

# Static keyword level 1



static keyword

now we go level by level growth like a game on this java topic, provide coding example with its explanation, real-world use case example, necessary hints, points to remember, usual mistakes, etc.

start lvl1

Perfect! Let's start **Static Keyword in Java — Level 1**. We will take it step by step like a game, starting with the basics.

## Level 1: Static Variables (Class Variables)

### Definition

A **static variable** belongs to the **class**, not to any specific object. This means all objects of the class share the same copy of that variable.

### Basic Syntax

```
java

class Counter {
    static int count = 0; // static variable

    void increment() {
        count++;
    }

    void display() {
        System.out.println("Count: " + count);
    }
}

public class Main {
    public static void main(String[] args) {
        Counter c1 = new Counter();
        Counter c2 = new Counter();

        c1.increment();
        c1.display(); // Count: 1

        c2.increment();
        c2.display(); // Count: 2

        c1.display(); // Count: 2 (shared)
    }
}
```

## Explanation

1. `count` is **static**, so **shared among all objects**.
2. When `c1` increments it, the value changes for all instances.
3. No matter how many objects you create, there is **only one copy** of `count`.

## Real-World Use Case

- Tracking number of users logged in
- Counting how many objects of a class are created

Example:

```
java

class User {
    static int totalUsers = 0;

    User() {
        totalUsers++;
    }
}

public class Main {
    public static void main(String[] args) {
        new User();
        new User();
        new User();
        System.out.println("Total Users: " + User.totalUsers); // Total Users: 3
    }
}
```

## Hints / Points to Remember

1. **Static variables are memory efficient** because only one copy exists.
2. **Access via class name:** `ClassName.variable`.
3. Can be used **without creating an object**.
4. Can be accessed in **static and non-static methods**.

## Usual Mistakes

1. Trying to access **non-static variables** inside static methods directly.

```
java

class Demo {
    int x = 10;
    static void show() {
        System.out.println(x); // ❌ Error: non-static variable x cannot be referenced from
a static context
    }
}
```

2. Assuming **each object has its own static variable** → wrong, it is shared.

✓ **Level 1 Completed:** You now know about **static variables**.

Next, we can go to **Level 2: Static Methods** which explains how methods can also be static and when to use them.

Do you want me to go to **Level 2**?



next

Perfect! Let's level up.

## Level 2: Static Methods

### Definition

A **static method** belongs to the **class**, not to any specific object. It can be called **without creating an object**.

**Note:** A static method **can only directly access static variables or other static methods** of the class.

### Basic Syntax

```
java

class Calculator {
    static int add(int a, int b) {
        return a + b;
    }

    static int multiply(int a, int b) {
        return a * b;
    }
}

public class Main {
    public static void main(String[] args) {
        // Calling static methods without creating an object
        int sum = Calculator.add(5, 3);
        int product = Calculator.multiply(4, 2);

        System.out.println("Sum: " + sum);           // Sum: 8
        System.out.println("Product: " + product); // Product: 8
    }
}
```

### Explanation

1. `add` and `multiply` are **static methods** → belong to the class.

2. Called directly using `Calculator.add()` without creating an object.
3. Static methods **cannot access non-static variables directly**.

## Real-World Use Cases

1. **Utility methods:** Methods that don't need object data.

Example: `Math.sqrt()`, `Collections.sort()`, `Integer.parseInt()`.

2. **Counting or managing shared resources:**

```
java

class Bank {
    static int totalAccounts = 0;

    static void newAccount() {
        totalAccounts++;
        System.out.println("Account created. Total: " + totalAccounts);
    }
}

public class Main {
    public static void main(String[] args) {
        Bank.newAccount(); // Account created. Total: 1
        Bank.newAccount(); // Account created. Total: 2
    }
}
```

## Hints / Points to Remember

1. Static methods **cannot use this** because there is no object context.
2. Static methods **can call other static methods** directly.
3. Static methods **can access static variables directly**.
4. Can be called **without creating an object** → saves memory.

## Usual Mistakes

1. Trying to access **non-static variables** inside static methods:

```
java

class Demo {
    int x = 10;
    static void show() {
        System.out.println(x); // ❌ Error
    }
}
```

2. Using **this** keyword inside static methods → ❌ not allowed.

✅ **Level 2 Completed:** You now know **static methods**, how they work, and when to use them.

Next is **Level 3: Static Blocks and Initialization**, where we learn how static blocks are used to initialize static variables and execute code when the class loads.

Shall we move to **Level 3**?



next

Awesome! Let's move to **Level 3**.

## Level 3: Static Blocks

### Definition

A **static block** is a block of code inside a class marked with `static { }`.

It **executes only once when the class is loaded**, before any object is created or static methods are called.

It's mainly used to **initialize static variables** or perform one-time setup.

### Basic Syntax

```
java

class Demo {
    static int data;

    // Static block
    static {
        System.out.println("Static block executed");
        data = 100; // initialize static variable
    }

    void show() {
        System.out.println("Data: " + data);
    }
}

public class Main {
    public static void main(String[] args) {
        System.out.println("Main started");
        Demo obj1 = new Demo();
        obj1.show();

        Demo obj2 = new Demo();
        obj2.show();
    }
}
```

### Output:

```
vbnet

Static block executed
Main started
```

```
Data: 100
Data: 100
```

## Explanation

1. `static { }` block runs **once when class is loaded**.
2. It initializes **data** before `main()` or any object creation.
3. Even if you create multiple objects, the **static block runs only once**.

## Real-World Use Case

- Database connection setup
- Configuration loading
- Logging setup

Example:

```
java

class Config {
    static String dbUrl;

    static {
        dbUrl = "jdbc:mysql://localhost:3306/mydb";
        System.out.println("Database URL initialized: " + dbUrl);
    }
}

public class Main {
    public static void main(String[] args) {
        System.out.println("Accessing config: " + Config.dbUrl);
    }
}
```

## Hints / Points to Remember

1. **Executes only once** when the class loads.
2. Multiple static blocks **execute in the order they appear**.
3. Used for **one-time initialization**.
4. Static variables can be initialized in static blocks if computation or exception handling is needed.

## Usual Mistakes

1. Expecting the static block to run **every time an object is created** → ❌ It runs **once** per class loading.
2. Throwing checked exceptions inside a static block without handling → ❌ must handle inside try-catch.

```
java

class Demo {
    static {
        // int a = 5/0; // ❌ Exception will stop class loading
    }
}
```

✅ **Level 3 Completed:** You now know **static blocks** for one-time initialization and their importance.

Next is **Level 4: Static Classes (Nested/Inner Classes)**, where we explore how static works with inner classes.

Do you want to go to **Level 4**?



next

Great! Let's move to **Level 4: Static Nested Classes**.

## Level 4: Static Nested Classes

### Definition

A **static nested class** is a **class defined inside another class** with the **static** keyword.

Unlike regular inner classes, a **static nested class does not need an instance of the outer class** to be created.

### Basic Syntax

```
java

class Outer {
    static int outerData = 10;

    // Static nested class
    static class Nested {
        void display() {
            System.out.println("Outer data: " + outerData);
        }
    }
}

public class Main {
    public static void main(String[] args) {
        // Creating object of static nested class without outer class instance
        Outer.Nested nested = new Outer.Nested();
        nested.display(); // Outer data: 10
    }
}
```

### Explanation

1. **Nested** is a **static nested class** inside **Outer**.
2. Can access **only static members of outer class** directly.
3. Does **not need an outer object** to be created.
4. Acts like a **regular top-level class**, but **namespaced inside outer class**.

## Real-World Use Case

- **Helper classes** that logically belong to outer class but don't need instance data.

Example:

```
java

class Computer {
    static String brand = "Intel";

    static class Processor {
        void showDetails() {
            System.out.println("Processor of brand: " + brand);
        }
    }
}

public class Main {
    public static void main(String[] args) {
        Computer.Processor p = new Computer.Processor();
        p.showDetails(); // Processor of brand: Intel
    }
}
```

- **GUI frameworks** often use static nested classes for **event listeners or adapters**.

## Hints / Points to Remember

1. Can **access static members** of outer class.
2. Cannot directly access **non-static members** of outer class.
3. **Memory efficient**: does not hold reference to outer class object.
4. Used when inner class **does not require outer instance**.

## Usual Mistakes

1. Trying to access **non-static outer variables** directly → ❌
2. Confusing **static nested class** with **inner class** (inner class requires outer object)

```
java

class Outer {
    int data = 10;
    static class Nested {
        void show() {
            System.out.println(data); // ❌ Error: cannot reference non-static variable
        }
    }
}
```



```
}
}
```

3. Trying to create **static local classes inside methods** → ❌ local classes cannot be static.

✅ **Level 4 Completed:** You now know **static nested classes** and their use cases.

Next is **Level 5: Combining Static (Variable, Method, Block, Class) & Best Practices** – a full practical example showing **all static elements together**.

Shall we move to **Level 5**?



next

Awesome! Let's go to **Level 5: Combining All Static Elements & Best Practices**.

## Level 5: All Static Together

This level shows **how static variables, methods, blocks, and nested classes** can work together in a single class.

### Example: Library Management System (Simplified)

```
java

class Library {
    // Static variable: shared by all books
    static int totalBooks;

    // Instance variable: specific to each book
    String title;

    // Static block: initialize totalBooks
    static {
        totalBooks = 0;
        System.out.println("Library system initialized.");
    }

    // Constructor
    Library(String title) {
        this.title = title;
        totalBooks++; // increment total books whenever a new book is created
    }

    // Static method: shows total books
    static void showTotalBooks() {
        System.out.println("Total Books: " + totalBooks);
    }

    // Static nested class: helper to manage library info
    static class LibraryHelper {
        static void welcomeMessage() {
            System.out.println("Welcome to the Library!");
        }
    }

    // Instance method
```

```

    void showBook() {
        System.out.println("Book: " + title);
    }
}

public class Main {
    public static void main(String[] args) {
        // Call static method and nested class without object
        Library.showTotalBooks();           // 0
        Library.LibraryHelper.welcomeMessage();

        // Create books
        Library b1 = new Library("Java Basics");
        Library b2 = new Library("Advanced Java");

        // Access instance and static members
        b1.showBook();                       // Book: Java Basics
        b2.showBook();                       // Book: Advanced Java
        Library.showTotalBooks();           // Total Books: 2
    }
}

```

## Explanation

1. **Static variable** `totalBooks` is shared across all book instances.
2. **Static block** runs once to initialize the library system.
3. **Static method** `showTotalBooks()` can be called without creating an object.
4. **Static nested class** `LibraryHelper` acts as a helper and can be used without outer class object.
5. **Instance variable** `title` is unique for each book.

## Real-World Use Cases

- **Static variables:** Count number of users, objects, or sessions.
- **Static methods:** Utility/helper methods (e.g., `Math`, `Collections`).
- **Static blocks:** Initialize configurations, database connections.
- **Static nested classes:** Organize logically related helper classes (like adapters in GUI frameworks).

## Hints / Best Practices

1. Use static **for shared data** and **utility functions**.
2. Avoid overusing static as it can lead to **tight coupling**.
3. Static members **reduce memory** by not duplicating data per object.
4. Always initialize **complex static variables in static blocks** if needed.
5. Prefer **static nested classes over inner classes** if inner class does not need outer instance.

## Common Mistakes

1. Accessing **non-static members** inside static methods or blocks.

2. Assuming **static variables are separate per object** → they are shared.
3. Overusing static → can reduce **OOP flexibility and testability**.
4. Using **static nested class unnecessarily** when an inner class is better.

✅ **Level 5 Completed:** You now know **everything about static keyword** in Java:

- **Static variable** (shared across objects)
- **Static method** (utility method, no object needed)
- **Static block** (initialization on class load)
- **Static nested class** (class inside class, no outer object needed)

If you want, I can create a **visual diagram and summary table for all static concepts** to make it **super clear in one glance**.

Do you want me to do that?



now start game for final keyword

Perfect! Let's start the **Final Keyword in Java – Level by Level Game**. We'll go step by step like a journey from beginner → advanced.

## Final Keyword in Java – Level 1

### Definition

The **final** keyword in Java is used to **restrict modification**. It can be applied to:

1. **Variables** → Cannot be reassigned.
2. **Methods** → Cannot be overridden by subclasses.
3. **Classes** → Cannot be inherited.

## Level 1: Final Variables (Constants)

### Basic Syntax

```
java

class Circle {
    final double PI = 3.14159; // final variable

    void area(double radius) {
        double area = PI * radius * radius;
        System.out.println("Area: " + area);
    }
}
```

```

    }
}

public class Main {
    public static void main(String[] args) {
        Circle c = new Circle();
        c.area(5);

        // c.PI = 3.14; // ❌ Error: cannot assign a value to final variable PI
    }
}

```

## Explanation

1. **PI** is **final**, so its value **cannot be changed** after initialization.
2. Final variables are **constants** → usually declared in **uppercase**.
3. Must be **initialized** either:
  - **At declaration**
  - **Inside constructor** (for instance-level final variables)

## Real-World Use Case

- Constants like **PI**, **MAX\_USERS**, **DATABASE\_URL**, etc.

```

java

class AppConfig {
    final String DB_URL;

    AppConfig(String url) {
        DB_URL = url; // can initialize final variable in constructor
    }
}

```

## Hints / Points to Remember

1. **Final variables** can be **static** → shared constants.
2. **Final instance variables** must be initialized before object creation.
3. Cannot **reassign final variable**.

## Usual Mistakes

1. Forgetting to **initialize a final variable** → ❌ compilation error.
2. Trying to reassign → ❌
3. Confusing **final variable** with **immutable object** → object reference is final, but object contents can still change (for mutable objects like arrays, lists).

java

```
final int[] arr = {1,2,3};
arr[0] = 10; // ✓ Allowed, reference is final, array content can change
arr = new int[]{4,5}; // ✗ Error
```

✓ **Level 1 Completed:** You now know **final variables**.

Next, **Level 2: Final Methods** – prevent method overriding in subclasses.

Do you want me to go to **Level 2**?



nxt

Great! Let's level up.

## Level 2: Final Methods

### Definition

A **final method** cannot be **overridden** by subclasses.

Useful when you want to **lock the implementation** of a method.

### Basic Syntax

java

```
class Vehicle {
    final void startEngine() {
        System.out.println("Engine started");
    }
}

class Car extends Vehicle {
    // ✗ Cannot override startEngine() because it is final
    /*
    void startEngine() {
        System.out.println("Car engine started");
    }
    */
}

public class Main {
    public static void main(String[] args) {
        Car c = new Car();
        c.startEngine(); // Engine started
    }
}
```

### Explanation

1. **startEngine()** is **final** → cannot be overridden in subclass.

2. Ensures **method behavior is fixed**.
3. Final methods can still be called via objects normally.

## Real-World Use Cases

- Security-sensitive methods (e.g., `Object.clone()` in Java is final)
- Framework methods that should not be overridden by developers

```
java

class BankAccount {
    final void deposit(double amount) {
        System.out.println("Deposited: " + amount);
    }
}
```

## Hints / Points to Remember

1. Final methods **can be inherited** → but cannot be overridden.
2. Can be **static or non-static**.
3. Helps in **preventing accidental behavior changes**.

## Usual Mistakes

1. Trying to override a final method → ❌ compilation error.
2. Misunderstanding: final **does not prevent method usage**; only overriding.

✅ **Level 2 Completed:** You now know **final methods**.

Next, **Level 3: Final Classes** – prevent class inheritance entirely.

Shall we go to **Level 3**?



next

Awesome! Let's move to **Level 3: Final Classes**.

## Level 3: Final Classes

### Definition

A **final class** cannot be **subclassed**.

Used when you want to **lock the entire class** so that no one can extend it.

## Basic Syntax

```
java

final class MathUtils {
    static int square(int x) {
        return x * x;
    }
}

// ❌ Cannot inherit from final class
/*
class AdvancedMath extends MathUtils {
}
*/

public class Main {
    public static void main(String[] args) {
        int result = MathUtils.square(5);
        System.out.println("Square: " + result); // Square: 25
    }
}
```

## Explanation

1. **MathUtils** is **final** → cannot have any subclasses.
2. Methods inside a final class **cannot be overridden** (even if they are not final).
3. Useful for **utility classes** or **security-critical classes**.

## Real-World Use Cases

- **Utility/helper classes:** `java.lang.Math` is a final class
- **Immutable classes:** `String` is final → cannot be extended
- **Security classes:** To prevent modification or overriding in sensitive frameworks

```
java

final class ImmutableEmployee {
    private final String name;

    ImmutableEmployee(String name) {
        this.name = name;
    }

    String getName() {
        return name;
    }
}
```

## Hints / Points to Remember

1. Final class **cannot be subclassed**, but objects can still be created.
2. All methods in a final class are **implicitly non-overridable**.
3. Often combined with **final variables** for immutable objects.

## Usual Mistakes

1. Trying to extend a final class → ❌ compilation error.
2. Confusing `final` class with `abstract` class → abstract is meant to be extended, final cannot.

✅ **Level 3 Completed:** You now know **final classes** and why they are used.

Next, **Level 4: Final Parameters and Local Variables** – using `final` in methods and local variables for safety.

Do you want to go to **Level 4**?



next

Perfect! Let's move to **Level 4: Final Parameters and Local Variables**.

## Level 4: Final Parameters and Local Variables

### Definition

- **Final parameters:** Method parameters declared as `final` **cannot be reassigned** inside the method.
- **Final local variables:** Local variables in a method declared as `final` **cannot change once assigned**.

### Basic Syntax

```
java

class Demo {
    // Final parameter
    void greet(final String name) {
        System.out.println("Hello, " + name);
        // name = "John"; // ❌ Error: cannot assign a value to final parameter
    }

    void showNumber() {
        final int num = 10;
        System.out.println("Number: " + num);
        // num = 20; // ❌ Error: cannot reassign final local variable
    }
}

public class Main {
    public static void main(String[] args) {
        Demo d = new Demo();
        d.greet("Alice");    // Hello, Alice
        d.showNumber();      // Number: 10
    }
}
```



## Explanation

1. **final** prevents **reassignment**.
2. Makes code **safer** → no accidental changes.
3. Can be combined with **objects**: final reference cannot change, but object contents can.

```
java

final StringBuilder sb = new StringBuilder("Hello");
sb.append(" World"); // ✅ Allowed
// sb = new StringBuilder("New"); // ❌ Not allowed
```

## Real-World Use Cases

1. **Immutable method parameters**: Prevent accidental changes inside methods.
2. **Anonymous classes & lambdas**: Must refer to **final or effectively final variables**.

```
java

public class LambdaDemo {
    public static void main(String[] args) {
        final int x = 5;
        Runnable r = () -> System.out.println("Value: " + x);
        r.run(); // Value: 5
    }
}
```

## Hints / Points to Remember

1. Final local variables **must be initialized** before use.
2. For **parameters**, it is optional but recommended for clarity and safety.
3. Final reference → object cannot be reassigned, but object contents **can change if mutable**.

## Usual Mistakes

1. Reassigning a final variable → ❌ compilation error.
2. Confusing **final reference** with **immutable object**.
3. Forgetting that **lambdas and anonymous classes require final or effectively final variables**.

✅ **Level 4 Completed**: You now know **final parameters and local variables** and how they improve safety.

Next is **Level 5: Combining Final Variables, Methods, and Classes – Immutable Object Example**, which will show a **practical use case combining everything**.

Shall we go to **Level 5**?



next

Awesome! Let's move to **Level 5: Combining Final Variables, Methods, and Classes – Immutable Object Example.**

## Level 5: Final in Practice – Immutable Class

### Definition

An **immutable class** is a class whose **object state cannot be changed after creation**.

We use `final` for:

- Class → prevent inheritance
- Variables → prevent reassignment
- Methods → prevent overriding

### Example: Immutable Employee Class

```
java

final class Employee {
    private final String name;    // final variable
    private final int age;       // final variable

    // Constructor to initialize final variables
    public Employee(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // Getter methods (no setters!)
    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }

    // Final method (optional)
    public final void displayInfo() {
        System.out.println("Employee Name: " + name + ", Age: " + age);
    }
}

public class Main {
    public static void main(String[] args) {
        Employee e1 = new Employee("Alice", 25);
        e1.displayInfo(); // Employee Name: Alice, Age: 25

        // ❌ Cannot modify variables
        // e1.name = "Bob";
        // e1.age = 30;
    }
}
```

## Explanation

1. **final class** → cannot be subclassed.
2. **final variables** → cannot be reassigned after constructor initialization.
3. **final methods** → cannot be overridden.
4. No setters → object is **completely immutable**.

## Real-World Use Cases

- **String class** in Java → immutable
- **Security-sensitive classes** → prevent modification
- **Thread-safe objects** → immutable objects are safe to share between threads
- **Configuration objects** → constants and settings

```
java

final class Config {
    private final String DB_URL;
    private final int TIMEOUT;

    Config(String url, int timeout) {
        DB_URL = url;
        TIMEOUT = timeout;
    }

    public String getDBUrl() { return DB_URL; }
    public int getTimeout() { return TIMEOUT; }
}
```

## Hints / Points to Remember

1. Final class → prevent subclassing.
2. Final variables → initialize once in constructor or declaration.
3. Final methods → prevent overriding.
4. Combining final in classes, variables, and methods → helps create **immutable, safe, and robust objects**.

## Common Mistakes

1. Forgetting to initialize final variables → ❌ compilation error
2. Adding setters in immutable class → ❌ breaks immutability
3. Confusing final **reference** with **immutable object**

✅ **Level 5 Completed:** You now know **everything about the final keyword**:

- **Final variables** → constants

- **Final methods** → prevent overriding
- **Final classes** → prevent inheritance
- **Final parameters/local variables** → safe assignments
- **Immutable objects** → practical combination

I can now create a **one-page visual summary table comparing static vs final in Java** so you can **remember all rules quickly**.

Do you want me to do that?



now abstract

Perfect! Let's start the **Abstract Keyword in Java – Level by Level Game**. We'll go step by step like a journey from beginner → advanced.

## Abstract Keyword in Java – Level 1

### Definition

The **abstract** keyword is used to **declare incomplete entities** in Java that **cannot be instantiated directly**.

It can be applied to:

1. **Classes** → abstract class cannot be instantiated.
2. **Methods** → abstract method has **no body**; must be implemented in subclasses.

Abstract classes are used when you want to provide **common base functionality** but leave some methods **to be implemented by child classes**.

## Level 1: Abstract Classes

### Basic Syntax

```
java

abstract class Vehicle {
    String brand;

    Vehicle(String brand) {
        this.brand = brand;
    }

    // Abstract method (no body)
    abstract void start();

    // Non-abstract method (can have implementation)
    void info() {
        System.out.println("Vehicle brand: " + brand);
    }
}
```

```

    }
}

// Subclass must implement abstract method
class Car extends Vehicle {
    Car(String brand) {
        super(brand);
    }

    @Override
    void start() {
        System.out.println(brand + " car is starting");
    }
}

public class Main {
    public static void main(String[] args) {
        // ❌ Vehicle v = new Vehicle(); // Cannot instantiate abstract class
        Car c = new Car("Toyota");
        c.info(); // Vehicle brand: Toyota
        c.start(); // Toyota car is starting
    }
}

```

## Explanation

1. **Vehicle** is **abstract class** → cannot create objects.
2. **start()** is **abstract method** → no implementation, subclass must override.
3. **info()** is **regular method** → can provide shared functionality to all subclasses.
4. Abstract class can have **constructors, fields, and non-abstract methods**.

## Real-World Use Case

- **Vehicle management system** → all vehicles have a brand, but **start()** differs per type.
- **GUI frameworks** → base components with abstract methods to be implemented by concrete components.

## Hints / Points to Remember

1. Abstract class **cannot be instantiated**.
2. Can contain **both abstract and non-abstract methods**.
3. Can have **constructors** to initialize common fields.
4. Abstract methods **must be implemented** by non-abstract subclasses.

## Usual Mistakes

1. Trying to create an object of abstract class → ❌ compilation error
2. Forgetting to implement abstract methods in subclass → ❌ compilation error
3. Declaring abstract method **with body** → ❌ not allowed

✓ **Level 1 Completed:** You now know **abstract classes and methods basics**.

Next, **Level 2: Abstract Methods Only Class vs Mixed Abstract Class** – deeper dive into variations.

Do you want me to go to **Level 2**?



next

Great! Let's move to **Level 2: Abstract Methods Only vs Mixed Abstract Class**.

## Level 2: Types of Abstract Classes

### 1. Abstract Class with Only Abstract Methods

- Contains **only abstract methods** → acts like an **interface with implementation flexibility**.
- Subclasses must implement **all abstract methods**.

```
java

abstract class Shape {
    abstract double area();
    abstract double perimeter();
}

class Circle extends Shape {
    double radius;

    Circle(double radius) {
        this.radius = radius;
    }

    @Override
    double area() {
        return Math.PI * radius * radius;
    }

    @Override
    double perimeter() {
        return 2 * Math.PI * radius;
    }
}

public class Main {
    public static void main(String[] args) {
        Circle c = new Circle(5);
        System.out.println("Area: " + c.area()); // 78.53981633974483
        System.out.println("Perimeter: " + c.perimeter()); // 31.41592653589793
    }
}
```

### 2. Abstract Class with Mixed Methods

- Contains **abstract and non-abstract methods** → provides **default behavior** and **mandatory implementation**.

```
java

abstract class Vehicle {
    abstract void start(); // abstract

    void fuelType() {      // non-abstract
        System.out.println("Fuel: Petrol/Diesel");
    }
}

class Bike extends Vehicle {
    @Override
    void start() {
        System.out.println("Bike started");
    }
}

public class Main {
    public static void main(String[] args) {
        Bike b = new Bike();
        b.fuelType(); // Fuel: Petrol/Diesel
        b.start();    // Bike started
    }
}
```

## Explanation

1. **Only abstract methods class** → forces full implementation.
2. **Mixed class** → allows **common methods** to be shared and **abstract methods** to enforce customization.
3. Subclasses **cannot skip abstract method implementation** unless they are also abstract.

## Real-World Use Cases

1. **Shape/Vehicle hierarchy** → different types of shapes/vehicles share **some methods** and have **custom methods**.
2. **Frameworks** → abstract base class provides common tools (logging, validation) while forcing certain methods (render, process) to be implemented.

## Hints / Points to Remember

1. Abstract class **cannot be instantiated**.
2. Can have **fields, constructors, non-abstract methods**.
3. Subclass **must override all abstract methods** unless subclass is abstract.
4. Abstract class is **heavier than interface**, can hold state.

## Usual Mistakes

1. Forgetting to implement abstract methods → **✗** compilation error.
2. Trying to instantiate abstract class → **✗** compilation error.

3. Declaring abstract method inside non-abstract class without using **abstract** keyword → **✗** error.

✓ **Level 2 Completed:** You now know the **two types of abstract classes** and when to use them.

Next, **Level 3: Abstract Class vs Interface** – understanding the differences and choosing the right one.

Do you want me to go to **Level 3**?



now volatile

Perfect! Let's start the **Volatile Keyword in Java – Level by Level Game**. We'll go step by step to understand its concept, usage, and pitfalls.

## Volatile Keyword in Java – Level 1

### Definition

The `volatile` keyword in Java is used for **variables** to ensure that:

1. **Changes made by one thread are visible to all other threads immediately.**
2. **Prevents threads from caching variables locally.**

Essentially, it provides a **lightweight synchronization mechanism** for variables shared between threads.

## Level 1: Basic Volatile Variable

### Example

```
java

class SharedData {
    volatile boolean flag = true; // volatile variable
}

class ThreadExample extends Thread {
    SharedData data;

    ThreadExample(SharedData data) {
        this.data = data;
    }

    public void run() {
        while (data.flag) { // read the latest value of flag
            // do some work
        }
        System.out.println("Thread stopped as flag is false");
    }
}

public class Main {
    public static void main(String[] args) throws InterruptedException {
        SharedData data = new SharedData();
    }
}
```



```
ThreadExample t = new ThreadExample(data);
t.start();

Thread.sleep(1000);
data.flag = false; // update volatile variable
System.out.println("Flag set to false");
}
}
```

## Explanation

1. `flag` is **volatile** → any change is **immediately visible to other threads**.
2. Without **volatile**, the thread may **cache flag locally**, so it might not see updates in real time.
3. **volatile does not lock** the variable; it only ensures visibility.

## Real-World Use Cases

1. **Stop signals for threads**
  - Example: `while (!shutdown) { ... }`
2. **State flags in concurrent systems**
3. **Lightweight synchronization** when atomicity is not required

## Hints / Points to Remember

1. Volatile variables **cannot guarantee atomicity** for operations like `count++`.
2. Suitable for **boolean flags or single read/write variables**.
3. Avoid using volatile for **complex operations** → use `synchronized` or `AtomicInteger`.
4. Volatile works at the **hardware and JVM memory level** → ensures visibility across CPU cores.

## Usual Mistakes

1. Assuming **volatile provides atomicity** → ❌ it only guarantees visibility.
2. Using **volatile for compound operations** → ❌ `counter++` is not thread-safe.
3. Forgetting volatile → thread may **never see updated value** in loops.

```
java

volatile int count = 0;

count++; // ❌ Not atomic, can cause race conditions
```

✅ **Level 1 Completed:** You now know **volatile variables ensure visibility across threads**.

Next is **Level 2: Volatile with Multi-threaded Example (Stop Thread Safely)**, which will show practical usage.

Do you want me to go to **Level 2**?



using synchronized method also ensure same functionality, how

Exactly! Let's carefully compare `volatile` and `synchronized` in Java, because both deal with **threads and memory visibility**, but they work differently.

## 1. Visibility Guarantee

- **Volatile:** Ensures that **changes made by one thread to a variable are visible immediately to other threads**.
  - Example: A flag that stops a thread:

```
java

class SharedData {
    volatile boolean flag = true;
}
```

- **Synchronized:** Also ensures **visibility** because entering a synchronized block **flushes the thread's working memory** and reads the latest value from main memory.

```
java

class SharedData {
    boolean flag = true;

    synchronized void setFlag(boolean value) {
        flag = value;
    }

    synchronized boolean getFlag() {
        return flag;
    }
}
```

So, yes — `synchronized` also guarantees visibility like `volatile`.

## 2. Atomicity

- **Volatile:** Does **not guarantee atomicity**. Operations like `count++` are not safe.
- **Synchronized:** **Guarantees atomicity** inside the synchronized method/block.

```
java

class Counter {
    int count = 0;

    synchronized void increment() {
        count++; // atomic inside synchronized
    }
}
```

```

    }

    synchronized int getCount() {
        return count;
    }
}

```

## 3. When to Use Which

Feature	Volatile	Synchronized
Visibility	✓ Yes	✓ Yes
Atomicity	✗ No	✓ Yes
Performance	✓ Lightweight	✗ Heavier (locks)
Use Case	Flags, simple state variables	Counters, complex operations, critical sections
Blocking / Locking	✗ None	✓ Acquires lock

## 4. Practical Example: Stopping Thread Safely

### Using Volatile

```

java

class SharedData {
    volatile boolean flag = true;
}

class MyThread extends Thread {
    SharedData data;

    MyThread(SharedData data) { this.data = data; }

    public void run() {
        while (data.flag) {}
        System.out.println("Thread stopped");
    }
}

public class Main {
    public static void main(String[] args) throws InterruptedException {
        SharedData data = new SharedData();
        MyThread t = new MyThread(data);
        t.start();

        Thread.sleep(1000);
        data.flag = false; // stop thread
    }
}

```

### Using Synchronized Method

```

java

class SharedData {
    private boolean flag = true;

    synchronized void setFlag(boolean value) { flag = value; }
    synchronized boolean getFlag() { return flag; }
}

```

```

}

class MyThread extends Thread {
    SharedData data;

    MyThread(SharedData data) { this.data = data; }

    public void run() {
        while (data.getFlag()) {} // reads latest value due to synchronized
        System.out.println("Thread stopped");
    }
}

public class Main {
    public static void main(String[] args) throws InterruptedException {
        SharedData data = new SharedData();
        MyThread t = new MyThread(data);
        t.start();

        Thread.sleep(1000);
        data.setFlag(false); // stop thread
    }
}

```

✓ Both **volatile** and **synchronized** ensure that **the change to the flag is visible to the thread**, so the thread stops.

## Key Takeaways

1. **volatile** → **lightweight, visibility only**, no locks.
2. **synchronized** → **visibility + atomicity**, involves **locking**, heavier.
3. Use **volatile for simple flags**; use **synchronized for critical sections**.

If you want, I can make a **clear visual diagram showing volatile vs synchronized memory visibility and thread behavior**, which makes it **super easy to remember**.

Do you want me to do that?