Polymorphism and `final` Keyword in Java – Pin-to-Pin Complete Notes

---

1. Introduction (From Scratch)

Java is an **object-oriented language**, and two of its most powerful pillars are:

- **Inheritance** – reusing code
- **Polymorphism** – same reference, different behavior at runtime

To control inheritance and polymorphism, Java provides the **`final` keyword**.

Understanding **why we need parent references**, **why upcasting exists**, and **why downcasting is sometimes required** is the core of mastering Java OOP.

---

2. `final` Keyword – Complete Explanation

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

1. Class
2. Method
3. Variable

---

3. `final` Class

```
final class Vehicle {
    void move() {
        System.out.println("Vehicle moving");
    }
}
```

Rules

- A **final class cannot be inherited**
- It **cannot be a parent class**
- Prevents inheritance completely

```
class Car extends Vehicle { } // □ ERROR
```

Why final class is needed?

- Security (e.g., `String` class)
- Prevent behavior change
- Design decision: "This class is complete"

Real example: `String`, `Math`, `Wrapper classes`

---

4. `final` Method

```
class Aeroplane {
    final void takingOff() {
        System.out.println("Aeroplane taking off");
    }
}
```

Rules

- **Final methods ARE inherited**
- **Cannot be overridden**

```
class CargoPlane extends Aeroplane {
    // void takingOff() { }  ERROR
}
```

Final methods CANNOT be overridden.

Why final method?

- Lock critical logic
- Avoid accidental override
- Improve readability & safety

---

5. `final` Variable

```
final int x = 10;
```

Rules

- Value **cannot be changed**
- Must be initialized

```
x = 20; //  ERROR
```

`final` with static

```
static final double PI = 3.14;
```

- Becomes **constant**
- Stored in **method area**

---

6. `final` + Inheritance Summary Table

Usage	Inherited?	Overridden?
final class	□	□
final method	□	□
final variable	□ (value copied)	□
static final	□	□
private method	□	□
constructor	□ final not allowed	

---

7. `final` Cannot Be Applied To

- □ Constructor (constructors are not inherited)
- □ Abstract methods

---

8. Polymorphism – Core Concept

Definition

Polymorphism means one reference behaving in many forms at runtime.

```
Aeroplane a = new CargoPlane();
```

Same reference → different behavior

9. Upcasting (VERY IMPORTANT)

```
Aeroplane a = new CargoPlane();
```

What is Upcasting?

- Creating **parent class reference** pointing to **child object**
- Happens **implicitly**

Why do we need upcasting?

1 Runtime Polymorphism

```
a.landing(); // calls CargoPlane version
```

2 Loose coupling

- Code depends on parent, not child

3 Scalability

```
Aeroplane a;
a = new CargoPlane();
a = new PassengerPlane();
```

Same code → multiple behaviors

4 Industry-level design

Frameworks, APIs, collections ALL use parent references

---

10. Method Execution Rules

```
class Aeroplane {
    void takingOff() {}
    void landing() {}
}

class CargoPlane extends Aeroplane {
    void landing() {} // overridden
    void flyCargo() {}
}
```

Case 1: Parent reference

```
Aeroplane cp = new CargoPlane();
cp.takingOff(); // inherited
cp.landing();   // overridden (child version)
```

Case 2: Child reference

```
CargoPlane cp = new CargoPlane();
cp.takingOff();
cp.landing();
```

✔ Output is SAME for overridden methods

---

11. Then WHY Parent Reference?

Very important doubt □

"If both parent and child reference give same output, why use parent reference?"

Answer:

Because **polymorphism works ONLY with parent reference**

```
Aeroplane a = new CargoPlane(); // polymorphism
CargoPlane c = new CargoPlane(); // NO polymorphism
```

Without parent reference:

- No runtime binding
- No flexibility
- No dynamic behavior

---

12. Specialized Methods Problem

```
Aeroplane a = new CargoPlane();
a.flyCargo(); // □ ERROR
```

Why error?

- Reference type = Aeroplane
- Compiler checks reference, not object

---

13. Downcasting

```
((CargoPlane) a).flyCargo();
```

What is Downcasting?

- Temporarily converting parent reference → child reference
- Used to access **specialized methods**

Important points

- Happens **explicitly**
- Risky (may cause `ClassCastException`)

```
Aeroplane a = new PassengerPlane();
((CargoPlane) a).flyCargo(); // □ Runtime error
```

---

14. Safe Downcasting (Best Practice)

```
if (a instanceof CargoPlane) {
    ((CargoPlane) a).flyCargo();
}
```

---

15. Final + Polymorphism Relationship

Feature	Effect
final method	Stops runtime polymorphism
final class	Stops inheritance completely
final variable	Constant behavior

---

16. Key Interview Lines (MEMORIZE)

- "Polymorphism works through parent reference"
- "Method execution depends on object, not reference"
- "Compiler checks reference, JVM checks object"
- "Final methods are inherited but not overridden"
- "Downcasting is used to access specialized behavior"

---

17. Final Summary

✔ `final` controls modification
✔ Parent reference enables polymorphism
✔ Upcasting = flexibility + scalability
✔ Downcasting = access specialization
✔ Child reference = no polymorphism