Runtime Polymorphism in Java (Method Overriding / Dynamic Dispatch)

Definition: Runtime polymorphism happens when a method call is resolved at runtime based on the actual object type,
not the reference type. It is achieved via method overriding.

Key Points:

- Achieved through method overriding (same signature in parent and child).
- Decision made by JVM at runtime (dynamic method dispatch).
- Requires inheritance and typically upcasting (Parent ref -> Child object).
- @Override annotation helps catch signature mismatches at compile time.
- Access modifiers: overriding method cannot reduce visibility.
- Final/static/private methods are not overridden (static/private are hidden).
- Return type can be covariant (child type) in overriding.
- Exceptions: child method cannot throw broader checked exceptions than parent.

Code Example

```
class Animal {
    void sound() {
        System.out.println("Animal makes sound");
    }
}
class Dog extends Animal {
    @Override
    void sound() {
        System.out.println("Dog barks");
    }
}
class Cat extends Animal {
    @Override
    void sound() {
        System.out.println("Cat meows");
    }
}
public class RuntimePolymorphismDemo {
    public static void main(String[] args) {
        Animal a;
        a = new Dog();
        a.sound();   // Dog barks
        a = new Cat();
        a.sound();   // Cat meows
    }
}
```

Notes & Tips

Output:
Dog barks
Cat meows
Why it’s Runtime Polymorphism:
• The reference type is Animal, but the actual object is Dog or Cat at runtime.
• The JVM picks Dog.sound() or Cat.sound() dynamically (late binding).
Quick Tips:
• Prefer programming to abstractions: use interface/abstract class references for flexibility.
• Mark methods final in base class only when you want to prevent overriding.
• Use @Override in child methods to avoid accidental overloading.
Overloading vs Overriding (1-liner):
• Overloading = compile-time (same name, different params).
• Overriding = runtime (same signature in subclass).