Nested classes in Java

give me the deep and detailed explanation of Nested Classes in java so that i will understand it well
so that do not have any doubt in this

so give me the deep and detailed explanation of this

Let's go step-by-step and make *nested classes* crystal clear.
I'll explain *what* they are, the *types*, *how they work*, and *where to use* them — with simple examples.

---

1. What is a Nested Class in Java?

Definition:
A **nested class** is a class that is defined **inside another class**.

```java
class Outer {
    class Inner {
        // this is a nested class (inner class)
    }
}
```

- `Outer` → outer class (also called *enclosing class*).
- `Inner` → nested class (since it is inside `Outer`).

Why does Java have nested classes?

1. **Logical grouping**
   If a class is used only by one other class, we can keep it inside that class.
2. **Better encapsulation**
   A nested class can be `private`, `protected`, etc. So we can hide implementation details.
3. **Readability**
   Related code stays together. You don't pollute the package with too many separate top-level classes.
4. **Can access outer class members** (for inner classes)
   Non-static inner classes can directly use outer class fields (even private ones).

---

2. Types of Nested Classes in Java

There are **two big categories**:

1. **Static nested classes**
2. **Non-static nested classes (Inner classes)**
   Inner classes are again of three types:
   - Member inner class (normal inner class)
   - Local inner class
   - Anonymous inner class

Let's go through them one by one.

---

3. Static Nested Class

3.1. What is a Static Nested Class?

A **static nested class** is a `static` class defined inside another class.

```java
class Outer {

    static class StaticNested {
        void display() {
            System.out.println("Inside Static Nested Class");
        }
    }
}
```

Key points:

- It is like a **normal top-level class**, but logically grouped inside `Outer`.
- It **cannot access non-static members** of `Outer` directly.
- It can access **only static members** of `Outer` directly.
- It does **not** need an instance of `Outer` to create its object.

3.2. How to create an object of Static Nested Class?

```java
Outer.StaticNested obj = new Outer.StaticNested();
obj.display();
```

Notice:

- No `new Outer()` required.
- We use `Outer.StaticNested` for fully qualified name.

3.3. Access rules

```java
class Outer {
    private static int x = 10;
    private int y = 20;

    static class StaticNested {
        void show() {
            System.out.println("x = " + x);  // OK: x is static
            // System.out.println("y = " + y); // ERROR: cannot access non-static y
        }
    }
}
```

Static nested class behaves like a **static member** of `Outer`.

3.4. When to use Static Nested Class?

Use it when:

- Nested class **does not need** access to instance (non-static) members of outer.
- You just want to **group** code logically.
- Often used for things like:
  - Helper/builder classes
  - Enums inside classes
  - `Map.Entry` in `HashMap` (implementation detail)

---

4. Member Inner Class (Non-static inner class)

4.1. What is a Member Inner Class?

A **member inner class** is a **non-static** class defined inside another class **directly** (not inside a method).

```java
class Outer {

    private int x = 10;

    class Inner { // member inner class
        void show() {
            System.out.println("x = " + x); // can access outer's members
        }
    }
}
```

Key points:

- It is **associated with an instance** of the outer class.
- It **can access all members** of `Outer` (including `private`, `protected`, etc.).
- To create an object of `Inner`, you need an object of `Outer`.

4.2. How to create an Inner Class object?

```java
public class Test {
    public static void main(String[] args) {
        Outer outer = new Outer();          // 1. Outer object
        Outer.Inner inner = outer.new Inner(); // 2. Inner object using outer
        inner.show();
    }
}
```

Syntax:

```java
Outer.Inner innerRef = outerObject.new Inner();
```

4.3. How Inner Class accesses outer members?

```java
class Outer {
    private int x = 10;
    int y = 20;

    class Inner {
        int y = 30;

        void display() {
            System.out.println("Inner y = " + y);           // 30 (inner's y)
            System.out.println("Outer y = " + Outer.this.y); // 20 (outer's y)
            System.out.println("Outer x = " + x);            // 10 (outer's x)
        }
    }
}
```

Important:

- `Outer.this` → reference to outer class object from inside inner class.
- If variable names clash, use `Outer.this.varName` to refer to outer's variable.

4.4. Modifiers allowed on Member Inner Class

You can use almost all access modifiers:

```java
class Outer {
    // These are all valid
    public class PublicInner {}
    private class PrivateInner {}
    protected class ProtectedInner {}
    class DefaultInner {}
    static class StaticNested {} // static → static nested class, not inner
}
```

Also, inner classes can be `final`, `abstract`, etc.

---

5. Local Inner Class (Class inside a method)

5.1. What is a Local Inner Class?

A **local inner class** is a class defined **inside a method**, **constructor**, or **block**.

```java
class Outer {
    void outerMethod() {
        class LocalInner {
            void display() {
                System.out.println("Inside Local Inner Class");
            }
        }

        LocalInner obj = new LocalInner();
        obj.display();
    }
}
```

Key points:

- It is **local to the method**. You cannot use it **outside** that method.
- It behaves like a **local variable with a class definition**.
- Cannot have access modifiers like **`public`**, **`private`**, **`protected`**, or **`static`** (except for static final constants).

5.2. Access rules (local variables)

Important concept: **Effectively final**

Local inner class can access:

- All members (fields, methods) of outer class.
- Local variables of the method **only if they are effectively final**.

```java
class Outer {
    void outerMethod() {
        int a = 10;            // effectively final if not changed
        int b = 20;

        class LocalInner {
            void show() {
                System.out.println("a = " + a); // OK
                // b also OK if not modified later
            }
        }

        b = 30; // now b is NOT effectively final → cannot be used inside LocalInner
    }
}
```

Why effectively final?
Because the local variable actually gets copied into the inner class object. To avoid confusion and inconsistent values, it must not change after being captured.

5.3. When to use Local Inner Class?

- When the class is **needed only in one method**.
- For **small helpers** inside complex methods.
- Often used with **event handling**, small utilities, etc.

---

6. Anonymous Inner Class

6.1. What is an Anonymous Inner Class?

An **anonymous inner class** is:

- A **class without a name**.
- Created at the same time as its object.
- Used to **override methods on the fly**, usually for:
  - Interfaces
  - Abstract classes
  - Concrete classes (sometimes)

Example with an interface:

```java
interface Animal {
    void makeSound();
}

public class Test {
    public static void main(String[] args) {
        Animal dog = new Animal() {
            @Override
            public void makeSound() {
                System.out.println("Woof! Woof!");
            }
        };

        dog.makeSound();
    }
}
```

Here:

- We didn't create a named class like `Dog`.
- We created an **anonymous class** implementing `Animal` and overriding `makeSound()`.

6.2. Anonymous Inner Class with Abstract Class

```java
abstract class Person {
    abstract void greet();
}

public class Test {
    public static void main(String[] args) {
        Person p = new Person() {
            @Override
            void greet() {
                System.out.println("Hello from anonymous inner class");
            }
        };

        p.greet();
    }
}
```

6.3. Anonymous Inner Class with Concrete Class

```
java

class Hello {
    void say() {
        System.out.println("Hello");
    }
}

public class Test {
    public static void main(String[] args) {
        Hello h = new Hello() {
            @Override
            void say() {
                System.out.println("Hello, overridden in anonymous class");
            }
        };

        h.say();
    }
}
```

6.4. When to use Anonymous Inner Classes?

- When you need a **one-time-use** class.
- When you want to **quickly override** methods without creating a separate `.java` file.
- Common in:
  - GUI code (Swing, Android: e.g., button click listeners),
  - Small callbacks,
  - Comparators (`new Comparator<>() { ... }`) before Java 8 lambdas.

---

7. Summary of All Types (Comparison)

Type	Static?	Where defined?	Needs Outer object?	Can access outer instance members?	Common usage
Static Nested Class	Yes	Inside class	❌ No	❌ No (only static)	Helper classes, grouping logic
Member Inner Class	No	Inside class	✅ Yes	✅ Yes	When logic needs outer's instance data
Local Inner Class	No	Inside method/block	✅ Yes (implicitly)	✅ Yes, and effectively final locals	Helper classes used only inside a method

Type	Static?	Where defined?	Needs Outer object?	Can access outer instance members?	Common usage
Anonymous Inner Class	No	At object creation site	✅ Yes (if non-static)	✅ Yes (if inner)	One-time implementations, callbacks

---

8. Access Modifiers and Static Rules

8.1. What can be `static` inside what?

- **Static nested class** → allowed (`static class Nested {}`).
- **Inner classes (non-static)** → cannot be declared `static`.
- Inside inner classes, you **cannot** have static members **unless they are static final constants**.

```
java

class Outer {
    class Inner {
        // static int x;            // ❌ Not allowed
        static final int X = 10;   // ✅ Allowed (constant)
    }
}
```

8.2. Access modifiers for nested types

- **Member inner classes** and **static nested classes**:
  - Can be `public`, `private`, `protected`, or default.
- **Local inner classes** and **anonymous inner classes**:
  - Cannot have `public/private/protected`.
  - They are local to the method/block.

---

9. Shadowing (same variable name in inner & outer)

If inner class has a variable with the same name as outer class:

```
java

class Outer {
    int x = 10;

    class Inner {
        int x = 20;

        void show() {
            int x = 30;

            System.out.println(x);           // 30 (local variable)
            System.out.println(this.x);      // 20 (Inner's x)
            System.out.println(Outer.this.x);// 10 (Outer's x)
        }
    }
}
```

- `x` → nearest scope (local variable).
- `this.x` → inner class's field.
- `Outer.this.x` → outer class's field.

---

10. Real-Life Use Cases

10.1. Builder pattern using static nested class

```java
class User {
    private String name;
    private int age;

    private User(Builder builder) {
        this.name = builder.name;
        this.age  = builder.age;
    }

    static class Builder {
        private String name;
        private int age;

        Builder setName(String name) {
            this.name = name;
            return this;
        }

        Builder setAge(int age) {
            this.age = age;
            return this;
        }

        User build() {
            return new User(this);
        }
    }
}

class Test {
    public static void main(String[] args) {
        User u = new User.Builder()
                        .setName("Piyush")
                        .setAge(21)
                        .build();
    }
}
```

Here:

- `Builder` is a **static nested class**.
- It does not need access to outer instance, but just helps construct it.

10.2. Event handling with anonymous inner class

```java
button.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View v) {
        System.out.println("Button clicked");
    }
});
```

Instead of creating a separate class, we use an **anonymous inner class**.

---

11. Memory & Design Considerations

- Non-static inner classes hold an **implicit reference** to the outer object.
  So if you store inner objects for a long time, outer objects also stay in memory → possible memory leak if not used carefully.
- Static nested classes **do not** hold such reference, so they are lighter in that sense.
- If nested class **does not need** outer instance → prefer **static nested class**.

---

12. Very Short Recap

- **Nested class** = class inside another class.
- Two main types:
  - **Static nested class** → uses `static`, no outer instance needed.
  - **Inner classes** (non-static):
    - Member inner class → normal inner class.
    - Local inner class → inside a method.
    - Anonymous inner class → no name, created on the spot.
- Inner classes can access **all outer members**, including private.
- Local inner + anonymous inner can use method's local variables only if they are **effectively final**.
- Use nested classes for:
  - Grouping related classes
  - Hiding implementation details
  - Small helper / listener / callback classes

If you want, next I can:

- Give you **MCQ-style questions** on nested classes and then explain answers, or
- Give you **coding exercises** where you write your own inner classes and I'll check and correct them.