**Static Keyword:**

**Static Variables (Class Variables)**

A **static variable** is shared across all instances of a class. It is **stored in the method area** of memory and does **not** belong to any specific object.

**Example:**

java

CopyEdit

public class Employee {

static String company = "TechCorp"; // Static variable (shared across all instances)

String name; // Instance variable (specific to each object)

Employee(String name) {

this.name = name;

}

void display() {

System.out.println(name + " works at " + company);

}

public static void main(String[] args) {

Employee e1 = new Employee("Alice");

Employee e2 = new Employee("Bob");

e1.display(); // Alice works at TechCorp

e2.display(); // Bob works at TechCorp

Employee.company = "OpenAI"; // Changing static variable

e1.display(); // Alice works at OpenAI

e2.display(); // Bob works at OpenAI

}

}

**Key Points:**

✔ static variables **belong to the class**, not instances.  
✔ Changing static variables affects **all objects**.  
✔ Stored in **method area (not heap memory).**

**2) Static Methods**

A **static method** belongs to the class and **cannot access instance variables or methods directly** because it does not depend on an instance.

A **static method** in Java belongs to the **class** rather than any specific instance (object) of the class. Since static methods are called on the class itself (not an object), they **do not have access to instance variables or instance methods** that belong to a specific object.

Instance variables and methods require an instance of the class to be used because they store data **specific to each object**. A static method, however, does not have any knowledge of which object it is being called on because it operates at the **class level**.

**Example:**

public class MathUtil {

static int square(int x) { // Static method

return x \* x;

}

public static void main(String[] args) {

System.out.println(MathUtil.square(5)); // Output: 25

}

}

**Key Points:**

✔ Can be called **without creating an object**.  
✔ **Cannot** access instance variables/methods directly.  
✔ Mostly used for **utility functions (e.g., Math.pow() in Java).**

**OOPS Part - 3**

**What is an Interface?**

* **A contract** that a class must follow.
* Defines **what** a class should do, but **not how**.
* Used to achieve **full abstraction** (before Java 8).
* Supports **multiple inheritance**, unlike classes.

**Syntax:**

interface Animal {

void makeSound(); // Abstract method (no body)

}

**Implementing an Interface**

A class **must implement all methods** of an interface.

**Example:**

// Defining an interface

interface Animal {

void makeSound(); // Abstract method

}

// Implementing the interface

class Dog implements Animal {

public void makeSound() { // Must override the method

System.out.println("Woof Woof!");

}

}

// Main class

public class Test {

public static void main(String[] args) {

Animal myDog = new Dog();

myDog.makeSound(); // Output: Woof Woof!

}

}

**Key Points:**

✔ **Cannot create objects** of an interface.  
✔ A class **must implement** all abstract methods of the interface.  
✔ Methods in an interface are **public and abstract by default**.

**Multiple Interface Implementation (Multiple Inheritance)**

Unlike classes, a class **can implement multiple interfaces**.

**Example:**

java

CopyEdit

interface Animal {

void eat();

}

interface Pet {

void play();

}

class Dog implements Animal, Pet {

public void eat() {

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

}

public void play() {

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

}

}

public class Test {

public static void main(String[] args) {

Dog d = new Dog();

d.eat(); // Output: Dog is eating

d.play(); // Output: Dog is playing

}

}

**Why is this useful?**

✔ **Java does not support multiple inheritance with classes** but allows it via interfaces.  
✔ A class can **implement multiple behaviors** without conflicts.

**Interface variables:**

=>An interface can contain variables to define requirement level constants.

=> Every interface variable is always public static and final whether we are declaring or

not.

**Example:**

interface interf

{

int x=10;

}

**Public:** To make it available for every implementation class.

**Static:** Without existing object also we have to access this variable.

**Final:** Implementation class can access this value but cannot modify.

**What is the difference between interface, abstract class and concrete class?**

**When we should go for interface, abstract class and concrete class?**

-> If we don’t know anything about implementation just we have requirement

specification then we should go for interface.

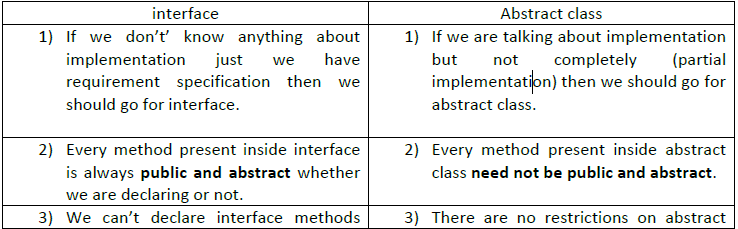
-> If we are talking about implementation but not completely (partial implementation)

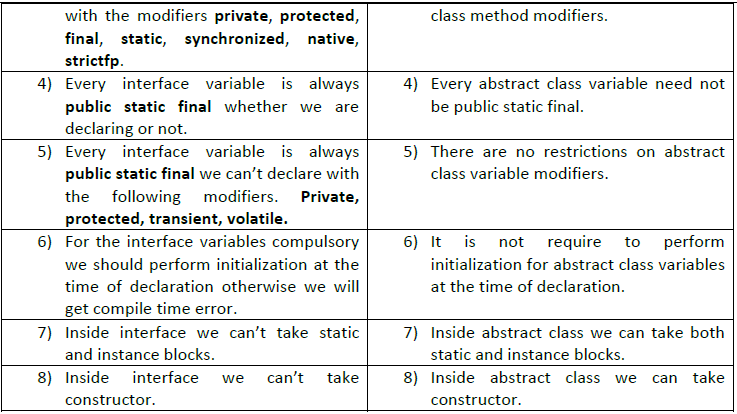
then we should go for abstract class.

-> If we are talking about implementation completely and ready to provide service then we

should go for concrete class.

**What is the Difference between interface and abstract class?**

****

****

**We can’t create object for abstract class but abstract class can contain constructor what is the**

**need?**

=> This constructor will be executed for the initialization of child object.

Example:

class Parent{

Parent()

{

System.out.println(this.hashCode());

}

}

class child extends Parent{

child(){

System.out.println(this.hashCode());

}

}

class Test{

public static void main(String args[]){

child c=new child();

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

}

}

=>Every method present inside interface is abstract but in abstract class also we can take only

abstract methods then what is the need of interface concept?

=> We can replace interface concept with abstract class. But it is not a good programming

practice. We are misusing the roll of abstract class.

### **🔹 When to Use Abstract Classes?**

✔ When you want to **share code** (common behavior) among related classes.  
✔ When classes share **some common functionality**, but each subclass **must implement specific behaviors**.

### **🛠 Real-World Example: Vehicle System**

Imagine a vehicle system where all vehicles have **wheels, engine, and movement**, but their specific behavior (fuel type, max speed) differs.

### **🔹 When to Use Interfaces?**

✔ When you want **full abstraction** (before Java 8, interfaces had only abstract methods).  
✔ When unrelated classes **need common behavior** (e.g., Flyable for both birds and airplanes).  
✔ When you need **multiple inheritance** (Java does not support multiple inheritance with classes).