Core Java

Java is platform independent, JDK,JRE,JVM

Static Keyword:

The static keyword in Java is a **non-access modifier** used to define class-level members. It allows variables, methods, blocks, or nested classes to belong to the **class itself** rather than any specific instance of the class. This means that static members are shared across all instances of the class and can be accessed without creating an object.

Usage of Methods:

No methods are allowed to write inside the main method can be written inside the class.

Then how to access the data from the outside of the main block 🡪 We can create an objects

Example:

public class CodeJava {

// static int a = 4;

public void getData(){

System.***out***.println("I am in method");

}

public static void main(String[] args) {

// to access the method data we are creating the object here

// object should be same as class name

CodeJava obj = new CodeJava();

// System.out.println(a);

System.***out***.println("hi");

System.***out***.println("hello world");

}

If-else conditions:

for(initalization:condition;increment)

while loop:

while(booleanCondition)

===============================================================

do while loop:

Ex: // do while loop

int j = 10;

do {

System.***out***.println(j);

j++;

}

while(j>20);

}

Nested for loop:

public class NestedLoops {

public static void main(String[] args) {

for(int i=0;i<=2;i++) {

System.***out***.println("outer loop started");{

for(int j=0;j<=3;j++) {

System.***out***.println("inner loop");

}

System.***out***.println("outer loop finished");

}

}

}

Arrays:

Arrays is a collection of similar data type.

For example, if we want to store the names of 100 people then we can create an array of the string type that can store 100 names.

String array[] = new String[100];

Here, the above array cannot store more than 100 names. The number of values in a Java array is always fixed.

Dimensional Array:

Two-dimensional arrays, also known as matrices, are widely used in Java for solving problems involving grids, tables, or matrix-based computations. Below are some common problems and their solutions

String:

In Java, a **String** is an object that represents a sequence of characters. The java.lang.String class is used to create and manipulate strings. Strings in Java are immutable, meaning once a string object is created, its value cannot be changed. If any modification is made, a new string object is created.

1. String is one of the pre-built class in java
2. Alternatively called **literal(s)**, a **literal string** is several [characters](https://www.computerhope.com/jargon/c/charact.htm) enclosed in [double](https://www.computerhope.com/jargon/q/quote.htm) or [single quotes](https://www.computerhope.com/jargon/s/singquot.htm), depending on the [programming language](https://www.computerhope.com/jargon/p/programming-language.htm) or command line. A program does not [interpret](https://www.computerhope.com/jargon/i/interpreted.htm) characters in a literal string until it encounters the next double or single quote.
3. String a ="hello";// string literal
4. String str = new String("helloworld"); // string with class

**String Literals**

In Java, we can use String Literals to initialize a string. A sequence of characters enclosed in double quotes is known as a String Literal.

How to pull out the specific char from given string

a.charAt();

a.indexOf();

a.subString(beginIndex,endIndex);

String class & their method:

String input ="javatraining";

System.***out***.println(input.charAt(3));

System.***out***.println(input.indent(3));

System.***out***.println(input.indexOf("v"));

System.***out***.println(input.substring(4, 10));

System.***out***.println(input.substring(2));

System.***out***.println(input.concat("coding"));

System.***out***.println(input.trim());

input.toUpperCase();

input.toLowerCase();

String arr[] = input.split("t");

System.***out***.println(arr[0]);

System.***out***.println(arr[1]);

System.***out***.println(input.replace("t", "s"));

**Why strings are immutable in java:**

In Java, **strings are immutable**, meaning their values cannot be changed once they are created. This immutability is a fundamental characteristic of the String class in Java and has several important implications for performance, security, and thread safety.

String Buffer, String Builder:

The **StringBuffer** and **StringBuilder** classes are used when there is a necessity to make a lot of modifications to Strings of characters.

Unlike Strings, objects of type StringBuffer and String builder can be modified over and over again without leaving behind a lot of new unused objects.

It is recommended to use **StringBuilder** whenever possible because it is faster than StringBuffer. However, if the thread safety is necessary, the best option is StringBuffer objects.

|  |  |
| --- | --- |
| **Sr.No.** | **Methods & Description** |
| 1 | [**public StringBuffer append(String s)**](https://www.tutorialspoint.com/java/stringbuffer_append.htm)  Updates the value of the object that invoked the method. The method takes boolean, char, int, long, Strings, etc. |
| 2 | [**public StringBuffer reverse()**](https://www.tutorialspoint.com/java/stringbuffer_reverse.htm)  The method reverses the value of the StringBuffer object that invoked the method. |
| 3 | [**public delete(int start, int end)**](https://www.tutorialspoint.com/java/stringbuffer_delete.htm)  Deletes the string starting from the start index until the end index. |
| 4 | [**public insert(int offset, int i)**](https://www.tutorialspoint.com/java/stringbuffer_insert.htm)  This method inserts a string **s** at the position mentioned by the offset. |
| 5 | [**replace(int start, int end, String str)**](https://www.tutorialspoint.com/java/stringbuffer_replace.htm)  This method replaces the characters in a substring of this StringBuffer with characters in the specified String. |

**StringBuilder** is mutable, allowing direct modifications to the same object, making it faster but not thread-safe, suitable for single-threaded environments.

**StringBuffer** is also mutable and thread-safe due to [synchronization](https://www.geeksforgeeks.org/synchronization-in-java/), but slightly slower than StringBuilder, ideal for multi-threaded scenarios requiring safe string operations

Interface:

In Java, an **interface** is a reference type, similar to a class, that can contain only constants, method signatures, default methods, static methods, and nested types. Interfaces are used to achieve **abstraction** and **multiple inheritance** in Java.

**Key Features of Interfaces:**

* An interface defines a **contract** that classes must follow.
* A class that implements an interface must provide implementations for all its methods.
* Interfaces cannot have constructors.
* From Java 8 onwards, interfaces can have **default** and **static** methods.
* From Java 9 onwards, interfaces can also have **private** methods.

**🔹 Syntax Example:**

// Define an interface

interface Animal {

    void eat();

    void sleep();

}

// Implement the interface

class Dog implements Animal {

    public void eat() {

        System.out.println("Dog is eating");

    }

    public void sleep() {

        System.out.println("Dog is sleeping");

    }

}

**Abstraction:**

**Abstraction** in Java is a fundamental **object-oriented programming (OOP)** concept that allows you to hide complex implementation details and show only the essential features of an object. It helps in reducing programming complexity and effort.

**Types of Abstraction in Java:**

1. **Using Abstract Classes**
2. **Using Interfaces**
3. **1. Abstraction with Abstract Classes**
4. An **abstract class** is a class that cannot be instantiated and may contain abstract methods (methods without a body).

abstract class Animal {

    abstract void makeSound(); // abstract method

    void breathe() {

        System.out.println("Breathing...");

    }

}

class Dog extends Animal {

    void makeSound() {

        System.out.println("Bark");

    }

}

* Animal is abstract and defines a template.
* Dog provides the actual implementation of makeSound()
* **2. Abstraction with Interfaces**
* An **interface** is a contract that a class agrees to follow. It can contain abstract methods, default methods, and static methods.

interface Vehicle {

    void start();

    void stop();

}

class Car implements Vehicle {

    public void start() {

        System.out.println("Car started");

    }

    public void stop() {

        System.out.println("Car stopped");

    }

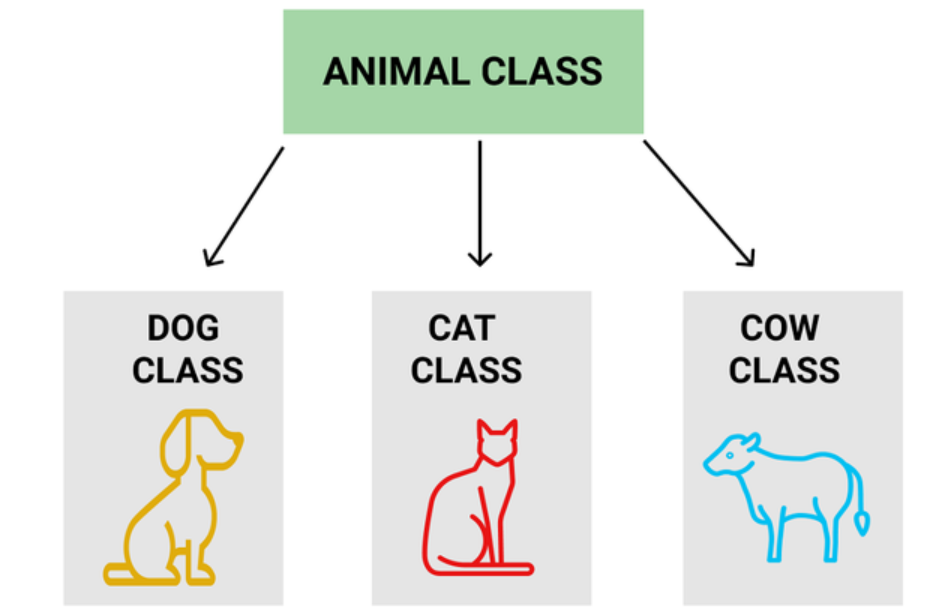
}

* Vehicle defines the behavior.
* Car implements the behavior.

Inheritance:

Java Inheritance is a fundamental concept in [OOP(Object-Oriented Programming)](https://www.geeksforgeeks.org/java/object-oriented-programming-oops-concept-in-java/). It is the mechanism in Java by which one class is allowed to inherit the features(fields and methods) of another class. In Java, Inheritance means creating new classes based on existing ones. A class that inherits from another class can reuse the methods and fields of that class.

**Example:**Animal is the base class and Dog, Cat and Cow are derived classes that extend the Animal class.



Polymorphism:

Polymorphism is a fundamental concept in object-oriented programming (OOP) that allows objects of different types to be accessed through the same interface. This means that a single function or method can operate on different types of objects, and each type can provide its own implementation of the function or method.

**1. Compile-Time Polymorphism (Method Overloading)**

**✅ Definition:**

Occurs when multiple methods in the same class have the same name but different **parameter lists** (type, number, or order). The method to be called is determined at **compile time**.

**How it works:**

The compiler selects the appropriate method based on the method signature during compilation.

class Calculator {

    int multiply(int a, int b) {

        return a \* b;

    }

    double multiply(double a, double b) {

        return a \* b;

    }

    int multiply(int a, int b, int c) {

        return a \* b \* c;

    }

}

====================================================

**2. Runtime Polymorphism (Method Overriding)**

**✅ Definition:**

Occurs when a subclass provides a specific implementation of a method that is already defined in its superclass. The method to be called is determined at **runtime** based on the object type.

**How it works:**

Java uses **dynamic method dispatch** to decide which method to invoke.

class Animal {

    void makeSound() {

        System.out.println("Animal makes sound");

    }

}

class Dog extends Animal {

    void makeSound() {

        System.out.println("Dog barks");

    }

}

class Cat extends Animal {

    void makeSound() {

        System.out.println("Cat meows");

    }

}

public class Test {

    public static void main(String[] args) {

        Animal a;

        a = new Dog();

        a.makeSound();  // Dog barks

        a = new Cat();

        a.makeSound();  // Cat meows

    }

}

Data Classes:

Date d = new Date();

SimpleDateFormat sdf = new SimpleDateFormat();

System.***out***.println(sdf.format(d));

System.***out***.println(d.toString());

In Java, **constructors** are special methods used to initialize objects. They have the same name as the class and do not have a return type—not even void. Here's a breakdown of the key concepts:

**🔹 Types of Constructors in Java**

1. **Default Constructor**
   * Provided by Java if no constructor is defined.
   * Has no parameters.
   * Initializes objects with default values.
2. **Parameterized Constructor**
   * Accepts arguments to initialize fields with specific values.
3. **Copy Constructor**
   * Creates a new object by copying another object's values.
   * Java doesn't provide this by default, but you can define it manually.

**🔹 Key Points**

* Constructors **cannot be abstract, static, final, or synchronized**.
* You can **overload constructors** (i.e., define multiple constructors with different parameter lists).
* If you define any constructor, Java **won’t provide the default constructor** unless you explicitly define it.

**🔹 Constructor Overloading Example**

class Car {

    String model;

    int year;

    Car() {

        model = "Unknown";

        year = 0;

    }

    Car(String m) {

        model = m;

        year = 2025;

    }

    Car(String m, int y) {

        model = m;

        year = y;

    }

}

**Constructor Overloading**

Constructor overloading means having **multiple constructors** in the same class with **different parameter lists**.

class Vehicle {

    String type;

    int wheels;

    // Default constructor

    Vehicle() {

        type = "Unknown";

        wheels = 0;

    }

    // Constructor with one parameter

    Vehicle(String t) {

        type = t;

        wheels = 4;

    }

    // Constructor with two parameters

    Vehicle(String t, int w) {

        type = t;

        wheels = w;

    }

}

Vehicle v1 = new Vehicle();

Vehicle v2 = new Vehicle("Car");

Vehicle v3 = new Vehicle("Bike", 2);

**🔹 Constructor Overriding (⚠️ Misconception Alert)**

**Constructor overriding does not exist in Java** the way method overriding does. Constructors are **not inherited**, so you **cannot override** them in subclasses.

However, you can **call superclass constructors** using super() and define your own constructors in the subclass.

**🔹 Constructors with Inheritance (super() and this())**

**Example:**

**Output:**

Animal constructor called

Dog constructor called

**🔹 Using this() to Call Another Constructor in Same Class**

class Person {

    String name;

    int age;

    Person() {

        this("Unknown", 0); // Calls another constructor

    }

    Person(String n, int a) {

        name = n;

        age = a;

    }

}

Parameterized constructor:

package coreJava;

public class ConstructorDemo {

// will not return values

//name of the constructor should be same as class

// whenever we create an object constructor is called

// block of code when ever object has created

public ConstructorDemo() {

System.***out***.println("I'm in constructor");

}

// parameterized constructor

public ConstructorDemo(int a, int b) {

System.***out***.println("I'm in parameterized constructor");

int c=a+b;

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

}

public ConstructorDemo(String str) {

System.***out***.println("I'm in parameterized constructor1");

System.***out***.println(str);

}

public void getData() {

System.***out***.println("I'm in the method");

}

public static void main(String[] args) {

ConstructorDemo cd = new ConstructorDemo();

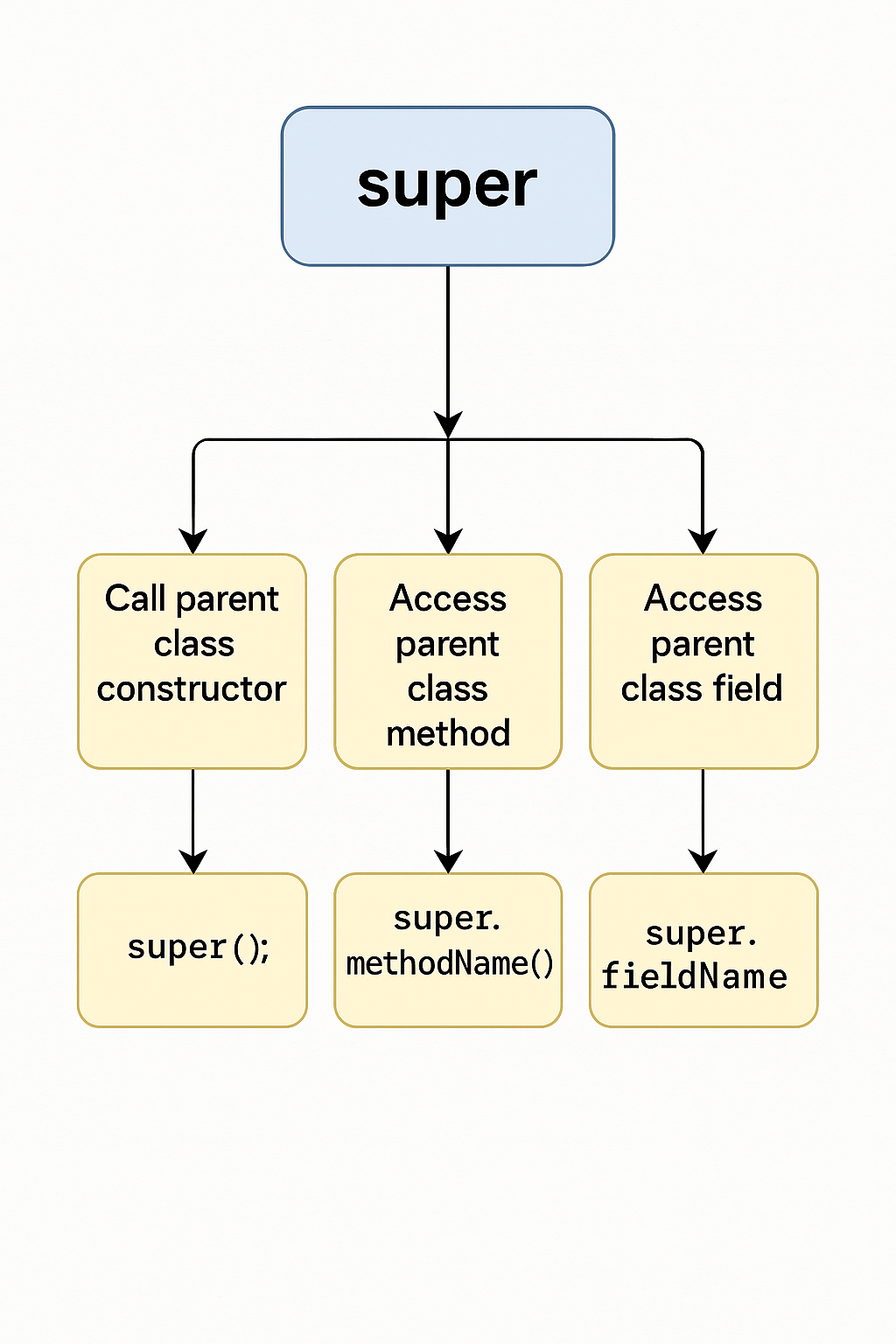
ConstructorDemo cd1 = new ConstructorDemo(4,6);

ConstructorDemo cd2 = new ConstructorDemo("hello");

}

}

Super Keyword :



**🔹 Flowchart Explanation**

1. **Subclass Constructor is Called**
   * Execution starts in the subclass constructor.
2. **super() is Invoked**
   * If super() is present, it calls the **parent class constructor**.
   * This must be the **first statement** in the subclass constructor.
3. **Parent Class Constructor Executes**
   * Initializes parent class fields and runs its logic.
4. **Control Returns to Subclass**
   * After the parent constructor finishes, control returns to the subclass constructor.
5. **Subclass Constructor Continues**
   * Initializes subclass-specific fields and logic.

**this Keyword**

In Java, this is a reference variable that refers to the **current object**. It is used to:

* Differentiate between instance variables and parameters.
* Call other constructors in the same class.
* Pass the current object as a parameter.
* Return the current object for method chaining.

POM example:

public class LoginPage {

    WebDriver driver;

    // Constructor using 'this' to assign driver

    public LoginPage(WebDriver driver) {

        this.driver = driver; // 'this.driver' refers to the instance variable

    }

    By usernameField = By.id("username");

    By passwordField = By.id("password");

    By loginButton = By.id("login");

    public void login(String username, String password) {

        this.driver.findElement(usernameField).sendKeys(username);

        this.driver.findElement(passwordField).sendKeys(password);

        this.driver.findElement(loginButton).click();

    }

}

**Explanation:**

* this.driver ensures that the WebDriver passed to the constructor is assigned to the class-level driver.
* This is crucial in automation frameworks where multiple page classes share the same WebDriver instance.

**🔹 Benefits in Automation Testing**

* **Avoids confusion** between constructor parameters and class fields.
* **Improves readability** and maintainability of test code.
* **Supports method chaining** in fluent interfaces (e.g., this.clickLogin().verifyHomePage()).

**Collections in Java:**

**1. List**

List is an ordered collection (Sometimes called a sequence). In list duplicates are allowed

ArrayList

LinkedList

Vector

2.**Set**

Set collection cannot contain the duplicate elements

However it does not guarantees concerning about order of the iteration.

**3.Map:**

Map is an object that maps to key values. A map cannot contain duplicate keys.

HashMap

TreeMap

LinkedHashMap

Java Iterator:

An **Iterator in Java** is an interface used to traverse elements in a [Collection](https://www.geeksforgeeks.org/java/collections-in-java-2/) sequentially. It provides methods like **hasNext()**, **next()**, and **remove()** to loop through collections and perform manipulation. An Iterator is a part of the **Java Collection Framework**, and we can use it with collections like **ArrayList, LinkedList,** and other classes that implement the Collection interface.

HashMap Vs HashTable:

| **Feature** | **HashMap** | **Hashtable** |
| --- | --- | --- |
| **Thread Safety** | Not synchronized (not thread-safe) | Synchronized (thread-safe) |
| **Performance** | Faster (no overhead of sync) | Slower (due to synchronization) |
| **Null Keys/Values** | Allows one null key and multiple null values | Does **not** allow null key or value |
| **Legacy Status** | Part of Java 1.2 Collections Framework | Legacy class from Java 1.0 |
| **Iterator Type** | Uses **fail-fast** iterator | Uses **enumeration**, not fail-fast |
| **Preferred Use** | Recommended for single-threaded or concurrent use with ConcurrentHashMap | Rarely used in modern code |

// HashMap example

HashMap<integer, string> hm = new HashMap<>();</integer, string>

hm.put(null, "Hello"); // Allowed

hm.put(1, null);       // Allowed

// Hashtable example

Hashtable<integer, string> ht = new Hashtable<>();</integer, string>

ht.put(null, "Hello"); // Throws NullPointerException

ht.put(1, null);       // Throws NullPointerException

Final Keyword:

Access Modifier:

A screen shot of a computer

AI-generated content may be incorrect.

Access Modifier: are essential tools how members of the class like variables, methods and even class can be accessed from other parts of the program.

In Java,**access modifiers** are essential tools that define how the members of a class, like**variables, methods**, and even the**class** itself, can be accessed from other parts of our program. They are an important part of building secure and modular code when designing large applications.

i.Default -- > Access method anywhere only in package  
ii.Private 🡪 variable/method as private we can access these in the subclass  
iii.Protected 🡪 we can not access the variable or method outside of the class same package  
iv.Public 🡪 variable/method as public you can access anywhere in same package

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.