Polymorphism in Object-Oriented Programming

Polymorphism in Object-Oriented Programming (OOP) is the ability of different classes to respond to the same method call in different ways. It allows one interface to be used for a general class of actions, making code more flexible and extensible.

Key Ideas

- "Many forms": The word *polymorphism* means having many forms.
- Objects of different classes can be treated through the same interface.
- The specific behavior depends on the object's actual class.

Types of Polymorphism

1. Compile-Time Polymorphism

Also called Static Polymorphism, Early Binding, or Method Overloading.

- Resolved at compile time.
- Same method name with different parameters.
- Common in statically-typed languages like Java or C++.

Example (C++)

```
// Function Overloading
int add(int a, int b) {
    return a + b;
}

float add(float a, float b) {
    return a + b;
}
```

---

2. **Run-Time Polymorphism**

Also called Dynamic Polymorphism, Late Binding, or Method Overriding.

- Resolved at run time.
- A subclass provides a specific implementation of a method already defined in its superclass.
- Achieved with virtual functions (C++), overriding (Java), or dynamic dispatch (Python).

Example (Python – Runtime Polymorphism)

```python
class Animal:
    def speak(self):
        print("Animal speaks")

class Dog(Animal):
    def speak(self):
        print("Dog barks")

class Cat(Animal):
    def speak(self):
        print("Cat meows")

# Same interface, different behavior
def make_animal_speak(animal):
    animal.speak()

make_animal_speak(Dog())  # Output: Dog barks
make_animal_speak(Cat())  # Output: Cat meows
```

Why Polymorphism Matters

Benefit	Description
Flexibility	Swap implementations without changing calling code.
Maintainability	Reduce code duplication and isolate changes.
Open/Closed Principle	Open for extension, closed for modification.
Scalability	Add new subclasses with minimal impact on existing code.

Quick Comparison

Feature	Compile-Time (Overloading)	Run-Time (Overriding)
Binding Time	Compile-time	Run-time
Mechanism	Different signatures	Inheritance + virtual
Language Keywords	N/A	`virtual`, `override`
Example	`add(int)`, `add(float)`	`Animal.speak()`

Method Overloading & Overriding:

---

1. Method Overloading

What is it?

Method Overloading is a form of compile-time polymorphism where multiple methods in the same class share the same name but differ in number or type of parameters.

Key Features:

Same method name with different parameter lists. Can differ by: Number of parameters Type of parameters Order of parameters

Helps in code readability and reusability by logically grouping similar actions

Why Use Method Overloading?

To perform similar operations with different inputs. Makes the code more readable and easier to maintain.

Examples & Diagram

```
class Calculator {
    // Method 1: two ints
    int add(int a, int b) {
        return a + b;
    }

    // Method 2: two doubles
    double add(double a, double b) {
        return a + b;
    }

    // Method 3: three ints
    int add(int a, int b, int c) {
        return a + b + c;
    }
}
```

> **Compile-Time Errors** you might see at this stage:
>
> - Syntax errors
> - Type-mismatched assignments
> - Undeclared variables

---

2. Method Overriding

What is it?

Method Overriding occurs when a subclass provides a specific implementation of a method that is already defined in its superclass. This is a form of runtime polymorphism, meaning the method that gets called is determined during program execution.

Key Features:

The method name, parameters, and return type must be the same in both the parent and child classes. Only the method body (behavior) is changed in the subclass. Used to provide specialized behavior in the child class. The parent class method is overridden (replaced) when called through an object of the child class.

Why Use Method Overriding?

To customize behavior in subclasses. To follow polymorphic design, where a single interface can represent different types of behavior. To make code more extensible and reusable.

Examples & Diagram

```
class Animal {
    void sound() {
        System.out.println("Animal makes sound");
    }
}

class Dog extends Animal {
    @Override
    void sound() {
        System.out.println("Dog barks");
    }
}
```

Quick Comparison

Feature	Method Overloading	Method Overriding
Class Scope	Same class	Child class
Signature	Must differ	Must be identical
Binding Time	Compile-time	Run-time
Purpose	Convenience / readability	Polymorphic behavior
Inheritance	Not required	Required

> **Tip:**
> Overloading is **compile-time polymorphism**; Overriding is **run-time polymorphism**.

Virtual Functions in C++

---

1. **What is a Virtual Function?**

A virtual function is a member function in a base class that can be redefined (overridden) in derived classes while preserving the same signature.

- Enables dynamic (run-time) polymorphism.
- Guarantees that the correct method is called for an object, regardless of the type of pointer/reference used.

---

2. **Why Virtual Functions?**

Without `virtual`, calling a method through a base-class pointer will always invoke the base-class version, even if the object is actually of a derived type.

With `virtual`, the object's actual type (not the pointer's type) determines which method runs — this is called late / dynamic binding.

---

3. **Code Example**

```
#include <iostream>
using namespace std;

// Base class
class Base {
public:
    void show() {                       // Non-virtual
        cout << "Base show" << endl;
    }

    virtual void speak() {              // Virtual function
        cout << "Base speak" << endl;
    }
};

// Derived class
class Derived : public Base {
public:
    void show() {                       // Hides Base::show
        cout << "Derived show" << endl;
    }

    void speak() override {             // Overrides Base::speak
        cout << "Derived speak" << endl;
    }
};

int main() {
    Base* b = new Derived();

    b->show();    // Output: Base show   (static binding)
    b->speak();   // Output: Derived speak (dynamic binding)
```

```
    delete b;
    return 0;
}
```

---

4. **Key Takeaways**

Term	Meaning
`virtual`	Tells the compiler to use dynamic binding
Override	Provides a new implementation in the derived class
Base pointer	Can point to any derived object, yet call the correct virtual method

Use virtual functions whenever you expect to manipulate derived-class objects through base-class pointers/references and need run-time polymorphism.