A **nested class** is a class defined **inside another class**. It helps **logically group** classes and improve **encapsulation**. Java provides **four types** of nested classes:

1 **Static Nested Class**  
2 **Non-Static Inner Class**  
3 **Local Inner Class**  
4 **Anonymous Inner Class**

**Applications of Nested Classes in Java**

**1 Encapsulation & Grouping Related Classes**

* If a class is **closely associated** with another, nesting improves readability.
* Example: **A car has an engine**, so Engine can be a nested class inside Car.

class Car {

private String model;

Car(String model) {

this.model = model;

}

// Static Nested Class

static class Engine {

void start() {

System.out.println("Engine started!");

}

}

}

public class Main {

public static void main(String[] args) {

Car.Engine engine = new Car.Engine();

engine.start();

}

}

**Application**: Grouping Engine inside Car makes sense because the engine is **meaningful only inside a car**.

**2 Better Encapsulation & Code Organization**

* **Private nested classes** **hide implementation details** from outside classes.
* Example: **Security mechanisms inside an authentication system.**

class SecuritySystem {

private static class AuthHelper {

static boolean validateUser(String username, String password) {

return username.equals("admin") && password.equals("1234");

}

}

public boolean authenticate(String user, String pass) {

return AuthHelper.validateUser(user, pass);

}

}

public class Main {

public static void main(String[] args) {

SecuritySystem security = new SecuritySystem();

System.out.println(security.authenticate("admin", "1234")); // true

}

}

**Application**: AuthHelper is completely hidden from outside classes.

**3 Used in Event Handling (Anonymous Inner Classes)**

* **GUI frameworks like Swing & Android use nested classes for event handling.**
* Example: **Button click listener in Java Swing**

import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;

import javax.swing.\*;

public class ButtonExample {

public static void main(String[] args) {

JFrame frame = new JFrame("Button Example");

JButton button = new JButton("Click Me");

// Anonymous Inner Class for event handling

button.addActionListener(new ActionListener() {

public void actionPerformed(ActionEvent e) {

System.out.println("Button Clicked!");

}

});

frame.add(button);

frame.setSize(300, 200);

frame.setLayout(null);

frame.setVisible(true);

}

}

**Application**: Anonymous inner classes are **commonly used in event handling** in Java GUI development.

**4 Used in Iterators & Data Structures**

* **Nested classes help implement iterators for collections.**
* Example: **Creating a custom list iterator**

import java.util.Iterator;

class CustomList implements Iterable<Integer> {

private int[] data = {1, 2, 3, 4, 5};

// Inner class for iterator

private class CustomIterator implements Iterator<Integer> {

private int index = 0;

public boolean hasNext() {

return index < data.length;

}

public Integer next() {

return data[index++];

}

}

public Iterator<Integer> iterator() {

return new CustomIterator();

}

}

public class Main {

public static void main(String[] args) {

CustomList list = new CustomList();

for (int num : list) {

System.out.println(num); // 1 2 3 4 5

}

}

}

**Application**: The CustomIterator is **logically linked** only to CustomList, so it makes sense to keep it as an **inner class**.

**5 Improving Code Readability with Local Inner Classes**

* **Use a local inner class when you need a helper class for a specific method only.**
* Example: **String reversal inside a method**

class StringUtil {

public void reverseString(String input) {

// Local Inner Class

class ReverseHelper {

String reverse() {

return new StringBuilder(input).reverse().toString();

}

}

ReverseHelper helper = new ReverseHelper();

System.out.println("Reversed: " + helper.reverse());

}

}

public class Main {

public static void main(String[] args) {

new StringUtil().reverseString("Java");

}

}

**Application**: ReverseHelper is **only needed inside reverseString()**, so defining it outside would be unnecessary.

**When to Use Nested Classes?**

When a **class is closely related** to another class.  
To **hide implementation details** and enforce **encapsulation**.  
To create **event handlers** in GUI frameworks (like Swing & Android).  
To **implement iterators** inside custom collections.  
To improve **code readability** and **organization**.