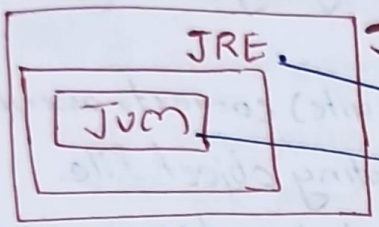
JAVA (collection of Tools)

- JDK → Java development kit
  - → oracle jdk (commercial) → restrences
  - → openjdk (2007) (open source jdk) (original)
- → install binaries → from distributions (freetouse, open source) → Eclipse Temurin provides binaries. ← provides distribution.
- java -version



JDK → Tools and libraries to write java code → Debugger, compiler (javac)

Java Runtime Enviroment : provides runtime enviroment for programs.

Java virtual machine : programs runs on it

- **WorkFlow:**

Test.java (Source code) —compilation (javac (compiler))→ Bytecode (platform Independent) .class —Interpretation (JVM)→ Machine code (platform dependent) 01001001 110100110

→ javac Test.java

→ java Test (runs JVM)

- **Interpreter:** Takes two passes on code → 1st for parsing the code (Parsing) (Grouping) (Tokenization); 2nd Line by Line execution of instruction (Execution)
  - ↑ worst execution.
  - i.e why interpreter are slower than compiler.
- **JIT: Just-In-Time** compilation: JIT compiles bytecode into native machine code at runtime. JVM identifies parts of code that are frequently executed. These parts are compiled into native code by JIT compiler, reducing the overhead of interpreting the bytecode repeatedly.
  - ↑ component of JVM for optimization.

- JVM perform **dynamic translation** i.e bytecode to machine code at runtime. therefore **no standalone Binary** i.e .exe or .bin are produced The machine code exists only in memory during execution of program.
- **JIT compilation:** for frequently executed code, the JVM uses (JIT) compilation to translate bytecode into native machine code on the fly, caching the compiled code for future use.

→ C++ work flow (compilation steps) supports OOP's but C donot

gcc → GNU compiler

① Preprocessing → preprocessor handles directives # → header files, expands macros, remove comments.
O/P → preprocessed source code.

② Compilation → compiler (g++ for cpp) translates preprocessed src code into assembly language.
O/P → Assembly code .asm or .s

③ Assembly → Assembler (part of compiler suite) converts assembly code into machine code creating object file.
• This machine code is specific to target CPU architecture.
O/P → object file (.o or .obj)

④ linking: → linker combines object files and libraries into single executable.
O/P → executable file .exe

⑤ Execution.
→ C++ code is fully compiled into machine code before execution.

- Compiler: A program that translates entire source code of programing language into machine code or intermediate code producing executable file before execution. faster runtime execution.
- Interpreter: A program that translates and executes code simultaneously line by line. slower runtime execution.
- In Java, every code we write has to be part of class → code are collection of classes.

Class → Blueprint

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

Pass To give command line Arguments to main

This is the main method. In Java, the main method is the entry point of a program. It has a specific signature.

- public: Access modifier indicating that the method can be accessed from outside the class. (JVM can access it directly.

Test.main

- **Static**: indicates that the method belongs to the class rather than an instance of the class.
  - without creating an object (method)
  - ↳ Therefore we can access main function using class name itself. → Test.main()
- **void**: specifies that the method does not return any value.
  - main: The name of the method.
- **String[] args**: The method accepts an array of strings as parameters. → Java Test arg1 arg2
  - This is where command-line arguments can be passed to your program.
- System.out.println("Hello world"); ← indicates end of the statement.
  - ↳ This line print the string to the console.
- **System**: A class in the java.lang package that provides access to the system, including the console.
- **out**: An instance of the printstream class within the system class, representing the standard output stream.
- **println**: A method used to print a line of text to the console.

Primitive Data Types:

- Java is statically typed language.
  - ↳ A statically typed language requires variable types to be explicitly declared and known at compile time. C, C++, Java, Rust, Go
- Dynamically Typed language: A dynamically typed language determines the type of a variable at runtime, meaning variable can change types dynamically as program executes. Python, Javascript, Ruby, PHP.
- Integral/datatypes: stores integers in two's complement.
  - ① byte: 1byte(8bits) → -128 to 127 (2^8) → -2^7 to 2^7-1
  - ② short: 2bytes (16bits) → -32,768 to 32,767 → -2^{15} to $2^{15}-1$
  - ③ int: 4bytes (32bits) → -2147,483,648 to 2,147,483,647 → -2^{31} to $2^{31}-1$
    - → by default takes no. as int
  - ④ long: 8bytes (64bits) → -2^{63} to $2^{63}-1$
    - ↳ initializing with no. larger than int in long has to append 'L' to it. To be taken as long.

capacity ↓

- Decimal data types: → stores decimal no. can represent in scientific notation. by default initialization of decimal no. takes as double explicitly mention 'f' for float to assign. ← Here

① float: (less precision upto 7 decimal digits.) 32 bits (4 bytes)

```
float mysal = 1000.1111231121212f;
```

② double: (precision upto 16 decimal digits). 64 bits (8 bytes)

```
double mysal = 1000.1111231121212;
```

Double.MIN_VALUE

- Character data type: stores single character, symbol, integers (number)

① char: 2 bytes (16 bits) ASCII values (16 bit unicode character) → subset of unicode (0 → 127)

- Char var = 'A'; → Type cast → (int) var → Gives ASCII value of corresponding character.
- Char heart = '\u427A4'; ← unicode representation Hexadecimal
- Char heart = 1082; → automatically typecasts.
- 0 to 65535

- Boolean datatype: Stores true or false

① boolean: 8 bits (1 byte) represents logical value.

```
boolean isElg = false;
```

Widening / implicit / automatic conversion: • no loss of data

```
byte byteValue = 10; // 1byte
short shortValue = byteValue; // 2bytes
int intvalue = shortvalue; // 4bytes
long longValue = intvalue; // 8bytes
float floatvalue = longValue; // 4bytes
double doublevalue = floatvalue; // 8 bytes
```

- Bigger containers can store smaller containers values.
- 4bytes float can store long values of 8bytes because it represent them into scientific notation.

- Char charvalue = 'A';

```
int intvaluechar = charvalue; // widening conversion from char to int.
float floatvaluechar = charvalue; // widening conversion from char to float.
```

Narrowing / explicit conversion: • loss of data.

```
double doubleValue = 123.456;
float floatvalue = (float) doubleValue; // narrowing conversion from double to fl
```

long longValue = (long) floatValue; // Narrowing conversion from float to long.

int intValue = (int) longValue; // Narrowing conversion from long to int
↳ Takes 32 bits of right most side only
↳ consider sign bit as 31st MSB bit from right

JAVA essentials:

- Variable naming:
  - variable names are case sensitive.
  - can use letters, number, $, _
  - starting character must begin with a letter, dollar sign ($), or underscore (_)
  - cannot use Java keywords as variable name.
- multiple word variable name seperated by underscore.
  full_name → Snake case
- Multiple word variable name
  - first word's first letter small.
  - Remaining word's first letter capital.
  - Camel case → myFullName.
- make variable name meaningful.

Arithmetic operators: +, -, *, /, % ← modulo (remainder)

- long a = 4383476634;
  long i = a * 24332; → promoted integer to long. To get correct no. stored in container.
  ↳ it has to be long
- Divide by zero → Arithmetic Exception.
- double a = 10;
  int b = 3;
  double quo = a/b; → b will be promoted to double.

operator precedence:

( )
/, *, %
+, -

- if same precedence operators occurs precedence will be given for execution from left to right.
  a/b*c →

compound assignment operator: +=, -=, *=, /=, %=

- a+=1; ≈ a=a+1

byte a = 10;
a = a+5; ← error: integer a will get promoted to integer. But can't store integer (provided) result in Byte (required)
explicit typecasting • fix ⇒ a = (byte)(a+5);

byte a = 10;
a+=5; no errors.
↳ • compound assignment operators perform's implicit type-casting.

- Increment operator: ++a , a++
  increases by 1 — (preincrement) operator — (post increment) operator
- Decrement operator: --a , a--
  (pre decrement) operator — (post decrement) operator
- post operator's → first use the value then increment/decrement.
  int a = 1;
  int b = a++ + a; first use then increment.
  1 + 2 → 3
- pre operator's → first increment/decrement then use it.
  a = 1;
  int b = ++a + a;
  2 + 2 → 4
- Increment/decrement operators has more precedence.
  → ( )
  → ++ , --
  → / , * , %
  → + , -
  - int a = 1;
    int c = a + (++a); ≈ a + (++a)
    1 + 2 → 3
  - a = 1;
    int b = a++ + a; ≈ (a++) + a
    1 + 2 → 3
  - int a = 1;
    int b = a++ + --a; → +, -
    ≈ (a++) + (--a)
    1 + 1 → 2
    a=2 → a=1

Bitwise operators: computer store's values in binaries. operations on bits
↳ faster than arithmetic operators,

Integer.toBinaryString(a); converts integer value to binary string.
↳ Long, Short, Byte → According to datatype.

- operands → byte, short, int, long (Always) Integral data types.
- operators →
  1. and operator &
  2. or operator | (pipe)
  3. xor operator ^ (caret)
  4. not operator ~ (tilde)
  5. left shift << ; a<<1 → left shift's a's bits by 1. Similar to multiplying by 2
  6. Right shift >> ; a>>1 → Similar to dividing by 2
     - puts 0's in MSB for positive numbers.
     - puts 1's in MSB for negative numbers.
  7. Unsigned right shift >>> ; → a>>>2
     Always puts 0's in MSB after right shift.
     - i.e resultant no. always will be positive.

Println vs Print vs Printf :

- println: System.out.println("...");
  - system: → class of system utility methods which interact with systems runtime environment (JRE)
  - out: static member of system class which is connected with console.
  - println: overloaded method. This method adds newline character (cursor gets to the new line.)
  - print: does not add newline character.

(Can pass single Argument.)

- printf: similar to c, does not add newline character. "\n" (platform dependent) or "%n" (platform independent newline character.)

```
int a=1;        %d → format specifier.
String b="string";  %s
char c='A';     %c
Boolean d=true; %b
float e=1.20;   %f → for specifying no. of decimal digit → %.xf
                                                  i.e %.2f → 1.20
System.out.printf("%d,%f,%c,%s,%b",a,e,c,b,d);
```

→ Format specifiers

String (non-primitive data type) (a class which has methods)

→ strings are immutable in java

```
String name="YASH";
```

Java provides functionality to directly give string literal.

```
String a=new String("Ram");
String b=new String("Ram");
String c="Ram";
String d="Ram";
System.out.println(a==b); false
System.out.println(c==d); True
```

(string literal will be replaced with address assigned in string pool.)

(a and b stores addresses to two different object.)

(c and d points to same address in string pool)

- int a=1; → in stack
  - memory is allocated to 'a' and literal is stored in that memory location.
- Student b=new Student(); → in heap
  - a memory is allocated to create object of the class.
  - variable is reference of this memory location. i.e stores address of that location.

- whenever a distinct string literal is used a memory is allocated to it in string pool to store this literal in it. so that if this literal is used again no new memory is allocated to it. it will use it which is stored in pool.

== : not checks equality of string. it checks equality of reference.

string methods:

- str_name.length() : Gives length of the string.
- str_name.charAt(Index): Returns char value at specified index.

- str1-name.equals(str2-name) : compares this string with the specified object.
- str1-name.equalsIgnoreCase(str2-name) : compares the string by ignoring the case.
- str1-name.compareTo(str2-name) : compares string character by character and stops where characters are distinct and return there difference.
  ↳ considers case
- Str1-name.compareToIgnoreCase(str2_name) : ignores case while comparing.
- str_name.substring(begin_Index, end-index) : gives substring of given string.
  (begin_Index: Inclusive; end-index: exclusive.)
- str-name.toUpperCase() : converts whole string to upper case.
- str-name.toLowerCase() : converts whole string to lower case.
- str-name.trim() : returns string, with all leading and trailling space removed.
- name.replace("target-pattern", "Replacement pattern") : returns string where all the occurences of target_pattern is replaced with replacement pattern.
- name.contains("target-pattern") : checks whether given pattern is present in string or not.
- str-name.startsWith("prefix pattern") : checks if string starts with given prefix-pattern.
- str-name.endsWith("suffix-pattern") : checks if string ends with given suffix-pattern.
- str-name.isEmpty() : checks if string is empty or not. (considers whitespaces).
- str-name.isBlank() : ignores white spaces to check string is empty or not.
- str-name.indexOf("pattern") : returns starting index in string where first time give pattern matches in the string.
- str-name.lastIndexOf("pattern") : returns starting index in string where last time given pattern matched.
- String.valueOf(Integer) : Converts given integer into the string. (Integer-variable)
- String.format("name %s and %d years old.", "John", 25) : returns the string which is formated by repacing placeholders with values.

Relational Operators: compares two values and return a boolean result.

- >, <, ==, !=, >=, <=

logical operators: To combine multiple conditions / boolean expression.

- logical AND: &&
- logical OR: ||
- logical not: !

conditional statement:

→ if-else ladder

```
if(boolean_expression){
        //Block code
}
else if(boolean_expression){
        //Block code
}
else{
        //Block code
}
```

- can be used with this datatypes: byte, short, char, int, string, Enum types
- can't be used with double and float → precision problem.

→ switch-case

```
switch(variable){
    case variable_value1 :{
        //Block code.
    }
    case variable_value2 :{
        //Block code
    }
    default :{
        //Block code
    }
}
```

Add break; in each block to stop execution after statisfying case.

Loops:

- While-loop:

```
//intialization
while(boolean_expression){
        //Execute Block code
        //updation
}
```

→ first check's condition then runs the block code.

- Do-while loop:

```
do{
    //Execute block code;
}while(boolean_expression);
```

→ first execute block code then check the condition for next iteration.

- for-loop:

```
for(intialisation; condition; update){
    //Block-code.
}
```

- more readable.
- can reuse intialisation variable after block

ex→

```
for(int var1=x, var2=y; (var1<=a)&&(var2>=b); var1=var1*10, var2++)
{
}
```

can intialise multiple variable → done using comma seperator

can have multiple boolean condition using logical operator

multiple variable updation → done using comma seperator.

Flow control statement:

- Break: break; → stops execution of current block
- continue; → stops execution of further code in current block's iteration. And jumps to next iteration execution.

Arrays // type[] variableName;

→ collection of homogenous data.

→ int a=10; memory is allocated to 'a' in stack to store literal.

→ int[] arr=new int[20]; memory of given amount is allocated in heap and arr is reference variable storing the address of starting array cell in stack.
or
int arr[]=new int[20];

- Access elements using indexing: (arr[i]) arr[2]=256; sout(arr[3]);
- arr.length: returns length of array. here length is property of Array. Therefore no need to append parenthisis after it.
- Initialisation: int[] arr1={1,2,3,4};
  System.out.println(arr1);
  // op: [I@6ce253f1
  [Indicates array; I indicates it's integer type array; 6ce253f1 → Hashcode: unique identifier of the object.
- Range based for-loop for Arrays:
  for(auto ele : Array-name){
  // execute
  }
- Integer.MIN_VALUE → lowest integer value possible

2-D Array: (Matrix) // type[][] variableName;

→ int[][] arr = new int[3][3];
or
int[][] matrix = {{1,2,3},
{4,5,6},
{7,8,9}};

- Access elements using indexing: nums[i][j]

Jagged Array: 2-D array matrix of different length 1-d array.

// int[][]arr = new int[3][ ]; omit this place to create jagged array.

- And assign size of memory of each 1-d array extensively.

// arr[0]=new int[4]; Array of array's of different length.
// arr[1]=new int[2];

Object Oriented Programming (OOP's): Programming paradigm dealing with classes and objects (Blueprint)

- objects has → properties (methods)
  - Behaviour
- public static void main ( ) { ← main method is the entry point of the Java application.
  //code
  }

→ Four principles/pillars of OOPs

① Encapsulation: Bundling of data and methods into a single unit. Class { }
- used to hide data which we don't want to expose. using Private
  ↳ And this private values are accessed using getter's and setter's methods.

② Inheritance:
- code reusability.
- child class inherits properties and fields of parent class.

parent class → public class Animal { ← superclass
void makesound() {"..."} }

public class Cat extends Animal { ← child class of Animal
polymorphism → void makesound() {"meow"} } ← Cat class overrides method of animal class (superclass)

③ Polymorphism:
- Allows objects of different classes to be treated as objects of a common superclass.
- It enables a single interface to represent different underlying forms. (data types)

→ Compile-time polymorphism: (method overloading) The ability to define multiple methods with the same name but different parameters.

→ Run-time polymorphism: (method overidding) The ability of a subclass to provide a specific implementation of a method that is already defined in its superclass. Animal dog = new Dog(); → subclass's object can be treated as instance of it's superclass.
→ polymorphism allows for flexibility and the ability to extend and modify code without disrupting existing functionality.

④ Abstraction: Used to hide underlying implementational details.

Methods

→ accessModifier returnType methodName (parameters) {
//method body.
return ...; ← return variable of defined datatype.
}

← comma separated for multiple parameters in signature

- Non-static method cannot be referenced from a static context (method) → operates at instance level

→ Overloading:

```
private int sum(int a, int b) {
    return a+b;
}
```

same name.

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

- Signature is method name plus parameter's list. doesn't include return type and access modifier.
- Overloading: methods having same method name and different parameter list in signature.
- Non primitive - mutable datatypes when passed to arguments of method they are passed by reference. → methods can modify passed arguments references value.
- Passing variable number of args of same datatype and treating them as array inside method.

```
main(){
    sout(sum(1,2,3));
    sout(sum(new int[]{1,2,3}));
}
```

- one liner array creation → Redundant array creation.

```
sum(int ...a){        // Here we can have multiple integers.
    // variable args
    for(auto ele:a){ }
}
```

Package: (similar to folders) To organise classes into the folder.

- package package_name; name of our current package
- import package_name.class_name; To import particular class from the particular package.
- if need to create object of non-imported package class.
- package_name.class_name variable_x = new package_name.class_name();
- Java.lang package automatically gets imported. in java code.
- we can create package inside a package known as sub-package.
- package package_name.sub-package_name; → must be a first statement of code, represents path to the sub-package shown using '.' operator

Class path → a path where JVM tries to find class from the package.

A parameter - set either on command-line or through an environment variable that tells the JVM and Java compiler where to look for user defined classes and packages.

- Naming convention of package: reverse_domain_name.project_name (v1.1.7)
  - ex → com.purpleblood.corejava
- per java file
- we can access only public classe of the file by importing them across the packages. but other classes which are defined in file and not public are accessible only within a package called Restricted package. class
  - of same name as of file_name → can contain only one public class.

- Non public classes of .java file are not accessible across the packages but are accessible within a package. These classes are restricted package class. that is called package private.

- Note: if we want to use restricted package classes → make them inner class of public class.
- static inner classes can be accessed without instantiating outer.
- for non-static inner class first instantiate outer class than inner class
  - → OuterClass var1 = new OuterClass();
  - → OuterClass.Innerclass var2 = new OuterClass.InnerClass();

#Encapsulation: (Implements boundation)
→ data hiding. we cannot directly set or get values of object.
- we can set or get values of properties of object using methods.
- make all instance variable / fields private. (Accessible within a class.)
- write setter and getter function for this fields.

```
public class Student {
    private int age;
    public void setAge(int age){
        this.age = age;
    }
    public int getAge(){
        return this.age;
    }
}
```
setter →
getter →
this → current instance

- create getter and setter using generate functionality of IDE.

- IDE: (Integrated Development Environment)
→ A software application that provides comprehensive facilities to computer programmers for software development.
Includes:
  - Code Editor: text editor to write source code.
  - Debugger: A tool for testing and debugging code.
  - Compiler/Interpreter: converts your code into machine language.
  - Build Automation Tools: streamlines repetitive tasks in development process.
  - Version control Systems: Helps manage changes to your source code over time.
  - Integrations

#Constructors: (a special method) → same name as of class (performs initial setup)
→ constructors are used to initialise object. → set's default values.

Student student = new Student(); ← Default constructor, Automatically generated by the compiler.

- Can also define customized constructor or parameterised constructor
→ Public Student (String name, int rollNo, int age) { this.age = age;
}

- Note: when customized (or parameterized) constructor in a class is defined. the compiler no longer provides a default constructor automatically.
- had to explicitly define non parameterized constructor.
- Constructor overloading also can be done.

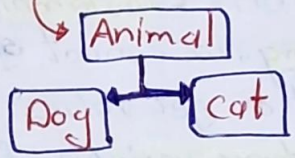
Inheritance:

- Child class inherits properties and methods of parent class. (Code Reusability)

```
Public class Parent {
    public void speaks() {sout("Hi");}
}
Public class Child extends Parent {
    public void speaks() {sout("Hello");}
}
```

↳ method overridding
- overrides method of parent class.

- Single level Inheritance
- multilevel Inheritance → Grandparent → parent → child
- Hierarchical level Inheritance.



- working of constructor call while object creation. Constructor chaining.
→ when a object of most derived class is created the constructors of all the base classes in the hierarchy are called in order, starting from the base class up to the most derived class.

child child = new Child();
↳ while object creation.
→ Grandparent constructor called
→ parent constructor called
→ child constructor called.

- Super → Refers immediate parent.
: This keyword denotes parent class.

flow ↓ by default

- whenever child class constructor is called Java hiddenly add's no argument constructor using super keyword: super(); → first statement of constructor.
→ don't have to explicitly add it to call parent class constructor.
- call to 'super()' must be first statement in constructor body.
↳ done to properly instantiate parent classes constructor first.

default

→ Super is used to invoke superclass / parentclass constructor, methods or fields: → super.age; ← this field should be public

super())
→ super.parentMethod();

→ other than constructor super keyword usage is not mandatory to be a first statement.

→ To call parameterized constructor, we have to explicitly write super keyword in child class constructor. → super(age, name);
calling parent class constructor with arguments.

- new: (keyword) used for dynamic memory allocation. To allocate memory in heap while runtime.
- Annotation: '@' → symbol is used to write annotations for fields and method. Annotation tells information about field, method or class.

→ For overriding annotation → Add following statement above the overriden method. @Override → override annotation. (Good practice to write)

↳ Indicates that method declaration is intended to overide a method declaration in a supertype.

→ method annotated with this annotation type holds atleast one of allowing condition. else generates error

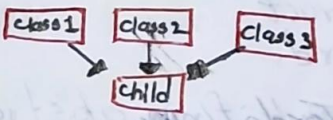
- The method does override or implement a method declared in a supertype.
- The method has a signature that is override-equivalent to that of any public method declared in object.

Multiple Inheritance: Java doesn't support multiple Inheritance.
↳ inherits from multiple parents class.

→ Due to ambiguity of parent class method Java doesn't allow multiple Inheritance.

```
public class Child extends Parent1, Parent2{
};
```

- To solve this problem: it supports Interface based approach using Interfaces



Polymorphism: (many forms)

- polymorphism is a core concept in object-oriented programming (OOP) that allows methods to do different things based on the object it is acting upon, even though the method name and its signature might be same.
  - compile time polymorphism (method overloading)
  - runtime polymorphism. (method overriding)

- Runtime polymorphism (method overriding): Dynamic method Dispatch. (Inheritance)
- JVM decides at runtime which method to run. On basis of object that is created in heap. dynamically decided

→ Animal dog = new Dog(); → subclass's object can be treated as instance dog.sayHello(); upcasting of it's superclass.

- we can have reference of parent class with object of subclass.

→ Animal dog = new Dog(); // upcasting: storing lower hierarchy object in upper hierarchy reference.

→ Dog myDog = (Dog) dog; // downcasting. (if we know the downcasting reference variable is object of current reference.)
↳ reference to the "Dog" object.

- Note: Reference variable: The type of a reference variable determines what methods and properties can be accessed at compile time. i.e can only access methods and fields that are defined in the class directly which instance we are creating. (reference variable)
- object: actual method executed is determined by the object's class. overriden methods in the subclass are executed at runtime.
- casting allows to access methods and properties of the subclass that are not available in the super class.

Abstraction (hiding internal details)

→ Concrete methods: The methods having defination / body.

→ Abstract methods: The methods which don't have defination/ body.

→ Defined as using abstract keyword. `public abstract void methodName();`

Note: for defining abstract method the class has to be defined as Abstract class.
- Future classes extends abstract class.

Abstract class: `public abstract class className{ }`
- can have constructors

↳ abstract class can also contain concrete methods.

- we had to implement methods (or override methods) in child class which are defined as abstract methods in parent class. (which is Abstract class)
↳ which don't have defination in parent class.

OR

- Define child class as a abstract class, which will use as it is abstract methods of parent class inside child class.

To avoid errors.

- Can perform upcasting → can create a reference of abstract superclass with an object of a non-abstract subclass. ex → Animal (Abstract class) myDog = new Dog() (Non-Abstract class);
  → The abstract methods of abstract class is defined in it's non abstract sub-class.
- Cannot create an instance of an abstract class, ex → Animal (Abstract class) dog = new Animal()
  → because abstract classes are meant to be incomplete and meant to be extended by other classes to provide complete implementation of abstract methods.
- public class Animal {
  int age;
  void sayHello() { }
  }
  If we don't mention access modifier to the field's and method's. This methods and field becomes package private.
  → accessible only within the package.
- Public abstract class Animal {
  abstract void sayHello(); package private
  → only subclass extending superclass within package can override this method.
  - subclass across the package extending this superclass must either be declared abstract or override this package private method by making this method protected in superclass.
- private abstract void sayHello(); private to the class.
  → can't access or override anywhere outside the class.
- protected abstract void sayHello(); protected
  → accessible within the package and can also override across the package in class which extends this superclass only
- protected: methods and fields which are protected are only accessible in subclasses which extends the superclass (where they are declared).
  }

Access modifiers: (keywords → Decides visiblity → class, methods, members)
- public: field, method or class is accessible from anywhere.
- Private: Accessible within the class only. class level
  ↳ private constructor: class with a private constructor don't allow to create it's instances

→ public class Utils {
    private Utils() {
    }
    public static void methodName() {
    }
}

← private constructor

→ Utils.methodName();

↑ No need to create instance to Access static members.

• Utils obj = Utils new();

↑ Can't create instance of class having private constructor.

• accessible only within the class, no to outside that's why.

Singleton pattern (Design Interview Imp)

→ if we want to make sure only one instance of the class can be created throughout the application.

• can be done using private constructor, and static keyword.

→ public class School {
    private static School instance; ← creates instance of same class initialised to null.
    private School() { // private constructor: ensures no instance can be created outside the class.
    }
    public static School getInstance() {
        if(instance == null) {
            instance = new School();
        }
        return instance;
    }

class _ _ . {
    → main() { School.getInstance() }
}

← This will be created only once.

* class can have only public and default access modifiers.

• Default (no keyword): if we donot provide any access modifier to
→ the fields, methods, class then accessible only within the package. package level access

• can't create object of default class accross the packages.

- **Protected**: The fields and methods which are protected in class are only accessible in subclass which extends / inherits this class as parent class.
- **Note**: can also access this's protected members without extending this class in subClass only within the package.

# Context \ Access modifier	Private	default (no modifier)	Protected	public
Same class	yes	yes	yes	yes
Same package	No	yes	yes	yes
Subclass (same package)	No	yes	yes	yes
Subclass (different pacakge)	No	No	yes	yes
Different package (non-subclass)	No	No	No	yes

Static (keyword) and also for making utility class.

- The static keyword in Java is used for memory management primarily.
- It can be applied to variables, methods, blocks, and nested classes.
- main concept behind static is that it belongs to the class rather than instances of the class.

Note: Non static fields / members cannot be referenced from a static context.

- can access static method or field using class name
  - ex → Class_name.static_field_Name;
  - Class_name.static-method_name();
- **Note**: can also access using instance of that class too if public.
  - ex → obj_name.staticFieldName;
- The static method cannot use non static data member or call non-static method directly → need to create instance to access.
- this and super cannot be used in static context.
- static methods and variables are associated with the class itself, whereas non-static methods and variables are associated with an instance of the class.

```
public class Test{                    ↓ can call this without creating object of Test
    public static void main (String args[]){
        Test test=new Test();
        test.sum1(1,2);  ← can access this non static method of the
                           class only after instance creation.
        // sum1(1,2); → cannot call it directly → non static field/member
                        cannot be referenced from a static context.

        int sum = sum2(1,2); // can access this static method.
                                from static context
    }
    public int sum1 (int a, int b){
            return a+b;
    }                                  • static data member
                                       private static int age;
    public static int sum2(int a, int b { static method.
            return a+b;
    }
}
```

• since static methods do not have access to instance variables, they can't call nonstatic methods or use non-static data members directly.

Static blocks : used for performing static initialisation.

→ used for one time setup task. → static {

```
ex→ public class Student{
        public static int count;
                              ← static block
        static { System.out.println("initialising static data members");
                 count=0;
        }
    }
```

• Static is also used to design singleton pattern design. and implement utility class.

Imp

Singleton pattern: ensures that a class has only one instance and provides a global point of access to it.

• clever way to manage resources by controlling the instantiation process.

• useful when exactly one object is needed to co-ordinate actions across the system.

→ Singleton pattern designs

```
public class School{
    public static School school;  → static data member

    static{
        sout("static member initialized");  → static block
        school = new School();
    }
    private School(){  → private constructor

    }
    public static School getSchool(){  → static method.
        return school;
    }
}
```

final (keyword) used to define constants.

- once a variable is declared as final, its value cannot be changed.
  public final double PI = 3.14159;
- only can be initialized once.
- can initialize them while declaring or can be intialized inside a constructor if not hard coded while defining.

```
public class Name(){
    PI = 3.14159;
}
```

- static block also can be used to initialize static final keyword variables.

```
public static final double PI;
static{
    PI = 3.4159
}
```

- can only generate getter of the final keyword variables.
- final keyword is used on methods to avoid overriding of this method in inherited class.

```
public final void method(Name(){
    // final keyword avoids overriding of this
} method.
```

- final keyword is used on class to ensure no other class can inherits this class declared as final.

```
public final class name{
    // can't inherits this class.
}
```

- Final → avoids overriding of methods
  → prevents inheritance of class
- Final keyword is not allowed on using on constructor

Interfaces (solution to the multiple inheritance problem of Java.)

- Class → Blueprint for object
- Interface → Blueprint for class
  - Interface class only have → abstract methods
  - Interface class only have → static constants, and static methods
- Interface usecase →
  - To achieve multiple Inheritance
  - To achieve pure abstraction.
- Abstract class can have also → abstract method, concrete methods
  - both declaration and definations is present
- modifier 'abstract' and 'public' is redundant for interface methods
  - ↳ interface method are by default public and abstract inside interface class

∴ public abstract void eat(); ← no need to write modifier explicity.
(public: redundant)
void eat();

- Modifier 'static', 'final' and 'public' are redundant for interface fields;
- Future classes implements interface class ; public static final int age=10;
  (public, static, final: redundant)
  int age=10;

```
public class Dog implements Animal{
// implementation of interface methods
}
```

```
public interface Animal{
// static constants
// declaration of interface methods
// static method public static void info(){
                    sout("info");
                }
}
```
Allowed in Java 8

- Interfaces do not have constructors → can't create object → cannot be instantiated on their own
- The class which implements interface has to implement abstract methods of interfaces or either has to be declared as abstract class.
- can access static constants using interface-name and also by implementing class-name. ex → Interface: Animal.staticConstant; Dog.staticConstant;
- static block is not allowed inside interfaces.
- Interfaces with static constants and static methods can be used as utility class → as it doesn't allow creation of it's object.
- static methods can be only accessed by interface name. And not by future class implementing interface.

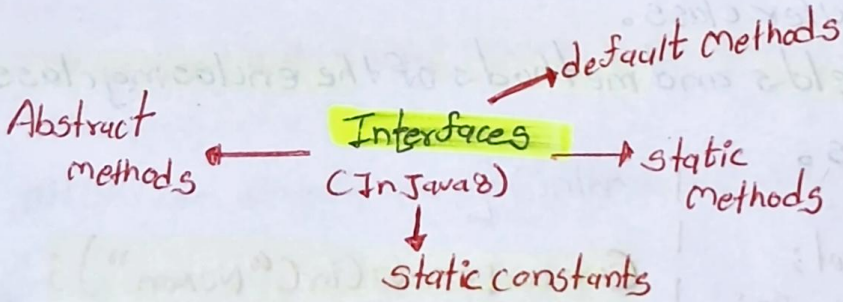
Animal.info(); Dog.info(); Not Allowed

- Static methods defined in an interface are not inherited by implementing classes. They belong solely to the interface and must be accessed using the interface name.
- This static methods in interfaces are meant to provide utility or helper methods related to the interface's domain.
  ex → info()
- In Java 8 interfaces can have static and Default methods also.

Default methods in interfaces → (a concrete method)

- implement class will inherit this default method.
- can be accessed by creating instance of implementing class. only.
- Default methods allows to add new methods to interfaces without breaking the existing implementation classes.
- ensures backward compatibility.
- Override capability: implementing class as inherits this default method they can override them also.

```
public interface {
    public default void run(){
        this.eat();
        sout("....");
    }
```



Multiple Inheritance: (can't achieve pure multiple inheritance.)

- Diamond problem (Ambiguity): if a class implements multiple interfaces with conflicting default methods, it must provide its own implementation of the method to resolve the conflict.
  → ensures single, unambiguous implementation, sidestepping the diamond problem entirely.

```
Class Myclass implements InterfaceA, InterfaceB {
    // implement abstract methods
}  // override default methods.
```

- Interfaces can have main method → As main methods are static. And JVM don't need creation of instance to access main method.

- Difference between Abstract class And Interfaces.
- Abstract classes have instance variables and a constructor to initialise them.
  but interfaces don't have instance variables and constructor.
- one class can only extend one abstract class.
  One class can implement multiple interfaces.

Inner classes

- Types of Inner Classes
  - member Inner Class
  - static Nested class
  - Local Inner class
  - Anonymous Inner class

Member Inner class : Innerclass is a member of outerclass. defined inside body of outer class.

- has access to all the fields and methods of the enclosing class, including private members.

→
```
public class Car{
    private String model;
    private boolean isEngineOn;
    public Car(){
        //constructor.
    }
    Class Engine{
        void start(){
            if(!Engon){
                Engon = True;
                sout(model);
            }
        }
    }
}
```
member inner class

```
main(){
    Car car1=new Car("Nexon");
    Car.Engine eng1=car1.new Engine();
    eng1.start();
}
```

Note: We need to create instance of outer class to create instance of inner class.

- Innerclass is Associated with instance of outerclass.

Static Nested class: Inner class belongs to the outer class rather than the instance.

- doesn't have access to the instance variables and methods of the outer class. Only access to the static variables and static methods of outer class.

```
public class Computer{
    private int ram;
    private static int port;
    public Computer(int ram, string model){
        //constructor.
    }
    static class USB{
        public string type;
        public USB(String type){ this.type = type;}
        public void getInfo(){
            sout(type + " " + port);
        }
    }
}
main(){
    Computer.USB usb1 = new Computer.USB("type-c");
    usb1.getInfo();
}
```

(static Inner class)

NOTE: can directly create instance of static nested class.

Anonymous Inner class:

- when we have to implement interface without creating its seperate implementation class. Or if we want to extend class without creating seperate subclass class → use Anonymous class.
- use when we have to create object for one time use.
- Use when need to override methods of a class or interface.
- Allows to create one-off class with a specific implementation on the fly. without needing to name the class.
- on the fly object creation of impletation class without creating implementation class.

Interface → Payment.java

```
public interface Payment{
    void pay(double amount);
}
```

Class → ShoppingCart.java

```
public class ShoppingCart{
    private double totalAmount;
    public ShoppingCart(double totalAmt){
        this.totalAmt = totalAmt;
    }
    public void processPayment(Payment pm){
        pm.pay(totalAmt);
    }
}
```

This method
- requires instance of interface class. to access pay function
- But can't create instance of interface class.

main: Test.java

```
public class Test{
    public static void main(){
        ShoppingCart shoppingCart = new ShoppingCart(100);
        shoppingCart.processPayment(new Payment(){
            @override
            public void pay(double amt){
                sout("Payint" + amt + "using UPI");
            }
        });
    }
}
```

creating instance of Anonymus class without implementation class of interface reference.

Anonymus class

Local Inner Class: To encapsulate the logic.

- implementation of class inside method for small work purpose.

ex → Hotel.java

```
Public class Hotel{
    private String name;
    private int totalRooms;
    Private int reservedRooms;
    Public Hotel(...){
        // constructor.
    }
    public void reserveRoom(String guestName, int numofRooms){
        Class ReservationValidator{
            boolean validate(){
                if(guestName == null || guestName.isBlank()){
                    sout("Invalid");
                    return false;
                }
                if(reservedRooms + numOfRooms > totalRooms){
                    return false;
                }
                return false;
            }
        }
```

local class

```
ReservationValidator validator = new ReservationValidator();
if(validator.validate()){        // creating instance of local class inside method.
    reservedRooms+=numofRooms;
    sout("guestName+" has reserved "+numofRooms+" rooms");
}
else{sout("Reservation not possible");
}
}
}
```

Exception handling:

#Types of error:
1. Syntax error } code won't be able to compiled — programers fault.
2. logical error } compiled → but logical error
3. Runtime error. → compiled but error at runtime.

→ if code runs except for some cases → perform Handling exceptions.

- The exception Handling is a way to handle the runtime errors so that the normal flow of the application can be maintained.
- Exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.
- In Java each and every class extends object class by default.
- sout() function internally call's to-string method toString() of class whenever argument is passed.

```
try{
    return a/b;
}
catch(ArithmeticException e){
    sout(e); return -1;
}
```

e → object → internally call's to string method of Arithmetic Exception class

- getclass().getName(); → Gives class name

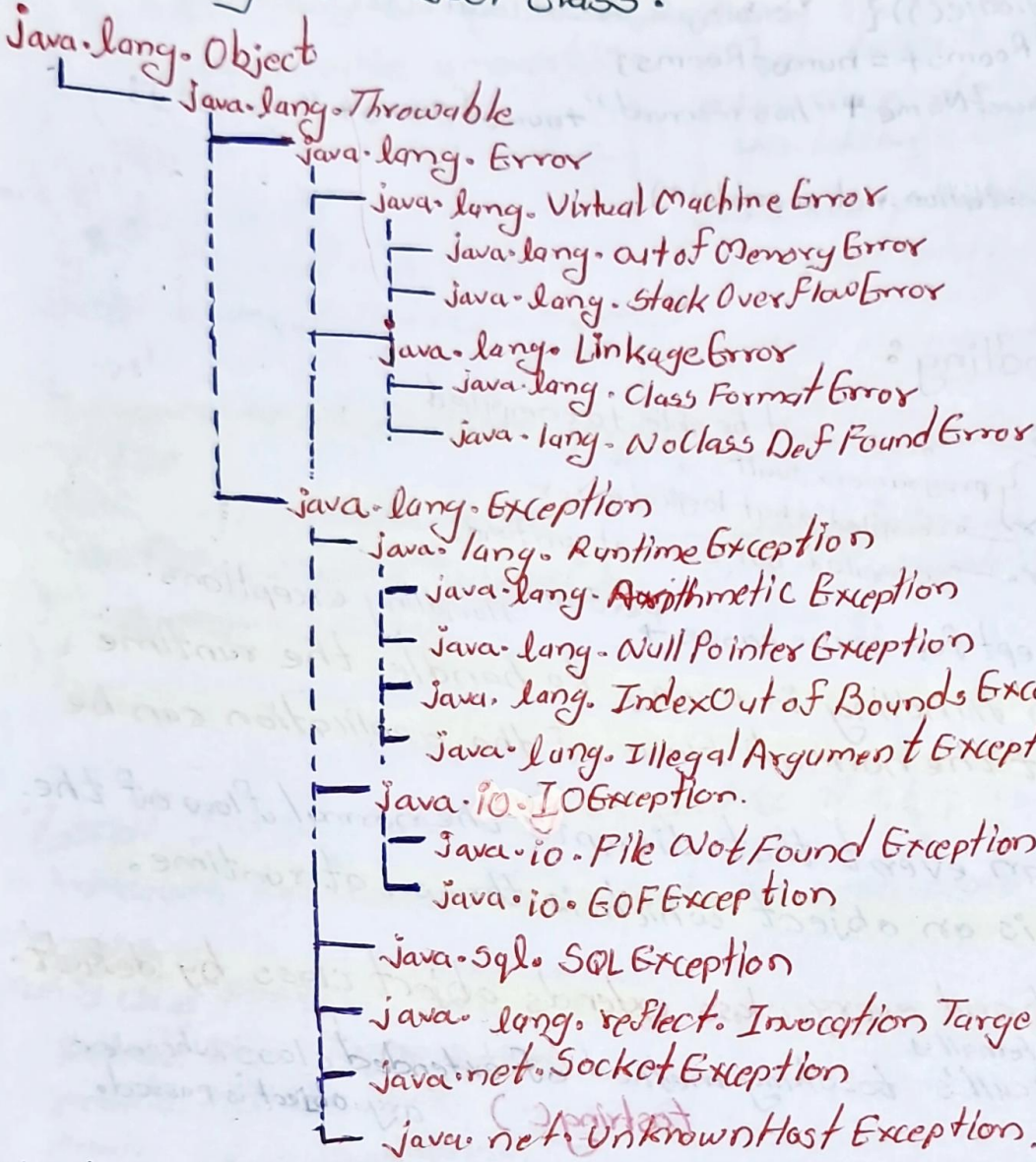
- use Hierarchy of object class to write multiple catch block depending on the type of object we are getting after exception thrown.
- depending on parent and child class Hierarchy.)
- can handle both object at same time using or

Types in multicatch must be disjoint.

→ `catch(ArithmeticException | nullpointerException e){ }`

multicatch

Hierarchy of Object class:



Stack trace → provides detailed information about method calls that led to exception.

↳ Gives snapshot of call stack

- Also can access by → exceptionObject.getStackTrace() = StackTraceElement[] name = ? ← gives object of datatype

- OR (do this)

error-Obj.printStackTrace(); → prints stack trace,

Uncheoked exceptions: This exceptions are not being Checked during compile time. but chaked will run time.

- ex→ nullpointer excep, indexoutofbound, Arithmetic exp

Checked exceptions: The exceptions which are checked during compile time.

ex→ File handling → need to enclose them inside try catch block.

- throws keyword: informs the caller that the method might throw one, or more exceptions. that it may need to handle these exceptions.

```
public static void main() throws Exception {        ← checked exception.
    FileReader fileReader = new FileReader("a.txt");
}
```

- throw keyword: use to throw exception forcefully.

→ throw new FileNotFoundException();

use throws to inform the caller for these forcefull exception or enclosed them inside try-catch block

- Finally block: irrespective of execution of try block or catch block. The finally block will always get executed.

```
finally {
}
```

- useful in return scenario. when you still want to do something after returning.

⇒
```
try { return a/b;
}
catch (Exception e) {
    return -1;
}
finally { sout("Bye");
}
```

- used in file handling. to manage resource and connections.
- To release resources.

try with resources: (in Java 7) helps to manage resources such as streams, files, or socket connections more efficiently and helps prevent resource leaks.

- automatically closes or releases resources.

```
try(BufferedReader br = new BufferedReader(new FileReader("a.txt"))){
    String Line;
    while((Line = br.readLine()) != null){
        sout(line);
    }
}
catch(Exception e){
    sout(e);
}
```

- uses Auto closable interface in class
- No need of adding finally to close connection or release resources.

Custom Exceptions : To handle specific error condition more effectively. easy to debug the code. → customized error name/message.

- Always extend exception class in custom error class

→ CustomException.java ← custom error class.

```
public class CustomException extends Exception {
    public CustomException(String message){
        super(message);      // constructor
    }
}
```

→ Test.java

```
public class Test {
    public static void main(String[] args){
        try{
            validateInput(-1);
        }catch(CustomException e){
            sout("exception" + e.getmessage());
        }
    }
    static void validateInput(int value) throws CustomException{
        if(value < 0){
            throw new CustomException("Negative value");
        }
        sout("value is valid" + value);
    }
}
```

- using throw to forcefully throw custom exception.

- Java is not a pure object oriented programming language.

Wrapper classes: wraps primitive data types value in an object.
- can hold null values.
- can access methods of this object.

→ `int a = 1;` // primitive variable type → stored in stack

→ `Integer b = 1;` // object → variable stores reference → reference variable → value stored in heap.

- wrapper class for int → wraps a value of the primitive type in an object

→ `boolean flag = true;` // primitive variable

→ `Boolean flag = true;` // reference variable → object → heap

- wrapper class for boolean → Boolean class wraps a value of the primitive type boolean in an object.

→ Boolean, Float, Double, Character, Byte, Short, Long

Boxing:

`Integer b = Integer.valueOf(1);` (boxing; returns an object) ≈ `Integer b = 1;`

- Java performs autoboxing no need to do it explicitly.

Unboxing:

`int c = b.intValue();` (unboxing; gives primitive datatype value) ≈ `int c = b;`

- Java performs autounboxing
- This wrapper classes also can hold null values.
- can't use primitive data type in List. because this classes are implemented as using objects.

- Integer.max(1,2);
- Integer.toBinaryString(10)
- Integer.min(1,2);
- Integer.valueOf(str);

Returns binary format of integer value

Return integer value given number in string

- For checking equality of value of these objects:

→
```
Integer a = 1;
Integer b = 1;
```

- sout(a == b); compares value → // True
- sout(b.equals(a)); i.e Actual content. also compares value → // True.

Math class in Java:

- Math.max(a,b) → returns maximum value
- Math.min(a,b) → returns minimum value.
- Math.abs(c) → returns absolute value.
- Math.ceil(d) → Return cell value.
- Math.floor(d) → Returns floor value.
- Math.round(d) → returns closest integer value.
- Math.sqrt(e) → returns squareroot of the given value.
- Math.pow(x,n) → returns value of x^n
- Math.log(x) → calculate log value of x with base e.
- Math.log10(x) → calculate log value of x with base 10.
- Math.PI → Constants → gives value of pi
- Math.sin(x) → Also can use trigonometric functions.
- Math.E → Gives value of e
- Math.random() → Generates a random number between 0.0 to 1 excluding 1.

Enums: Enumeration → listing things

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

this all are instance of DAY class.

$$\Downarrow$$

At compile time this code changes to following.

```
Public final class Day extends java.lang.Enum<Day> {
    Public static final Day SUNDAY = new Day("SUNDAY", 0);
    Public static final Day SATURDAY = new Day("SATURDAY", 6);
    Private static final Day[] VALUES = {SUNDAY, MONDAY, TUESDAY,
    WEDNESDAY, THURSDAY, FRIDAY, SATURDAY};
    Private Day(String name, int ordinal) {
        super(name, ordinal);
    }
}
```

private constructor

so that no new instance should get created outside of enum class.

ordinal → position.

```
public static Day[] values(){
    return values.clone();
}

public static Day valueOf(String name){
    for(Day day : values){
        if(day.name().equals(name)){
            return day;
        }
    }
    throw new IllegalArgumentException("No enum constant" +
                                        name);
}
```

⇒ `Day enumDay = Day.valueOf("MONDAY");`

↳ Returns object of enum → day with string name if present in given list.

`sout(enumDay);`

↳ toString function get's called on this object

- As enum changes to class at compile time;
  - Can also have methods in it.
  - Also can have fields → First things defined in enum body are enum constants.
    - ↳ Then we can have our fields

→ Also can provide custom private constructor to set defined field values.

⇒
```
public enum Day{
    SUNDAY("sunday"), MONDAY("monday");
    private String lower; → fields

    private Day(String lower){ → given custom constructor
        this.lower = lower;      ↳ executed after calling parent class
    }                              constructor
    public void display(){
        sout( this.name + " " + this.lower);
    }                    ↳ user defined methods.
}
```

- enum class are by default static.

Syntax of switch case → Java 12

- No need of using break statements.

```
String res = switch (day) {
    case MONDAY → "M";
    case TUESDAY → "T";
    default → "weekend";
};
```

means returns this value.

Also can do this.

```
case MONDAY → { sout("M"); }
```

Java String vs StringBuilder vs StringBuffer

Immutable — mutables performance — Thread safety

This classes are used to store sequence of characters.

Java String : → Immutable.

after creation, literals are stored in string pool.

```
String str1 = "Hello";
String str2 = str1.concat("World");
```

- concat method returns new string doesn't make changes in original string.

- **StringBuilder:**

```
StringBuilder sb = new StringBuilder("Hello");
```

creates StringBuilder object.

```
String str = sb.toString();
```

Returns Java string object.

```
sb.append("---");   // append text
sb.replace(1, 3, "world");
```

3 exclusive

```
sb.insert(1, "Java");
sb.delete(1, 4);
sb.reverse();
sb.charAt(i);   // returns character at given index
sb.length();
sb.substring(1, 4);
```

→
- mutable
- method chaining.
- Not thread safety.

StringBuffer: used synchronized keyword in all operation for thread safety.

- All other working is same as StringBuilder.
- mutable
- methodchaining
- Thread safety

Feature	String	StringBuilder	StringBuffer
mutability	Immutable	mutable	mutable
Thread safety	Yes	NO	Yes
Performance	slow(due to Immutability)	Fast(no synchronizont on)	slower(due to synchronization)
Storage	String pool (for literals)	Heap	Heap
Usecase	Small, fixed texts	single-threaded apps	multi-threaded apps.

MULTITHREADING

- **Core** → an individual processing unit within a CPU.
- **program:** A set of instructions written in programming language tells computer how to perform a specific task.
- **process:** An instance of a program that is being executed. when program runs, OS creates a process to manage its execution.
- **Thread:** smallest unit of execution within a process. A process can have multiple threads. which share the same resources but can run independently.
- **multitasking:** allows OS to run multiple processes simultaneously Done through time sharing, rapidly switching between tasks on single core.
  on-multicore CPUs, true parallel execution occurs. with task distributed across cores. Assigns different tasks to different cores.
- **Multithreading:** ability to execute multiple threads within a single process concurrently.