\*\*Structured Java Notes Document Outline\*\*

---

### \*\*Table of Contents\*\*

1. \*\*Introduction\*\*

1.1 What is Java?

1.2 Features of Java

1.3 Object Orientation

1.4 Association (Has-A Relationship)

1.5 Java Compilation

1.6 WORA Architecture

1.7 JVM, JRE, JDK

2. \*\*Class & Object\*\*

2.1 Class Definition

2.2 Object Creation & Initialization

2.3 Heap Memory & Object References

3. \*\*Data Types\*\*

3.1 Primitive Data Types

3.2 Non-Primitive Data Types

4. \*\*Methods\*\*

4.1 Instance Methods

4.2 Rules for Methods

4.3 Method Invocation

5. \*\*Class Loading\*\*

5.1 Process Overview

5.2 Program Execution Flow

6. \*\*Variables\*\*

6.1 Global vs. Local Variables

6.2 Static vs. Instance Variables

6.3 Variable Shadowing

7. \*\*Constructors\*\*

7.1 Default vs. Parameterized Constructors

7.2 Constructor Overloading

8. \*\*Inheritance\*\*

8.1 Types of Inheritance

8.2 Super Keyword

8.3 Method Overriding

9. \*\*Polymorphism\*\*

9.1 Compile-Time vs. Runtime Polymorphism

9.2 Overloading vs. Overriding

10. \*\*Type Casting\*\*

10.1 Primitive Casting (Widening/Narrowing)

10.2 Non-Primitive Casting (Up/Downcasting)

11. \*\*Garbage Collection\*\*

11.1 Memory Management

11.2 `finalize()` and `gc()` Methods

12. \*\*Wrapper Classes\*\*

12.1 AutoBoxing & Unboxing

13. \*\*Packages\*\*

13.1 Built-in vs. User-Defined Packages

13.2 Access Modifiers

14. \*\*Encapsulation\*\*

14.1 Java Bean Guidelines

15. \*\*Abstraction\*\*

15.1 Abstract Classes & Methods

15.2 Interfaces

16. \*\*Collection Framework\*\*

16.1 Lists, Sets, Queues

16.2 Generics

---

### \*\*Chapter 1: Introduction\*\*

#### \*\*1.1 What is Java?\*\*

- Object-oriented language developed by James Gosling (Sun Microsystems, 1995).

- Key features: platform independence, robustness, simplicity.

- WORA (Write Once, Run Anywhere) architecture enabled by JVM.

#### \*\*1.2 Features of Java\*\*

- \*\*Simple\*\*: No pointers or operator overloading.

- \*\*Portable\*\*: Platform-independent bytecode.

- \*\*Object-Oriented\*\*: Everything is an object.

- \*\*Robust\*\*: Strong memory management and error checking.

#### \*\*1.3 Object Orientation\*\*

- \*\*Object\*\*: Real-world entity with state (data) and behavior (methods).

- \*\*Class\*\*: Blueprint for creating objects.

#### \*\*1.4 Association (Has-A Relationship)\*\*

- \*\*Composition\*\*: Strong association (e.g., Plant and Leaves).

- \*\*Aggregation\*\*: Weak association (e.g., Driver and Car).

---

### \*\*Chapter 2: Class & Object\*\*

#### \*\*2.1 Class Definition\*\*

- Logical entity used to create objects.

- Example:

```java

class Bike {

String color = "blue";

Bike b1 = new Bike();

}

```

#### \*\*2.2 Object Initialization\*\*

- Direct initialization:

```java

Bike b1 = new Bike();

b1.color = "Orange";

```

---

### \*\*Chapter 3: Data Types\*\*

#### \*\*3.1 Primitive Data Types\*\*

| Type | Size | Range |

|---------|---------|------------------------|

| `byte` | 1 byte | -128 to 127 |

| `int` | 4 bytes | -2^31 to 2^31-1 |

| `double`| 8 bytes | Double-precision float |

#### \*\*3.2 Non-Primitive Types\*\*

- Classes, Interfaces, Arrays (e.g., `String`, `ArrayList`).

---

### \*\*Chapter 8: Inheritance\*\*

#### \*\*8.1 Types of Inheritance\*\*

1. \*\*Single-Level\*\*:

```java

class Animal {}

class Dog extends Animal {}

```

2. \*\*Multi-Level\*\*:

```java

class Animal → Dog → BabyDog

```

3. \*\*Hierarchical\*\*:

```java

class Animal → Dog, Cat

```

#### \*\*8.2 Method Overriding\*\*

- Subclass-specific implementation:

```java

class Bank {

int getRate() { return 0; }

}

class SBI extends Bank {

int getRate() { return 8; } // Overridden

}

```

---

### \*\*Chapter 15: Abstraction\*\*

#### \*\*15.1 Abstract Classes\*\*

- Incomplete class with abstract/concrete methods:

```java

abstract class Bike {

abstract void run();

void changeGear() { /\* concrete method \*/ }

}

```

#### \*\*15.2 Interfaces\*\*

- Rule repository with abstract methods:

```java

interface Iswitch {

void switchOn();

void switchOff();

}

```

---

### \*\*Chapter 19: Collection Framework\*\*

#### \*\*19.1 Key Interfaces\*\*

- \*\*List\*\*: Ordered, allows duplicates (`ArrayList`, `LinkedList`).

- \*\*Set\*\*: No duplicates (`HashSet`, `TreeSet`).

- \*\*Queue\*\*: FIFO order (`PriorityQueue`).

#### \*\*19.2 Generics\*\*

- Enforce type safety:

```java

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

list.add("Java");

```

---

**Class:**

**Allowed Inside a Java Class:**

Fields (Member Variables):

Instance variables (non-static) and static variables.

Example: private int age; or public static final double PI = 3.14;.

Methods:

Instance methods (e.g., void setName(String name) { ... }).

Static methods (e.g., static void printMessage() { ... }).

Abstract methods (if the class is abstract).

Constructors:

Special methods for object initialization (e.g., public MyClass() { ... }).

Initializer Blocks:

Static blocks: static { ... } (executed once when the class is loaded).

Instance blocks: { ... } (executed each time an object is created).

Nested Types:

Inner classes, static nested classes, interfaces, and enums.

Example: class InnerClass { ... } or enum Day { MON, TUE }.

Modifiers:

Access modifiers (public, private, protected).

Non-access modifiers (static, final, transient, volatile, etc.).

Annotations:

Applied to the class or its members (e.g., @Override).

**NOT Allowed Inside a Java Class:**

Package Declarations:

package com.example; must be at the top of the file, outside the class.

Import Statements:

import java.util.List; must appear after the package declaration but outside the class.

Top-Level Classes:

A .java file can only have one public top-level class (with the same name as the file). Additional classes must be nested.

Freestanding Code:

Executable statements (e.g., System.out.println("Hello");) must be inside a method, constructor, or block.

Duplicate Member Names:

Fields/methods cannot have the same name and signature in the same scope (overloading is allowed for methods).

Multiple Inheritance:

A class cannot extend multiple classes (e.g., class A extends B, C { ... } is invalid).

Invalid Modifiers:

Variables cannot be abstract or synchronized.

Non-static inner classes cannot declare static methods (except static final constants).

Nested Methods:

Methods cannot be defined inside other methods.

**Static variable:**

Can Do Cannot Do

Share data across instances Use this to access static variables

Access via class name (ClassName.var) Declare inside methods

Initialize with static blocks Override in subclasses (polymorphically)

Use as constants (static final) Serialize with object state

Use in static/non-static methods Assume thread safety without synchronization

**Instance Variable:**

In Java, instance variables (non-static fields) are tied to individual objects and define their state. Here’s a breakdown of what you can and cannot do with them:

What You CAN Do with Instance Variables:

Declare with Access Modifiers

Use public, private, protected, or default (package-private) access.

public class Person {

private String name; // Private instance variable

protected int age; // Protected instance variable}

Initialize at Declaration or in Constructors

Assign values directly or via constructors:

public class Car {

int speed = 0; // Initialized at declaration

String model;

Car(String model) {

this.model = model; // Initialized in constructor

}}

Access and Modify via Object Instances

Use object references to read or change their values:

Person p = new Person();

p.name = "Alice"; // ModifySystem.out.println(p.age); // Access

Define Object State

Instance variables hold the unique state of each object.

Car car1 = new Car("Tesla"); // car1.speed = 0Car car2 = new Car("BMW"); // car2.speed = 0

Use in Instance Methods/Constructors/Blocks

Instance methods and blocks can directly access them:

public class Student {

String name;

void printName() {

System.out.println(name); // Valid in instance method

}}

Inherit in Subclasses

Subclasses inherit non-private instance variables (subject to access modifiers):

class Animal {

protected String species;}class Dog extends Animal {

void bark() {

System.out.println(species); // Inherited

}}

Assign Default Values

If not initialized, they get default values:

int → 0, boolean → false, Object → null, etc.

Mark as final

Declare constants for objects:

class Circle {

final double PI = 3.14; // Constant for all instances}

What You CANNOT Do with Instance Variables:

Access Directly in Static Methods

Static methods cannot use instance variables without an object:

class Test {

int x = 10;

static void print() {

System.out.println(x); // Compilation error

}}

Declare Inside Methods/Blocks

Variables inside methods/blocks are local, not instance variables:

class Test {

void method() {

int localVar = 5; // Local variable (not an instance variable)

}}

Override in Subclasses

Instance variables are hidden, not overridden, in subclasses:

class Parent { int x = 10; }class Child extends Parent { int x = 20; } // Hides Parent.x

Use in Static Contexts Without an Instance

Static blocks/methods require an object to access instance variables:

class Test {

int x = 10;

static { System.out.println(x); } // Error: no instance}

Use abstract or synchronized Modifiers

These modifiers are invalid for variables:

abstract int x; // Compilation errorsynchronized int y; // Compilation error

Share Across Objects

Each object has its own copy of instance variables:

Car car1 = new Car();Car car2 = new Car();

car1.speed = 50; // car2.speed remains 0

Declare in Interfaces

Interfaces only allow public static final constants:

interface Vehicle {

int wheels = 4; // Implicitly public static final}

Use Block Scope

They are class-level and exist throughout the object’s lifetime.

class Test {

{ int x = 5; } // Instance block variable (still class-level)}