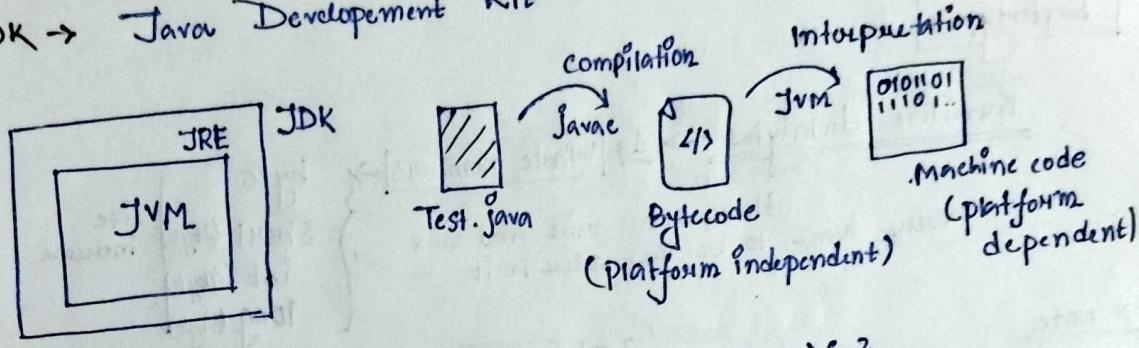
JAVA

JDK → Java Developement kit



Shortcut → main → Public void main (String [] args) { }
Sou → System.out.println (" ")

Everything in Java is object-oriented so
everything in Java will be made or created under a class.

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

main → The main method is the entry point of a Program.

Public → Access modifier indicating that the method can be accessed from outside the class.

Static → Indicates that the the method belongs to the class rather than an instance of the class.

void → Specifies that the method does not return any value.

String [] args → The method accepts an array of strings as parameters. This is where command-line arguments can be passed to your Program.

System → A class in the java.lang package that provides access to the system, including the console

out → An instance of printstream class within the System class, representing the standard output stream.

Println → A method used to print a line of text to the console.

① Datatypes

Primitive datatypes :→ 1. Whole numbers → byte, short (2byte), int (4byte), long (8byte) — size increase

Have some min and max value have to be the value in it

2. decimal numbers → float (4byte), double (8bytes)

3. Characters (2bytes)

4. Booleans (1bit)

* Note

Should write 'f' for float and 'L' for long at the end of the value.

float num = 13.56789f

long num = 1234789098L

Should be written between single quotes.

7 precision number after decimal.

15 precision numbers.

{From 0 to 127 there are → ASCII values.}

Widening Conversion

Automated conversion of one data type to another. It is also called Implesite conversion.

② Variable Name rules

1. Java is case sensetive.
2. Can be letters, digitals, dollar sign or underscores.
3. Can not start with digits.
4. Can not use Java keywords as variable names.
5. Camel case (Every starting letter will be in capital)

3 Arithmatic Operations

```
int Salary = 10000;
    bonus = 500;
    deduction = 100;
```

Yearly Salary = (10000 + 500 - 100) × 12 will be written as.

Yearly Salary = (Salary + bonus - deduction) * 12

Compound assign operator →

```
byte a = 10;
a += 5;  or  a *= 5;
System.out.println(a);
```

a = a+5
a = a*5

⇒ 15 or 50 → Will be saved in bytes.

Note

Year salary / 3 then [alt + enter] will be provided local variable automatically.

```
int p = Yearsalary/3
```

Increament or decreament operator →

++a
Pre increament operator

First increament then use.

a++
post increament operator.

First used then increament.

```
int a = 1;
int b = ++a + a;
System.out.println(b);
```

(++a: 2, a: 2)

⇒ 4

```
int a = 99
int b = a++;
System.out.println(a);
System.out.println(b);
```

Then increament

First used

⇒ 100
99

c = (a) + (++a)

(a: 1, ++a: 2)

will be solved left to right for Precidence rule.

```
System.out.println(c);
```

⇒ 3

Bit-wise Operator. ④

Operands → byte, short, long, int.

Operator → and operator &
or operator |
xor operator ^
not operator ~
left shift <<
right shift >>
unsigned right shift >>>

```
int c = 5 & 4
System.out.println(c);
```

5 = 101
4 = 100
100 ← multiply above digit to under.

⇒ 4 ← and operator = 4

```
int a = 5
System.out.println(a);
System.out.println(Integer.toBinaryString(a));
int b = (~a);
System.out.println(Integer.toBinaryString(b));
System.out.println(b);
```

Using not operator.

⇒ 5 ← As it is.
101 ← Binary form of 5
111010 ← change all 1 to 0 and 0 to 1 upto 32 bit
-6 ← Integer form of the digit.

if b = a << ;

Left shift.

101 ← 5 (Integer)
1010 ← shift by one bit to the left
10 (Integer)

Right Shift (Unsigned)

Shift the bit and then fill the left most bit with zeros.

Right Shift

Shift the bit and then fill the left most bit with one.

Printing ⑤

```
System.out.println("Hello, World");
```

System is a class. which have system related utility with method which interacts with system's run time environment.

out — Static member of system class. which is connected with console.

println("a");
println();
println("b");

→ a
b

⇒ Blank line space created by println().

print("a");
print();
print("b");

← No such thing exists.

→ ab No line gaps.


```
int a = 1;
int b = 2;
String c = "Sum";

System.out.println(c + "of" + a + "&" + b + ":" + (a+b));

System.out.printf("%s of %d & %d : %d", c, a, b, a+b);
```

⇒ Sum of 1 & 2 : 3
Sum of 1 & 2 : 3.

String data type ⑥

```
char c = 'a';
String name = "abcd";
```

String is a sequence of character. It is not a Primetive datatype it is a class.

storing multiple characters.

Another method to write string

```
String adress = new string (original:"India");
```

A new memory (Heap) will be allocated for this string in this case.

So here 'adress' variable will store the memory adress for the new string storing value India.

```
class student {
    String name;
    String add;
    int roll no;
    int standard;
}
```

class is a blueprint which displays how everything will look like

String methods

```
# String name = "Ram";
  int length = name.length();
  System.out.println(length);
```

⇒ 3

```
# String name2 = "Ram";
  System.out.println(name.equals(name2));
```

⇒ True

```
# Str = " "
```

no character but has some spaces.

- `System.out.println(str.isEmpty());` — give false counts the spaces.
- `System.out.println(str.isBlank());` — will give true not considers spaces.

every methods will suggested after typing this

```
name.
name.length()
```

↳ Alt + Enter

```
int length = name.length()
```

variable will be provided automatically, after pressing those keys.

Conditional Statements (7)

Relational operators → Compare two values and return boolean value. { >, <, >=, <=, !=, == }

Logical operators → Combine or match two values.

{ && Logical and
|| Logical or
! Logical not }

```
int age = 20;
    if (age >= 18) {
            S.O.u ("you are an adult");
        }
    else { Sou ("Not an adult")
           }
```

{ if
else if
else if
:
else

* To Scan and take an input value.

i) Import java.util.Scanner;
ii) Scannar sc = new Scanner (System.in)
iii) int number = sc.nextInt ()
iv) sc.close ();

(8) Switch case → cannot be used in double and float.

integer.

```
Switch (day) {
case 1: {
        Sou ("Monday");
        break; }

case 2: {
        S.O.u ("Tuesday");
        break; }

case 3: {
        So.u ("wednesday");
        break; }

default : {
        S.O.u ("Nothing");
    }
```

Loop ⑨

While loop →

```
int i=0;
while (i>0) {
    S.o.u ("Hello world");
    i = i+1;
}
```

condition

changes.

do-while loop →

```
do {
    Action
}
while (condition);
```

First action then check the condition.

For →

```
for (Initialization ; condition ; update ) {
    Action ;
}
```

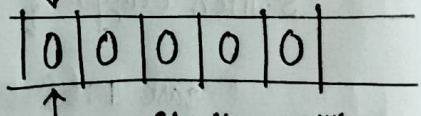
ARRAY ⑩

```
// Int [] array = {1, 2, 3};
// Int [] Specialarray = new int [10]
```

Specialarray ← Reference variable

new int [10] ← Allocating new memory for array.

Array object for 5 integer in heap memory.



Stack memory Specialarray

```
// int arr[] = new int [10]
for (int i=0 ; i <= arr.length ; i++) {
    S.o.u (arr [i]);
}
```

Assigning →

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

Loop works only for Arrays →

```
for (int i : arr1) {
    S.O.U (i);
}
```

Integer type.

Finding MAX value →

```
int [] arr2 = { 3, 92, 42, 33, 2 }
int sus = Integer.MIN_VALUE
```

(-∞ in java, lowest value in math)

```
for (int i=0; i<arr.length; i++){
    if (arr[i] > sus) {
        sus = arr[i];
    }
} S.O.U (sus);
}}
```

2D ARRAY

```
int [] [] arr = new int [3] [3]
```

0	0	0
0	0	0
0	0	0

```
int [] [] num = { {1,2,3}, {4,5,6},
                  {7,8,9} }
for (int i=0; i<num.length; i++) {
    for (int j=0; j<num.length; j++) {
        S.O.U (num [i] [j] + " ");
    }
    S.O.U ();
}
```

OOPS (11)

A style of programming, dealing with Class and Objects.

Blue-print

Properties, Behaviours

Creating a car blueprint:-

Car.java

```
Public class Car {
    String colour;
    String brand;
    String model;
    int year;
    int speed;

Public void acceleration (int increament) {
    Speed += increament;
}
Public void break (int decreament) {
    Speed -= decreament;
    if (speed < 0) {
        Speed = 0;
    }
}
```

Properties of an object

Behaviour of the car on the object car, explained through methods.

Test.java ← use the blueprint (Here Car) and make a real car out of it.

```
Package Test;
    Public class Test {
Public Static void main (String[], args) {
    Car car = new Car();
    car.colour = "Blue";
    car.Speed = 40;
    car.Brand = "Tata";
    car.acceleration (1);
    S.O.U (car.Speed);
} }
```

Starting point for Java

Class name or the provided blueprint

Reference variable, point to the newly created car object in the memory

constructor call the default one.

OOPs ⟶ Class and Objects.

- → Encapsulation → Binding of data & methods into a single unit.
- → Inheritance → Allows one class to inherit properties and behaviour from different class.
- → Polymorphism →
- → Abstraction →

* Other writing styles apart from oops → (i) Imperative (one by one)
(ii) Functional (Treating fn as variable)
(iii) Descriptive

Methods (12)

Return type: void
Method name: main

```
Public Static void main (String[] args) {
```

Public → access modifier.
String[] args → Parameter.

```
// method body
}
```

```
* Private Static int Sum (int a, int b){
      return a+b;
  }
  Private Static int Sum (int a, int b, int c) {
      return a+b+c;
  }
```

Method names are same but Parameters are different, called method overload.

Method Signature → Method name (parameter)

Variable numbers of argument →

```
Public class Test {
    Public static void main (String[] args){
System.out.println (sum (1,2));
}
Public static int Sum (int ... a){
    int Sum = 0;
      for (int i : a) {
        Sum += i;
      }
        return Sum;
      }
```

Recursion (13)

```
Public static void main (String [] args) {
    S.O.U (factorial (5));
    // 5.4.3.2.1
    // 5.4!
}
Public static int factorial (int n) {
      if (n==1) {
        return 1;
      }
    return n * factorial (n-1);
    }
```

Call Stack ← A part of memory which track the method whenever it is called

main()	f(5) main()	f(4) f(5) main()	f(3) f(4) f(5) main()	f(2) f(3) f(4) f(5) main()	f(1) f(2) f(3) f(4) f(5) main()

Package → It is a folder which contains organized classes. A package can't contains classes with same names. It improves reusability.

ex → Java.lang → contains core classes like String, Math, System.

Creating and using packages →

1. creating a new package : package test.
2. Place all related classes in the same package.
3. To use classes from another package:
   - use import statements:

     import test.Myclass;
   - Or use fully qualified names:

     test.Myclass obj = new test.Myclass();

Class path →

i) class path tells the JVM where to find class files.
ii) JVM searches for .class file according to package structure.
iii) Incorrect class path leads to "Class Not Found" errors.
iv) Set the class path properly when compiling or running Java Programs.

```
java -d . Myclass.java
java test.Myclass.
```

Package Naming Conventions →

use reverse domain name format

com.companyname.projectname.module.

Public classes → accessable from any package
Non-public → accessible only within the same package
Nested classes → follow specific access rules

Encapsulation (14)

This property allow to not change any properties directly. We should have get() and Set() method to acess any properties with civilian condition and keep the data private.

```
int roll number;
Private int age;

Public void SetAge (int age){
    if (age <0){
      age =0;
    }
    this. age = age;
}

Public int getAge (){
    return this. age;
}

Public Static void main (String[] args){
    Student student = new student ();
    Student . name = "Ram";      X
    Student . roll number = -1;  X
    Student . SetAge (20);
    S.O.U ( student . get Age ());
}
```

method to acess data

method to acess data

✓

* Shortcut in ide
  Right click and generate getter setter for every variable.

Constructor (15)

```
Public class Test {
    Public static void main (String [] args) {
        Student student = new student ();
```

For default constructor ⇒

class name ← student () → Syntax like a method

Student () is a constructor & method which initialise the object.

* Constructor has same name as the class.
* It sets the default value to all the attributes in the class.
* Constructor doesnot has any return type

For paramaterized constructor ⇒

```
Public student (String name, int rollNumber, int age) {
    this. age = age;
    this. name = name;
    this . rollnumber = rollnumber;
```

Driver class →

```
Student student = new Student (Arijit, 34, 22);
```

Constructor overloading → Methods with same name but different constructors Signatures.

Inheritance (16)

Parent class —(inherits: attributes, methods.)→ Child class

```
Public class Animal {
    Public String name;
    Public int age;
    Public void Sound () {
        S.O.U ("Bark");
    }
}
```

(Parent class) — Attributes — Method.

```
Public class Dog extends Animal {
}
```
Child class inheritance from Parent

```
Public static void main (String [] args) {
    Dog dog = new Dog();
    dog.name = "Sheru";
    dog.age = 21;
    dog.sound();
}
```
Driver class

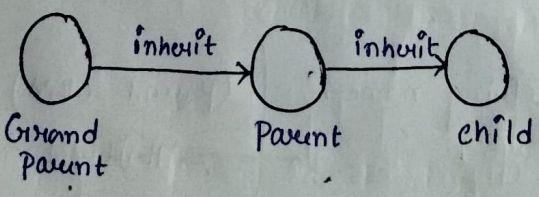
create obj of dog

use methods and names, age which is not in Dog class but is in it's parent class.

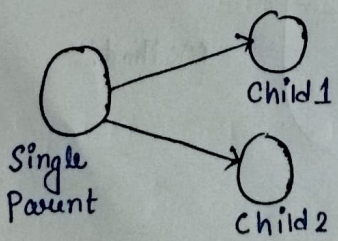
Method overriding (17)

What if the child acquires all properties from it's parent but change or modify one property. or methods which has to be same in name but ~~different~~ signature. should be same means along with the name the parameter and return type will be same. The body will be different otherwise it is useless. we can change the access modifier too.

```
Public class Dog extends Animal {
    Public void sound() {
        S.o.u ("woof");
    }
}
```
method overriding.



Multilevel inheritance.



Hierchial inheritance.

Super Keyword (12)

The Super keyword in Java is used to refer to the immediate parent (superclass) object. It is mainly used when a child class wants to access parent class variables, methods or constructors.

1. Access parent class variable →

When parent and child class have the same variable name Super keyword is used to avoid confusion.

```
class parent {
        int x = 10;
    }
class child extends parent {
        int x = 20;
void show() {
        System.out.println (super.x);  // Prints parent variable
    }
Public Static void main (String[] args) {
        child c = new child();
        c.show();
    }
```

Output = 10

2. Call parent class Methods →

used when the child class overrides a method and still wants to call the parent version.

```
class parent {
    void display() {
            S.O.U ("parent method")
    } }
```

```
class child extends parent {
    void display() {
    super.display(); //calls parent method
     S.O.U ("child method");
    }
Public static void main (string [] args ) {
      child c = new child();
      c. display();
      }
```

Output → Parent method
child method

3. Call parent class constructor.

Super() is used to call the constructor of the parent class.
it must be the first statement in the child constructor.

```
class parent {
    parent() {
        S.O.U ("parent constructor") }
    }
class child extends parent {
    child() {
        super(); //calls parent constructor
     S.O.U ("child constructor"); }
Public static void main (string [] args ) {
        child c = new child();
    }
}
```

Output → Parent constructor
Child constructor.

- Super() must be the first line in constructor
- Cannot be used in a Static context.

@Override (10)

@Override is used to ensure that a subclass method correctly overrides a superclass or interface method and allows the compiler to catch errors early.

```
class parent {
    void display() { }
}
class child extends parent {
    @Override
    void display() { }
}
```

this will ensure that display() should be exact same of parents display() No new method will be created.

Multiple inheritance

Java doesn't support multiple inheritance using classes.

Diamond problem

- If two parent classes have the same method, the child class gets confused about which method to inherit.
- This leads to ambiguity at compile time.

```
class A {
    void show() {
        S.O.U ("A"); }
}
class B {
    void show() {
        S.O.U ("B"); }
}
class C extends A, B {
}
```

X Not allowed.

Confusion will be created upon which show() will be inherited.

Polymorphism

Polymorphism allows method to do different things based on the object it is acting upon, even though the method name and its signature might be the same

Compile time polymorphism.

It takes decision while compiling, Method overloading is the perfect example for this.

```
Class Calculator {
    public int add(int a, int b){
        return a+b;
    }
    Public int add (double a, double b) {
        return a+b;
    }
}
Public static void main (String [] args) {
    Calculator c = new Calculator();
    S.O.U (c.add (3,5));
    S.O.U (c. add (3.1, 5.3));
}
```

Method Overloading

Compile time decision of what to call

Runtime polymorphism

Runtime polymorphism occurs in runtime in method overriding

if there is a class Animal which have two childers named class dog and class Cat.

There of them have same method name Hello(). Overriding will be done in two ways.

```
Public Static void main (String [], args) {
    Animal A = new Animal ();
      A.Hello ();
    dog D = new dog ();
      D. Hello ();
    Cat C = new Cat ();
      C. Hello ();
```

} Method overriding Same method names but will be called according to the class object's it is

Child object will be created from parent's class instance.

```
    Animal D = new dog (); // upcasting
      D. Hello ();
    Animal C = new Cat ();
      C. Hello ();
```

```
    Dog D = new Animal (); X Not possible.
```

```
    Dog D = (Dog) dog ; // Down casting
```

In upcasting only those methods will be called which presents in both class animal and dog. If there any method which is in dog class but not in Animal we can't call it with upcasting.

Abstraction (21)

Abstraction means the internal part will be hidden and only the functions will be visible.

There are two types of abstraction → Abstract class
→ Abstract Method.

An abstract method only will be in an abstract class. But an abstract class can have both normal or Concrete method (Method with body) or abstract class (class without body)

- Abstraction = showing what an object does, not how it does it.
- Internal implementation is hidden
- Only essential features are exposed

Abstraction is needed →
- Reduce complexity
- Improves code readability
- Forces a standard structure

Abstraction can be acheived → Abstract class
→ Interfaces.

Abstract class

- Declaring using 'abstract' Keyword
- Cannot be instantiated
- Can contain →
  - Abstract method
  - Concrete method
  - Constructors
  - Variables.

```
abstract class classmate {
    abstract void method1(); // Abstract method
    void method2() {         // Concrete method.
        S.O.V ("Hello");
    }
}
```

Abstract method

* • Declared without body
  • Ends with semicolon
* • Must be implemented by the Subclass
  • Abstract method cannot be private
  • Abstract method must be overridden.

```
Animal a = new Animal(); // Error
```
Abstract class can't be instantiated

```
Animal a = new Dog();
a.makeSound(); // woof
```
Valid (Polymorphism)

Inherit abstract method

When a class extends abstract class, it is forced to complete the unfinished work left by the abstract class.

```
abstract class Animal{
    abstract void sound();
}
```
Parent abstract class with abstract method

```
class dog extends Animal {
    void sound() {
        S.o.v("woof");
    }
```
Child class which is not an abstract class so should complete the abstract method.

```
abstract class dog extends Animal{
    // not implement the method
}
```
As the child is abstract too it does not need to complete the abstract method of this parent.

Constructors in Abstract class

- Abstract class can have constructors
- used to initialize variables
- Should usually be protected.

```
abstract class Vehicle {
    Protected Vehicle() {
        S.O.U ("Vichle created");
    }
}
```

* Private access modifier can not be used in Abstraction.

Acess modifier (22)

Acess modifiers decide where a class, method, field, or constructor can be accessed from.

Types →
- Public → Accessible everywhere.
- Private → accessible only inside the same class
- Protected → Same package + Subclass
- deafault → Same package only (NO keywords)

Rules →
- Top level classes → only public or default.
- Methods & variables → can use all modifiers.
- Private members → Never visible outside the class
- Protected → mainly for Inheritance
- If no modifier → default access.

```
Package accss;
    Public class Student {
        Public String name;  //accessible anywhere
        Protected int age;  // Same package + Subclasses
            String course;  // default (Package-private)
        Private int marks;  // only inside this class.
    }
```

Accessing from another package →

```
Package test;
        import access.Student;  // 'Student' class from
                                   'accss' package
Public class Test {
    Public static void main (String[] args) {
        Student s = new student();
        s.name = "Alex";  // ✓ public
        s.age = 20;  // x Protected (not Subclass)
        s.course = "CS";  // x default
        s.marks = 90;  // x Private.
    }
}
```

Protected + Inheritance →

```
Package Animal;
    Public class Animal {  // Animal class
        Protected String Sound;  // with one
                                    Protected
                                    variable
    Protected void makeSound() {   and
                                    method
        System.out.println(Sound);
    }
}
```

```
Package dogs;                    // Even it is a different
                                    Package but there is
import animals.animal;              Inheritance.

Public class Dog extends Animal {
    Public Dog() {
        Sound = "Bark";          // ✓ protected accessible via
    }                               inheritance

    Public void Speak() {
        makeSound();             // ✓ protected method
    }
}
```

✗ Not accessible from non-subclass outside package.

Private constructor + static Methods

- Prevent object creation
- Force class-level usage only

```
Public class MathUtils {
    Private MathUtils() {
        // Prevent object creation.
    }
    Public static int add(int a, int b) {
        return a+b;
    }
}

int sum = MathUtils(3,5);
```

Singleton Design Pattern

Only one object for the entire application.

- Private constructor
- Private static instance
- Public static getter method

```
Public class Singleton {

    Private static Singleton Instance; // Single Shared Instance
    Private Singleton() { // Private constructor
    }
        // Global access point
    Public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton(); // created only once.
        }
        return instance;
    }
}
```

```
Singleton S1 = singleton.getInstance();
Singleton S2 = singleton.getInstance();

S.O.U (S1 == S2); // true, same object.
```

Uses

- Database connections
- Logger
- Configuration manager
- Cache

* Overuse can create hidden global states.

Static Keyword (23)

what does static means →

The static keyword makes a variable, method or block belong to the class itself not to individual objects.

- Static members are loaded once, when the class is loaded by JVM.
- All objects of the class share the same static member.
- Use static → when something is common or global.

Static Vs Instance

Aspect	Static	Instance.
Belongs to	class	object
Memory	one shared copy	Seperate copy per object.
Acess	ClassName.member	Object.member.
Need obj	✗ NO	✓ Yes

* Instance methods can aces static members. But static methods cannot directly access instance members.

Static variables

- Static variables store data that must be shared accross all objects.
- only one copy exists in memory.

```
class Student {
        static int count = 0;
        Student() {
            count ++; }
    }
Student s1 = new student();
Student s2 = new student();

S.O.u (student.count) // 2
```

Instance Variable

- Instance variables represent object specified state
- Each object gets its own copy.
- They are used when values differ per object.
- Access required object creation.

Static Methods

- Static methods belong to the class, not objects.
- They can be called without creating objects.
- They cannot use [this] or [super] and cannot access instance members directly.
- Best suited for utility or helper logic.

```
Class MathUtils {
    Static int max (int a, int b) {
        return a>b ? a : b;
    }
}

S.O.U (MathUtils.max(5,10)); // 10
```

↑ Accessed without creating any object.

Instance Method

- Instance methods operates on object data.
- They can acces both Static and non-static members.
- Used when behaviour depends on object state.
- Supports inheritance and polymorphism.

Why (main) method is static

The JVM starts execution without creating an object.

Making (main) Static allows JVM to call it directly using the class name if (main) were non-static, JVM wouldn't know how to start the program.

Static Blocks

- Static blocks execute once, when the class is loaded.
- They run before any object is created.
- used for complex initialization that cannot be done in one line.

```
class config {
    static int value;
    static {
        value = 100;
        S.O.U ("static block executed");
    }
}
```

Utility Classes

- Utility classes contains only static methods.
- They are not meant to be instantiated.
- often have a private constructor to prevent object creation.

```
class Utils {
    Private Utils () { }
    static int square (int x) {
        return x*x;
    }
}
```

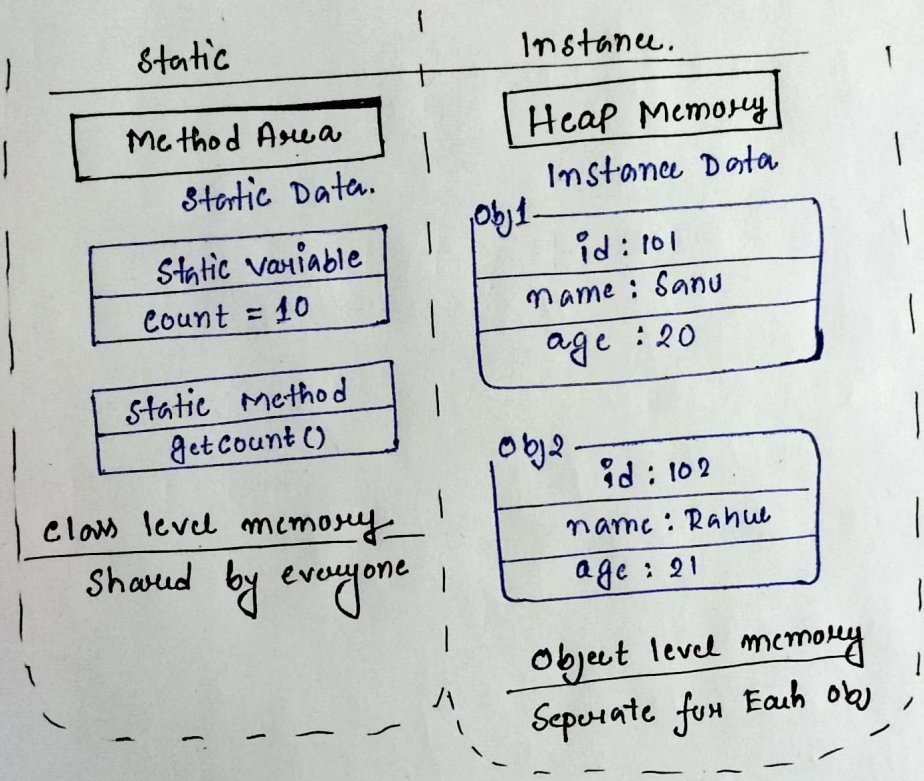
Singleton pattern (Static use)

- Ensure only one object exists in the application.
- uses a static variable to hold the single instance.
- A static method provides global access.

```
class Database {
        Private static Database instance =
                        new Database();
    Private Database () { }
  Public static Database getInstance () {
        return instance;
    }
}
```

Memory prespective

- Static members are stored in class level memory
- Instance members are stored in the heap per object.
- Static Improves memory efficiency for shared data.
- Overusing static reduces flexibility and testability



Final Keyword (24)

what final means

The final keyword is used to restrict modification.

Depending upon where it's applied, it prevents:

- Reassignment (variables)
- Overriding (methods)
- Inheritance (classes)

Final variables.

- A final variable can be assigned only once.
- After Initialization, it's value cannot be changed
- The compiler guarantees this rule.

```
class Car {
    final int Speed_Limit = 200;
}

Car C = new Car();
C.Speed_Limit = 180; // X Compile time error.
```

- Acts like a constant.
- Prevents accident changes.
- Improves code safety and readability.

Static final variables

When a constant is shared by all objects, use static final.
Only one fixed copy exists at class level.

```
class MathConstants {
    Public static final double PI = 3.14159;
}

S.O.U (MathConstants.PI); // 3.14159
```

Initialization rule.

↳ can be Initialized → at declaration
→ or, inside a static block

Must be Initialized exactly once.

Final Methods

A Final method cannot be overridden by subclasses. used to lock behaviours that must remain unchanged.

```
class car {
    final void airbag () {
        S.O.U ("4 airbags enabled");
    }
}

class EVcar extends car {
    void airbag () {} // X compile time error
}
```

- Prevents unsafe or incorrect behaviour changes.
- Important for security, safety and core logic.

Constructors and finals.

Constructors cannot be declaired final.

- constructors are not inherited
- overriding doesn't apply to constructors.

Final with objects.

Final applies to the reference, not the object itself.

```
final Car car = new car ();
car = new car (); // not allowed X
car.speed = 120; // Allowed ✓
```

- fields must be final.
- no setters.
- controlled construction.

Interference

What is Interface.?

An interface is a structure in Java that defines what a class must do, not how it does it.

It acts like a contract that a class agree to follow.

```
interface Animal {
    void eat();
    void sleep();
}
```

Any classes that implements this interface. must Provide the body of eat() and sleep().

```
class Dog implements Animal {
    Public void eat() {
        S.O.U ("Dog eats");
    }
    Public void sleep() {
        S.O.U ("Dog sleeps");
    }
}
```

An interface cannot be instantiated because it has no implementation.

Why interface exists.?

For Multiple Inheritance.

Java dosnot allow a class to extend multiple classes because it creates ambiguity. But Java allows a class to implement multiple interfaces because interfaces only contain method declarations.

Class Smartphone implements Camera, Phone, Music

This allows one class to have multiple behaviours.

Interface Vs Abstract Class.

Features.	Abstract	Interface.
Variables.	Instance + Static	only Static final.
Methods	Abstract + Concrete	Abstract, default, Static.
Constructors.	Yes	NO
Multiple Inheritance	NO	Yes.
Object creation	NO	NO

Abstract classes represent what something is
Interfaces represent what something can do.

Variable in interfaces.

All variable inside an interfaces are automatically.

Public Static final

So they are constants.

```
interface Limits {
    int max =100;
}
```

They belong to the Interface and cannot be changed.

Default Methods

Default methods allow interfaces to have method implementation

```
interface A {
    default void show() {
        S.O.U ("Default show");
    }
}
```

They were added so that the new methods can be added to interfaces without breaking old codes.

Classes can override default methods if needed.

Multiple Inheritance using Interfaces.

```
interface Camera {
    void takephoto();
}

interface musicplayer {
    void playmusic();
}

Class Smartphone implements Camera, Musicplayer {
    Public void takephoto() {
        S.O.U ("photo taken");
    }
    Public void playmusic() {
        S.O.U ("music play");
    }
}
```

Main method in Interfaces.

Java allows a `main` method inside an interface.

```
interface Test {
    Static void main (String[] args) {
        S.O.U ("Hello");
    }
}
```

This works because main is static and doesn't need an object.

Java inner classes (26)

what is an inner class. ?

⇒ An inner class defined inside another class.

it exists because Sometimes two objects are so closely related that Seperating them into independent classes makes the code messy and harder to understand.

A car and its Engine are not independent concept - an Engine belongs to Car. so Java lets us express this relationship using an inner class.

1. Member Inner class.

A member Inner class is a normal class written inside another class.

- It is tied to an instance of the outer class.
- It can directly access all data of the outer class (even private)
- Every object of the inner class is linked to one Specific outer object.

It models a "belong to" relationship. The Inner object cannot exists without the outer object.

```
class Car {
    Private String model;
    Car (String model){
        this.model = model;
    }

class Engine {
    void start (){
        S.O.U (model + " Engine started"); }
    }
}
```

Both classes are written in same place.

```
Car Car = new Car ("Tata");
Car.Engine engine = Car.new Engine ();
engine.start ();
```

Engine depends on car.
It automatically knows which car it belongs to

2. Static Nested Class.

A static nested class is declared with static inside another class.

- It does not depend on any instance of the outer class.
- It can only access static members of the outer class.
- It behaves like a normal top-level class, just grouped inside another.

It is used when a class is logically related but not physically dependent.

A USB drive is related to a Computer, but it does not belong to a Specific Computer object.

```
Class Computer {
    Static class USB {
        void connect () {
            S.O.U ("USB connected"); }
    }
}

Computer.USB usb = new Computer.USB ();
usb.connect ();
```

This avoids polluting the package with extra files while keeping code organized.

3. Local Inner class.

A local inner class is declared inside a method.

- It exists inside that method
- It can access:
  - fields of the outer class
  - local variables (if they are final or effectively final)

It is used to keep temporary logic close to where it is used.
Instead of turning it into a full class.
It improves encapsulation and prevents accidental reuse.

```
class Hotel {
    int room = 5;
    void reserve() {
        class Validator {
            boolean canReserve() {
                return rooms > 0;
            }
        }
        Validator v = new Validator();
        S.O.U(v.canReserve());
    }
}
```

Validator is meaningfull only inside reserve()
It does not pollute rest of the program.

4. Annonymous Inner class.

An anonymous inner class is created when want to use a class only once.

- No class name.
- We write and instance it in one step
- Mostly used with interfaces and abstract class.

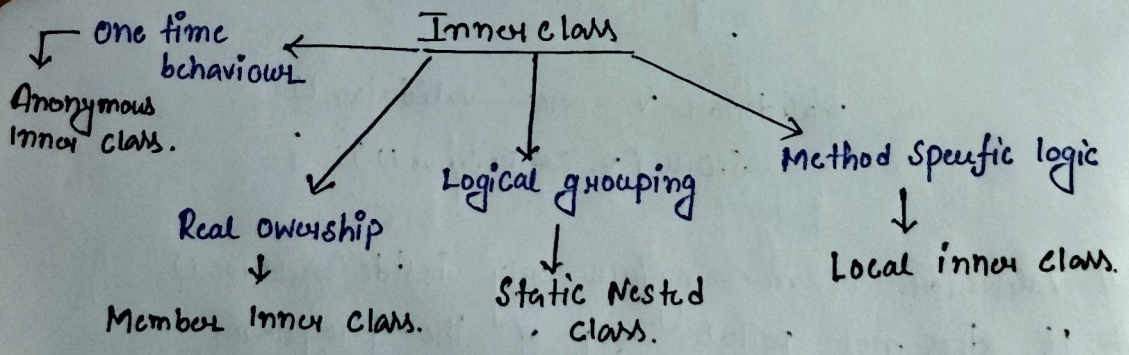
It elements boilerplate when we only need a small implementation once.

Instead of → class creditcard implements payment {...}

It can be written →

```
Interface payment {
    void pay (double amount);
}
Payment p = new payment () {
    Public void pay (double amount) {
        S.O.U ("paid" + amount); }
};
P.pay (500);
```

It is very common in →
- Event handling
- Threads
- Callbacks.



Java Expetion

27

What is an ~~exception~~ Error in Java?

⇒ Java program fails in three different ways.

Type	Meaning
Syntax Error	code is written incorrectly
Logical Error	code runs but gives logical error.
Runtime Error	code runs, compiles but crashes while running.

Syntax errors are like grammar mistakes - Java refuses to run the program. Logical errors are dangerous as the program runs but gives wrong answers. Runtime errors happen while the program is running and usually crash the program unless handled.

Runtime error → `int x = 10/0;` // Arithmatic exception.

What's an Exception

An exception is an object created at runtime that represents an abnormal condition which interrupts the normal execution of a program.

An exception is not just an error message. It is an object created by the JVM.

it contains:

- what went wrong
- where it happens.
- A message describing the problem.

when this happens :→ 30/0;

The JVM internally does :→

throw new ArithmeticException ("/ by zero");

So the JVM throws an object up the call stack.

Why programs crash without handling

```
int [] n = {10, 20, 30};
int [] d = {1, 0, 2};

for (int i : n) {
    S.o.u (n[i] / d[i]);
}
```

Output → 10

Exception in thread "main" java.lang.ArithmeticException : / by zero

The JVM stops execution as soon as it hits an execution. This prevents corrupted or unsafe program execution. without handeling, the program terminates immediately.

Exception Handeling

```
try {
    int x = 30/0;
}
catch (ArithmaticException e) {
    S.o.u (e);
}
```

- try marks code that may fail
- Catch tells Java what to do if it fails.
- Instead of crashing, the program moves into the catch block.

Exception Hierarchy

Object
↓
Throwable
↓
Exception
↓
RuntimeException
↓
ArithmeticException
NullPointerException
IndexOutOfBoundsException

Java orgnize all errors in a hierarchy. If we catch a Parent, we catch all children

Parent class
↓
catch (Exception e)
↑ variable

works for all type of exception

Why S.O.U(e) works

Print(e) internally calls:

e.toString() // This prints ClassName : message

Java.lang.ArithmeticException : /by zero

Every exception inherits (toString()) from Throwable, which formats useful debugging information.

Stack Trace

e.printStackTrace();

Stack trace shows the chain of method calls that led to the execution. it is the most powerful debugging tool in Java.

checked and unchecked Exceptions

Type		
nchecked	Not cheked by compilor.	NullpointorException, ArithmaticException.
ecked	cheked by compiler	IO Exception. FileNot Found Exception.

Unchecked exceptions are programming mistakes.
Checked exceptions are external failures (file, network, database)

Checked exception →

```
FileReader fr = new FileReader ("a.txt");
```

This file doesn't exists, so Java forces us to either handle it or declare it.

Throw vs throws

throw ⟶ Actually throws exception
throws → Declares that method may throw.

```
throw new IOException ("File missing");
void read() throws IOException
```

throw creates an exception.
throws tells caller that this method might fail

finally Block

```
finally {
    S.O.O ("Always runs");
}
```

finally is used for cleanup.
it runs if an exception or return happens.

Try with

```
try (BufferReader br = new BufferReader (new FileReader ("a.txt")
)) {
}
```

Java will automatically close the file.
Prevents memory leaks and crashes.

Custom Exceptions

Class MyRuleException extends Exception { }

Used when business rules fail, not System fail.

Wrapper Class. (28)

Is Java a pure object-oriented Programming Language.

⇒ Java is often marked as an object-oriented language, but that doesn't actually make it pure OOP.

Understanding this distinction is critical, otherwise we will blindly repeat wrong statements.

What's a pure Object oriented

A pure object oriented follows strict rules.

- Every value must be an object.
- No primitive data type.
- All operations must be performed via object methods.
- Everything lives inside classes and objects.

Why not Java?

Java violates the pure OOP defination by design.

it introduces primitive data types like int, float, boolean.

These types →
- are not classes.
- Do not create obj
- Do not have methods.
- Are handled differently by JVM.

Primitive Types Vs Objects

Primitive types exist for performance and memory efficiency.

Objects exists for flexibility, abstraction and behaviour.

Java deliberately Seperates these two worlds instead of forcing everything into objects (which would slow the language down).

int i = 10 has no methods

for int i = 10;

- 'i' holds a raw value.
- NO object is created.
- NO memory allocation happens in the heap
- NO method table exists.

So calling `i.someMethod()` is logically impossible.

String S = "Arijit" has methods

String is a class not a primitive.

String S = "Arijit";

- An obj is created
- Memory is allocated in heap

So → S.length(); } Method possible.
S.getBytes(); }

Why wrapper class exists

Java needed a way to

- Treat primitive values as obj when needed
- use primitives in obj-based systems
- provide utility methods for primitive values.

we wrap primitive value inside obj.

Wrapper class →

int → Integer	char → character	short → Short
float → Float	boolean → Boolean	long → Long
double → Double	byte → Byte	

Each wrapper →
- is a class
- Creates obj
- Has methods
- Can be stored in collections.

Why Collections Don't support primitives.

Collections are designed to store objects, not raw values.

because →
- Collections rely on polymorphism.
- They store references.
- They expect behaviour

Auto boxing (primitive → obj)

Auto-boxing means Java automatically converts a primitive into it's wrapper obj.

```
Integer x = 5;
```

internally →

```
Integer x = Integer.valueOf(5);
```

Unboxing (obj → primitive)

```
int y = x;
↳ int y = x.intValue();
```

Wrapper classes are immutable

```
Integer x = 1;
x = 2;
```

This doesn't modify the object.

- Old object stays unchanged
- New obj is created.
- Reference is reassigned.

Immutability ensures

- Thread safety
- Predictable behaviour.
- No side effects.

Java Enums (29)

whats an Enum?

An Enum (enumeration) is used when we have a fixed list values.

As → • Days of week
• Months of the year
• Departments in a college.

In real applications these values are stored many times. Using strings repeatedly is error-prone (Spelling mistakes, inconsistent values)

Enum solved this by → • Restricting values to a fixed set
• Preventing invalid input
• Making code safer and cleaner.

Problem of using Strings or Constants.

Bad approaches (strings everywhere)

```
S.O.U ("Monday");
S.O.U ("Monday");
S.O.U ("Monday");
```

Better →

Constants class or Interface. →

```
class DayConstants {
    public static final String Monday = "Monday";
    public static final String TUESDAY = "Tuesday";
}
```

Correct and best →

Enum declaration

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

- These are not strings
- These are instance of the enum type
- Monday is a Day, not a String

How enums works

At compile time java converts this enum into:

- A final class
- Each enum constant becomes

```
Public static final Day MONDAY = new Day
                                 ("Monday", 1);
```

- Enums are final → can't be extended
- Enum constructors are private.
- Enum constants are singleton obj.

Using Enum in Code

Assigning enum value.

```
Day day = Day.MONDAY;
```

- This is type-safe
- We can't assign any random value.

```
S.O.U (day);  // MONDAY
```

toString() internally return the enum name.

Built-in Enum Methods

name()

```
day.name() // "MONDAY"
```

returns exact enum constant name as string.

ordinal()

```
day.ordinal(); // 1
```

return index based on declaration order (starts from 0)

values()

```
Day [] days = Day.values();
for (Day d : days) {
        S.O.U (d);
}
```

return array of all enum constants.

valueOf (String)

```
Day d = Day.valueOf ("MONDAY");
```

Match enum strings.

Enum in Switch case

old Switch Syntax

```
Day day = Day.MONDAY;
Switch (day) {
    Case MONDAY:
        S.O.U ("Today - Monday");
        break;
    Case TUESDAY;
        S.O.U ("Today - Tuesday");
        break;
```

```
default:
    S.O.U ("Weekend");
}
```

New Syntax

```
String result = switch (day) {
    Case MONDAY -> "M";
    Case TUESDAY -> "T";
    default -> "weekend";
};

S.O.U (result);
```

Adding Fields, constructors, Methods to Enum.

Enums are real classes, so they can have;

- Fields
- Constructors.
- Methods.

```
Public enum Day {
    MONDAY ("সোমবার");
    TUESDAY ("মঙ্গলবার"),
    WEDNESDAY ("বুধবার");

    Private final String bengliName;

    Private Day (String bengaliName) {
        this.bengaliName = bengaliName;
    }

    Public String getBengaliName() {
        return bengaliName;
    }

S.O.U ( Day.MONDAY.getBengliName()); // সোমবার
```

constructors is always private
Enum instances are created automatically
Constructor runs once per constant

Enum as class Member

Enums can be inside classes.

```
class CalenderApp {
    enum Month {
        JANUARY, FEBRUARY, MARCH
    }
Public static void main (String[] args) {
    Month m = Month.JANUARY;
    S.O.U (m);
}
}
```

Why we need enum

Use enum when

- values are fixed
- values represent states or categories.
- Invalid values must be prevented.