, 2, 3},
  {4, 5, 6}
};
System.out.println(matrix[1][2]); // shows 6
```

- First number = row, second = column.

6.3 Array Methods

- Java has helpful tools (methods) to work with arrays:

Example:

```
import java.util.Arrays;

int[] nums = {4, 1, 3};
Arrays.sort(nums); // Sorts the array
System.out.println(Arrays.toString(nums)); // [1, 3, 4]
```

Other common methods:

- Arrays.copyOf()
- Arrays.equals()
- Arrays.fill()

6. ARRAYS

6.4 Array Iteration

- You can go through each item using a loop:

Example:

```
int[] nums = {1, 2, 3};

for (int i = 0; i < nums.length; i++) {
  System.out.println(nums[i]);
}
```

- Or use for-each:

Example:

```
for (int num : nums) {
  System.out.println(num);
}
```

7. STRINGS

7.1 String Declaration

- A String is a group of characters (letters, words).

Example:

```
String name = "Ravi";
```

- You can also do:

Example:

```
String city = new String("Ahmedabad");
```

7.2 String Methods

- Here are some magic tricks 🪄 you can do with Strings:

Example:

```
String name = "Ravi";
System.out.println(name.length());     // 4
System.out.println(name.charAt(0));    // R
System.out.println(name.concat(" Patel")); // Ravi Patel
System.out.println(name.equals("Ravi"));   // true
System.out.println(name.compareTo("Ravi")); // 0
```

7.3 StringBuilder and StringBuffer

- Both are used to build strings faster (like for games or apps where speed matters):

Example:

```
StringBuilder sb = new StringBuilder("Hi");
sb.append(" there");
System.out.println(sb); // Hi there
```

- StringBuffer is similar but safe for multi-users (thread-safe).

7.4 String Formatting

- You can make strings look fancy:

Example:

```
String name = "Ravi";
int age = 10;

System.out.printf("My name is %s and I am %d years old", name, age);
// Output: My name is Ravi and I am 10 years old
```

8. METHODS (FUNCTIONS)

8.1 Defining and Calling Methods

- Methods are blocks of code that do a task.

Example:

```
public static void sayHello() {
  System.out.println("Hello!");
}

sayHello(); // Call the method
```

- You can also send info:

Example:

```
public static void greet(String name) {
  System.out.println("Hi, " + name);
}
greet("Ravi"); // Hi, Ravi
```

8.2 Method Overloading

- You can make methods with the same name, but different inputs:

Example:

```
public static void add(int a, int b) {
  System.out.println(a + b);
}

public static void add(double a, double b) {
  System.out.println(a + b);
}
```

8.3 Recursion

- When a method calls itself:

Example:

```
public static int countdown(int n) {
  if (n == 0) return 0;
  System.out.println(n);
  return countdown(n - 1);
}
```

- It’s like a loop, but using self-calls.

8. METHODS (FUNCTIONS)

8.4 static Keyword

- If a method or variable is marked as static, it belongs to the class — not just one object.

Example:

```
public class MathHelper {
  static int square(int x) {
    return x * x;
  }
}

System.out.println(MathHelper.square(5)); // 25
```

- You don’t need to create an object to use static methods.

9. OBJECT-ORIENTED PROGRAMMING (OOP)

- Object-Oriented Programming helps you organize your code better using objects. Think of it like building things in real life: You make blueprints (classes) and then build objects from them.

9.1 Classes and Objects

- A class is like a blueprint. An object is something you create from that blueprint.

Example:

```
class Car {
  String color;
  void start() {
    System.out.println("Car is starting");
  }
}

public class Main {
  public static void main(String[] args) {
    Car myCar = new Car(); // object
    myCar.color = "Red";
    myCar.start();
  }
}
```

9.2 Constructors

- A constructor is a special method that runs when an object is created.

Example:

```
class Car {
  Car() {
    System.out.println("Car is created");
  }
}
```

- You can also create constructors with parameters:

Example:

```
class Car {
  String color;
  Car(String c) {
    color = c;
  }
}
```

9. OBJECT-ORIENTED PROGRAMMING (OOP)

9.3 this Keyword

- The this keyword refers to the current object.

Example:

```
class Car {
  String color;

  Car(String color) {
    this.color = color; // refers to the class variable
  }
}
```

9.4 Inheritance

- Inheritance means a class can use things from another class.

Example:

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

class Dog extends Animal {
  void bark() {
    System.out.println("Dog barks");
  }
}
```

9.5 super Keyword

- super is used to call the parent class methods or constructors.

Example:

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

class Dog extends Animal {
  Dog() {
    super(); // calls parent constructor
    System.out.println("Dog created");
  }
}
```

9. OBJECT-ORIENTED PROGRAMMING (OOP)

9.6 Method Overriding

- When a child class changes a method from the parent class.

Example:

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

class Cat extends Animal {
  @Override
  void sound() {
    System.out.println("Meow");
  }
}
```

9.7 Polymorphism (Compile-time and Runtime)

- Polymorphism means "many forms".
- Compile-time (Method Overloading):

Example:

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

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

- Runtime (Method Overriding):

Example:

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

class Cow extends Animal {
  void sound() {
    System.out.println("Moo");
  }
}
```

9. OBJECT-ORIENTED PROGRAMMING (OOP)

9.8 Abstraction (abstract Classes)

- Abstraction hides unnecessary details. Use abstract classes to create base templates.

Example:

```
abstract class Animal {
  abstract void sound(); // no body

  void sleep() {
    System.out.println("Sleeping");
  }
}

class Lion extends Animal {
  void sound() {
    System.out.println("Roar");
  }
}
```

9.9 Encapsulation

- Encapsulation means hiding data using private and accessing it with methods.

Example:

```
class Student {
  private int age;

  public void setAge(int a) {
    age = a;
  }

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

9.10 Interfaces

- Interfaces are like 100% abstract classes. All methods are without body.

Example:

```
interface Animal {
  void sound();
}

class Dog implements Animal {
  public void sound() {
    System.out.println("Bark");
  }
}
```

9. OBJECT-ORIENTED PROGRAMMING (OOP)

9.11 Access Modifiers (private, public, protected, default)

- private: Only inside the same class
- public: Accessible everywhere
- protected: Accessible in the same package and child classes
- default (no keyword): Same package only

Example:

```
public class A {
  private int x = 10;
  public int y = 20;
  protected int z = 30;
  int w = 40; // default
}
```

10. EXCEPTION HANDLING

10.1 try, catch, finally

- Java handles errors using try-catch blocks.

Example:

```
try {
  int x = 10 / 0;
} catch (ArithmeticException e) {
  System.out.println("Can't divide by zero");
} finally {
  System.out.println("Always runs");
}
```

10.2 throw and throws

- throw is used to throw an exception.

Example:

```
throw new ArithmeticException("Error happened");
```

- throws is used in method signature:

Example:

```
void test() throws IOException {
  // code that might throw IOException
}
```

10.3 Custom Exceptions

- You can create your own exception class.

Example:

```
class MyException extends Exception {
  MyException(String msg) {
    super(msg);
  }
}

public class Test {
  public static void main(String[] args) throws MyException {
    throw new MyException("Custom error");
  }
}
```

10. EXCEPTION HANDLING

10.4 Common Exceptions

- ArithmeticException – divide by zero
- NullPointerException – object is null
- ArrayIndexOutOfBoundsException – index too big
- NumberFormatException – wrong format for numbers
- IOException – input/output error

11. PACKAGES & IMPORTS

11.1 Built-in Packages

- Java comes with many ready-made packages (groups of classes).

Example:

- java.util – for tools like ArrayList, Scanner
- java.io – for reading/writing files

Example:

```
import java.util.Scanner; // uses Scanner class from util package
```

11.2 Creating Custom Packages

- A package is a folder to organize your Java files.

Step 1: Create a package

```
package mypackage;

public class Hello {
  public void sayHi() {
    System.out.println("Hello from mypackage");
  }
}
```

Step 2: Use it in another file

```
import mypackage.Hello;

public class Main {
  public static void main(String[] args) {
    Hello h = new Hello();
    h.sayHi();
  }
}
```

11.3 import Statement

- "import" helps bring in other classes or packages into your file.

Example:

```
import java.util.Scanner; // imports only Scanner

import java.util.*; // imports all from util
```

11. PACKAGES & IMPORTS

11.4 static import

- You can use static import to use methods directly without writing class name.

Example:

```
import static java.lang.Math.*; // import all static methods

public class Test {
  public static void main(String[] args) {
    System.out.println(sqrt(25)); // instead of Math.sqrt(25)
  }
}
```

12. JAVA COLLECTIONS FRAMEWORK

- The Collection Framework lets you store and manage groups of data (like a list of names or scores).

12.1 List, Set, Map Interfaces

- List = Ordered collection (can have duplicates)
- Set = No duplicates allowed
- Map = Key-value pairs (like a dictionary)

12.2 ArrayList, LinkedList

- ArrayList – fast to access, like an expandable array

Example:

```
ArrayList<String> names = new ArrayList<>();
names.add("John");
System.out.println(names.get(0)); // John
```

- LinkedList – better for inserting/deleting elements

Example:

```
LinkedList<Integer> numbers = new LinkedList<>();
numbers.add(10);
```

12.3 HashSet, TreeSet

- HashSet – no duplicates, no order

Example:

```
HashSet<String> set = new HashSet<>();
set.add("A");
set.add("B");
```

- TreeSet – sorted automatically

Example:

```
TreeSet<Integer> tSet = new TreeSet<>();
tSet.add(3);
tSet.add(1);
```

12.4 HashMap, TreeMap

- HashMap – key-value pair, fast access

Example:

```
HashMap<Integer, String> map = new HashMap<>();
map.put(1, "One");
map.put(2, "Two");
```

12. JAVA COLLECTIONS FRAMEWORK

12.4 HashMap, TreeMap

- TreeMap – sorted keys

Example:

```
TreeMap<String, Integer> map = new TreeMap<>();
map.put("B", 2);
map.put("A", 1);
```

12.5 Stack, Queue, PriorityQueue

- Stack – Last In First Out (LIFO)

Example:

```
Stack<Integer> stack = new Stack<>();
stack.push(10);
stack.pop();
```

- Queue – First In First Out (FIFO)

Example:

```
Queue<String> q = new LinkedList<>();
q.add("A");
q.remove();
```

- PriorityQueue – sorted automatically

Example:

```
PriorityQueue<Integer> pq = new PriorityQueue<>();
pq.add(5);
pq.add(1);
```

12.6 Iterators and Enhanced for-each

- Iterator – moves through a collection

Example:

```
Iterator<String> it = list.iterator();
while(it.hasNext()) {
  System.out.println(it.next());
}
```

- Enhanced for-each

Example:

```
for(String name : names) {
  System.out.println(name);
}
```

13. FILE HANDLING

13.1 FileReader, FileWriter

- FileWriter – write to a file

Example:

```
FileWriter fw = new FileWriter("output.txt");
fw.write("Hello");
fw.close();
```

- FileReader – read from file

Example:

```
FileReader fr = new FileReader("output.txt");
int i;
while((i = fr.read()) ≠ -1) {
  System.out.print((char)i);
}
fr.close();
```

13.2 BufferedReader, BufferedWriter

- Reads/writes faster by using a buffer.

Example:

```
BufferedReader br = new BufferedReader(new FileReader("file.txt"));
String line = br.readLine();
br.close();
```

Example:

```
BufferedWriter bw = new BufferedWriter(new FileWriter("file.txt"));
bw.write("Line");
bw.close();
```

13.3 Scanner for File Input

Example:

```
File file = new File("data.txt");
Scanner sc = new Scanner(file);
while(sc.hasNextLine()) {
  System.out.println(sc.nextLine());
}
```

13. FILE HANDLING

13.4 File Class Operations

- Check if file exists, or get file info:

Example:

```
File f = new File("data.txt");
if(f.exists()) {
  System.out.println("File found");
  System.out.println("Name: " + f.getName());
  System.out.println("Size: " + f.length());
}
```

14. MULTITHREADING

- Java can do many tasks at once using threads.

14.1 Threads using Thread Class and Runnable Interface

- Using Thread class:

Example:

```
class MyThread extends Thread {
  public void run() {
    System.out.println("Running ... ");
  }
}
```

- Using Runnable:

Example:

```
class MyRun implements Runnable {
  public void run() {
    System.out.println("Running from Runnable");
  }
}
```

14.2 Thread Lifecycle

States:

- New
- Runnable
- Running
- Blocked
- Terminated

Example:

```
Thread t = new Thread();
t.start(); // moves to running
```

14.3 Synchronization

- Stops threads from interfering with each other.

Example:

```
synchronized void print() {
  // only one thread can enter
}
```

14. MULTITHREADING

14.4 wait() and notify()

- wait() – pauses the thread
- notify() – wakes up waiting thread

Example:

```
synchronized(obj) {
  obj.wait();  // waits
  obj.notify(); // wakes another
}
```

15. JAVA 8+ FEATURES

15.1 Lambda Expressions

- Short way to write a method.

Example:

```
(a, b) → a + b
```

Example:

```
interface Addable {
  int add(int a, int b);
}

Addable ad = (a, b) → a + b;
System.out.println(ad.add(5, 3));
```

15.2 Functional Interfaces

- Interfaces with only one abstract method.

Example:

```
@FunctionalInterface
interface Sayable {
  void say();
}
```

15.3 Streams API

- For working with data like lists.

Example:

```
List<String> names = Arrays.asList("A", "B", "C");

names.stream().forEach(System.out::println);
```

15.4 Optional Class

- Helps avoid null pointer errors.

Example:

```
Optional<String> name = Optional.of("John");
name.ifPresent(System.out::println);
```

15. JAVA 8+ FEATURES

15.5 Method References

- Use :: to call methods.

Example:

```
list.forEach(System.out::println);
```

15.6 forEach and filter

- forEach – loop through each item
- filter – choose only some items

Example:

```
names.stream().filter(n →
n.startsWith("A")).forEach(System.out::println);
```

16. GENERICS

- Generics allow you to create classes, methods, and interfaces that work with any data type without losing type safety.

16.1 Why Generics?

- Generics help to reuse the same code for different data types without worrying about type errors.
- For example, if you want a list to store numbers, you use "List<Integer>".

16.2 Generic Classes and Methods

- Generic Class:

You can create a class that works with any type.

Example:

```
class Box<T> { // T is a placeholder for any type
  private T value;

  public void set(T value) {
    this.value = value;
  }

  public T get() {
    return value;
  }
}
```

- Usage:

Example:

```
Box<Integer> integerBox = new Box<>();
integerBox.set(100);
System.out.println(integerBox.get()); // 100
```

- Generic Methods:

A method can also be generic to accept different data types.

Example:

```
public <T> void print(T value) {
  System.out.println(value);
}
```

16. GENERICS

16.3 Bounded Type Parameters

- You can restrict the types to specific classes.

Example:

```
class Box<T> { // T is a placeholder for any type
  private T value;

  public void set(T value) {
    this.value = value;
  }

  public T get() {
    return value;
  }
}
```

16.4 Wildcards (<?>, <? extends>, <? super>)

- Wildcards let you use unknown types.
- <?> – Any type

Example:

```
public void printList(List<?> list) { }
```

- <? extends T> – Any type that is a subclass of T

Example:

```
public void printNumbers(List<? extends Number> list) { }
```

- <? super T> – Any type that is a superclass of T

Example:

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

17. ANNOTATIONS

- Annotations are special labels that add metadata to your code.

17.1 Built-in Annotations

- @Override – tells the program that a method is overriding a parent method.

Example:

```
@Override
public String toString() {
  return "Hello";
}
```

- @Deprecated – marks a method as old or no longer used.

Example:

```
@Deprecated
public void oldMethod() { }
```

- @SuppressWarnings – prevents warnings from appearing for specific code.

17.2 Custom Annotations

- You can create your own annotations.

Example:

```
@interface MyAnnotation {
  String value() default "default";
}

@MyAnnotation(value = "Custom Message")
public class MyClass { }
```

18. WRAPPER CLASSES

- Wrapper classes allow you to treat primitive data types (like int, char) as objects.

18.1 Autoboxing and Unboxing

- Autoboxing – automatically converts primitive types to wrapper objects.

Example:

```
Integer x = 10; // int to Integer (autoboxing)
```

- Unboxing – automatically converts wrapper objects back to primitive types.

Example:

```
int y = x; // Integer to int (unboxing)
```

18.2 Common Methods

- intValue() – returns the int value from an Integer object
- doubleValue() – returns the double value from a Double object

Example:

```
Integer number = 5;
int num = number.intValue();
```

19. ENUMS

- Enums are used to represent fixed sets of constants.

19.1 Declaring and Using Enums

Example:

```
enum Day {
  MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
}

Day today = Day.MONDAY;
```

19.2 Enum Methods

Enums come with built-in methods like:

- values() – returns an array of all enum constants.

Example:

```
Day[] days = Day.values();
```

- ordinal() – returns the position of the constant.

Example:

```
System.out.println(Day.MONDAY.ordinal()); // 0
```

20. DATE AND TIME API

- Java provides special classes to work with dates and times.

20.1 LocalDate, LocalTime, LocalDateTime

- LocalDate – represents only the date (e.g., 2025-05-07)

Example:

```
LocalDate date = LocalDate.now();
```

- LocalTime – represents only the time (e.g., 10:30 AM)

Example:

```
LocalTime time = LocalTime.now();
```

- LocalDateTime – represents both date and time

Example:

```
LocalDateTime dateTime = LocalDateTime.now();
```

20.2 DateTimeFormatter

- You can format dates and times with DateTimeFormatter.

Example:

```
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-
dd");
String formattedDate = LocalDate.now().format(formatter);
System.out.println(formattedDate); // 2025-05-07
```

20.3 Legacy Date Classes (Date, Calendar)

- Date – represents a specific point in time, now outdated.

Example:

```
Date date = new Date();
```

- Calendar – used for calculating date/time (e.g., adding days).

Example:

```
Calendar calendar = Calendar.getInstance();
calendar.add(Calendar.DAY_OF_MONTH, 5);
```

21. JAVA MATH & UTILITY CLASSES

- Java has built-in classes that help with mathematical operations and various utilities for better coding practices.

21.1 Math Class

- The Math class is used for mathematical operations like rounding, square roots, trigonometric functions, etc.

Example:

```
double result = Math.sqrt(16); // Square root of 16
System.out.println(result); // 4.0
```

- Math.pow(a, b) – raises a to the power of b.
- Math.round() – rounds a number to the nearest integer.

21.2 Random Class

- The Random class is used to generate random numbers.

Example:

```
Random random = new Random();
int randomNumber = random.nextInt(100); // Generates a random number
between 0 and 99
System.out.println(randomNumber);
```

21.3 Arrays Utility Class

- The Arrays class helps with operations on arrays, such as sorting and searching.

Example:

```
int[] numbers = {1, 5, 3, 2};
Arrays.sort(numbers); // Sorts the array
System.out.println(Arrays.toString(numbers)); // [1, 2, 3, 5]
```

21.4 Collections Utility Class

- The Collections class is used for manipulating collections like lists and sets.

Example:

```
List<Integer> numbers = Arrays.asList(5, 2, 8);
Collections.sort(numbers); // Sorts the list
System.out.println(numbers); // [2, 5, 8]
```

22. BASIC GUI (OPTIONAL)

- Creating a Graphical User Interface (GUI) allows you to build applications with windows, buttons, text fields, and other interactive elements.

22.1 Swing Components Overview

Swing is a popular library for creating GUI applications in Java. It provides components like:

- JFrame – represents a window.
- JButton – represents a button.
- JLabel – represents a text label.
- JTextField – represents a text input field.

- Example of a simple window with a button:

Example:

```java
import javax.swing.*;

public class MyFrame {
  public static void main(String[] args) {
    JFrame frame = new JFrame("My First GUI");
    JButton button = new JButton("Click Me");
    frame.add(button);
    frame.setSize(300, 200);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setVisible(true);
  }
}
```

22.2 AWT Basics

- AWT (Abstract Window Toolkit) is an older toolkit used for GUI creation. It includes components like buttons, labels, and text areas.

Example:

```java
import java.awt.*;
import java.awt.event.*;

public class MyAWTApp {
  public static void main(String[] args) {
    Frame frame = new Frame("AWT Example");
    Button button = new Button("Click Me");
    button.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        System.out.println("Button Clicked!");
      }
    });
    frame.add(button);
    frame.setSize(300, 200);
    frame.setVisible(true);
  }
}
```

23. JAVA BEST PRACTICES

- Following best practices helps you write clean, efficient, and maintainable code.

23.1 Naming Conventions

Java has specific naming rules:

- Classes: Start with an uppercase letter, e.g., MyClass.
- Variables: Start with a lowercase letter, e.g., myVariable.
- Constants: Use all uppercase letters, e.g., MAX_SIZE.
- Methods: Start with a lowercase letter, e.g., calculateTotal().

23.2 Code Optimization Tips

- Use loops efficiently, avoid unnecessary calculations.
- Use StringBuilder for string concatenation to improve performance.
- Always use final for variables that should not change.

23.3 Clean Coding Principles

- Meaningful variable names – e.g., totalAmount instead of a.
- Comment only when necessary – code should be self-explanatory.
- Avoid magic numbers – use constants instead, e.g., final int MAX_STUDENTS = 30;

24. MINI PROJECTS

- Building small projects is a great way to practice Java and improve your skills.

24.1 Calculator

- A simple console-based calculator app that performs operations like addition, subtraction, etc.

Example:

```java
public class Calculator {
  public static void main(String[] args) {
    double a = 5, b = 3;
    System.out.println("Sum: " + (a + b));
  }
}
```

24.2 ATM Simulation

- A simulation of an ATM where the user can check balance, withdraw, and deposit money.

Example:

```java
public class ATM {
  private double balance = 1000;

  public void deposit(double amount) {
    balance += amount;
  }

  public void withdraw(double amount) {
    if (amount <= balance) {
      balance -= amount;
    }
  }

  public double checkBalance() {
    return balance;
  }
}
```

24.3 To-Do List Console App

- A simple To-Do List application where you can add, view, and delete tasks.

24.4 File Encryption Tool

- This tool can encrypt and decrypt files using basic cryptography techniques.

25. JAVA INTERVIEW QUESTIONS (BONUS)

- Here’s a list of common Java interview questions you can prepare for:

25.1 Core Java Concepts

- What is the difference between JVM, JRE, and JDK?
- What are interfaces and abstract classes?
- How does exception handling work in Java?

25.2 OOP-based Questions

- What is inheritance in Java?
- Explain the concept of polymorphism with an example.
- How does encapsulation help in making code more secure?

25.3 Code Snippet Challenges

- Write a program to reverse a string.
- Write a program to find the factorial of a number.
- Write a program to sort an array without using built-in methods.